CSCI567 Machine Learning (Fall 2025)

Haipeng Luo

University of Southern California

Sep 19, 2025

Administration

Will discuss HW1 solutions in today discussion session.

HW2 will be released next week.

Outline

1. Review of Last Lecture
2. Multiclass Classification
3. Kernel methods

Outline

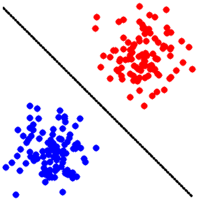
1. Review of Last Lecture
2. Multiclass Classification
3. Kernel methods

Linear classifiers

Linear models for **binary** classification:

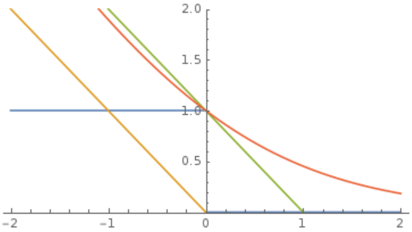
Step 1. Model is the set of **separating hyperplanes**

$$\mathcal{F} = \{f(\boldsymbol{x}) = \mathsf{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) \mid \boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}\}$$



Linear classifiers

Step 2. Pick the **surrogate loss**



- perceptron loss $\ell_{\text{perceptron}}(z) = \max\{0, -z\}$ (used in Perceptron)
- hinge loss $\ell_{\text{hinge}}(z) = \max\{0, 1-z\}$(used in SVM and many others)
- logistic loss $\ell_{\text{logistic}}(z) = \log(1 + \exp(-z))$ (used in logistic regression)

Linear classifiers

Step 3. Find empirical risk minimizer (ERM):

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}} F(\boldsymbol{w}) = \operatorname*{argmin}_{\boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n)$$

using

- **GD:** $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla F(\boldsymbol{w})$
- **SGD:** $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \tilde{\nabla} F(\boldsymbol{w})$ $\quad (\mathbb{E}[\tilde{\nabla} F(\boldsymbol{w})] = \nabla F(\boldsymbol{w}))$
- **Newton:** $\boldsymbol{w} \leftarrow \boldsymbol{w} - \left(\nabla^2 F(\boldsymbol{w})\right)^{-1} \nabla F(\boldsymbol{w})$

Convergence guarantees of GD/SGD

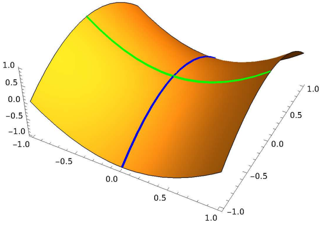
- GD/SGD converges to a stationary point

Convergence guarantees of GD/SGD

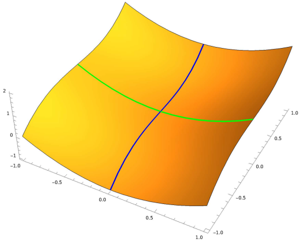
- GD/SGD converges to a stationary point
- for convex objectives, this is all we need

Convergence guarantees of GD/SGD

- GD/SGD converges to a stationary point
- for convex objectives, this is all we need
- for nonconvex objectives, can get stuck at local minimizers or “bad” saddle points (random initialization escapes “good” saddle points)



“good” saddle points



“bad” saddle points

Perceptron and logistic regression

Initialize $\boldsymbol{w} = \mathbf{0}$ or randomly.

Repeat:

- pick a data point $\boldsymbol{x}_n$ uniformly at random (**common trick for SGD**)

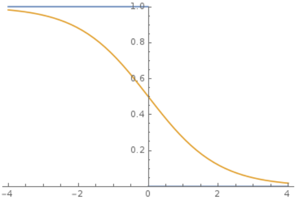
Perceptron and logistic regression

Initialize $\boldsymbol{w} = \boldsymbol{0}$ or randomly.

Repeat:

- pick a data point $\boldsymbol{x}_n$ uniformly at random (**common trick for SGD**)
- update parameter:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \begin{cases} \mathbb{I}[y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n \leq 0] y_n \boldsymbol{x}_n & \text{(Perceptron)} \\ \eta \sigma(-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) y_n \boldsymbol{x}_n & \text{(logistic regression)} \end{cases}$$



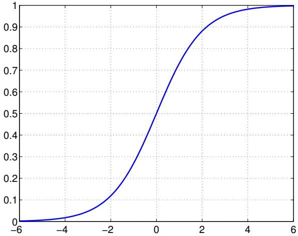
A Probabilistic view of logistic regression

Minimizing logistic loss = MLE for the sigmoid model

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} \sum_{n=1}^{N} \ell_{\text{logistic}}(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) = \operatorname*{argmax}_{\boldsymbol{w}} \prod_{n=1}^{N} \mathbb{P}(y_n \mid \boldsymbol{x_n}; \boldsymbol{w})$$

where

$$\mathbb{P}(y \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(y \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}) = \frac{1}{1 + e^{-y \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}}}$$



Outline

1. Review of Last Lecture
2. Multiclass Classification
   - Multinomial logistic regression
   - Reduction to binary classification
3. Kernel methods

Classification

Recall the setup:

- input (feature vector): $\boldsymbol{x} \in \mathbb{R}^{\mathsf{D}}$
- output (label): $y \in [\mathsf{C}] = \{1, 2, \cdots, \mathsf{C}\}$
- goal: learn a mapping $f : \mathbb{R}^{\mathsf{D}} \rightarrow [\mathsf{C}]$

Classification

Recall the setup:

- input (feature vector): $\boldsymbol{x} \in \mathbb{R}^{\mathsf{D}}$
- output (label): $y \in [\mathsf{C}] = \{1, 2, \cdots, \mathsf{C}\}$
- goal: learn a mapping $f : \mathbb{R}^{\mathsf{D}} \to [\mathsf{C}]$

Examples:

- recognizing digits ($\mathsf{C} = 10$) or letters ($\mathsf{C} = 26$ or 52)
- predicting weather: sunny, cloudy, rainy, etc
- predicting image category: ImageNet dataset ($\mathsf{C} \approx 20K$)

Classification

Recall the setup:

- input (feature vector): $\boldsymbol{x} \in \mathbb{R}^{\mathsf{D}}$
- output (label): $y \in [\mathsf{C}] = \{1, 2, \cdots, \mathsf{C}\}$
- goal: learn a mapping $f : \mathbb{R}^{\mathsf{D}} \to [\mathsf{C}]$

Examples:

- recognizing digits ($\mathsf{C} = 10$) or letters ($\mathsf{C} = 26$ or 52)
- predicting weather: sunny, cloudy, rainy, etc
- predicting image category: ImageNet dataset ($\mathsf{C} \approx 20K$)

Nearest Neighbor Classifier naturally works for arbitrary C.

Linear models: from binary to multiclass

Step 1: *What should a linear model look like for multiclass tasks?*

Linear models: from binary to multiclass

Step 1: *What should a linear model look like for multiclass tasks?*

Note: a linear model for binary tasks (switching from $\{-1, +1\}$ to $\{1, 2\}$)

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0 \\ 2 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0 \end{cases}$$

Linear models: from binary to multiclass

Step 1: *What should a linear model look like for multiclass tasks?*

Note: a linear model for binary tasks (switching from $\{-1, +1\}$ to $\{1, 2\}$)

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0 \\ 2 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0 \end{cases}$$

can be written as

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \geq \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} \\ 2 & \text{if } \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} > \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \end{cases}$$

for any $\boldsymbol{w}_1, \boldsymbol{w}_2$ s.t. $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$

Linear models: from binary to multiclass

Step 1: *What should a linear model look like for multiclass tasks?*

Note: a linear model for binary tasks (switching from $\{-1, +1\}$ to $\{1, 2\}$)

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0 \\ 2 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0 \end{cases}$$

can be written as

$$\begin{aligned} f(\boldsymbol{x}) &= \begin{cases} 1 & \text{if } \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \geq \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} \\ 2 & \text{if } \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} > \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \end{cases} \\ &= \operatorname*{argmax}_{k \in \{1,2\}} \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x} \end{aligned}$$

for any $\boldsymbol{w}_1, \boldsymbol{w}_2$ s.t. $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$

Linear models: from binary to multiclass

Step 1: *What should a linear model look like for multiclass tasks?*

Note: a linear model for binary tasks (switching from $\{-1,+1\}$ to $\{1,2\}$)

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0 \\ 2 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0 \end{cases}$$

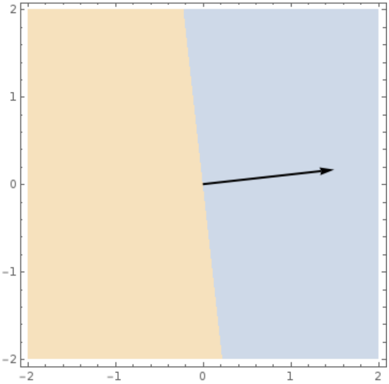
can be written as

$$\begin{aligned} f(\boldsymbol{x}) &= \begin{cases} 1 & \text{if } \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \geq \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} \\ 2 & \text{if } \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} > \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \end{cases} \\ &= \operatorname*{argmax}_{k\in\{1,2\}} \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x} \end{aligned}$$

for any $\boldsymbol{w}_1, \boldsymbol{w}_2$ s.t. $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$

Think of $\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$ as **a score for class k**.

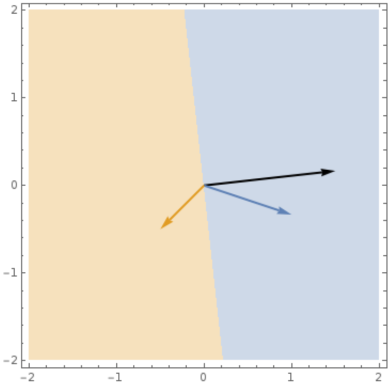
Linear models: from binary to multiclass



$$\boldsymbol{w} = \left(\tfrac{3}{2}, \tfrac{1}{6}\right)$$

- Blue class: $\{\boldsymbol{x} : \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0\}$
- Orange class: $\{\boldsymbol{x} : \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0\}$

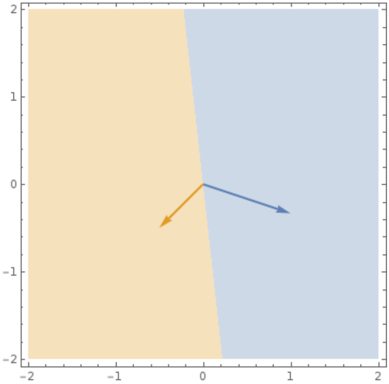
Linear models: from binary to multiclass



$\boldsymbol{w} = (\frac{3}{2}, \frac{1}{6}) = \boldsymbol{w}_1 - \boldsymbol{w}_2$
$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$

- Blue class:
  $\{\boldsymbol{x} : 1 = \mathrm{argmax}_k\, \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$
- Orange class:
  $\{\boldsymbol{x} : 2 = \mathrm{argmax}_k\, \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$

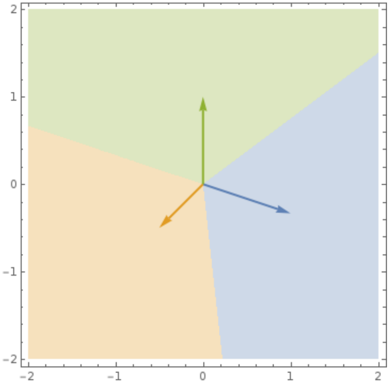
Linear models: from binary to multiclass



$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$

- Blue class:
  $\{\boldsymbol{x} : 1 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$
- Orange class:
  $\{\boldsymbol{x} : 2 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}\}$

Linear models: from binary to multiclass



$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$
$\boldsymbol{w}_3 = (0, 1)$

- Blue class: $\{\boldsymbol{x} : 1 = \mathrm{argmax}_k\, \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$
- Orange class: $\{\boldsymbol{x} : 2 = \mathrm{argmax}_k\, \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$
- Green class: $\{\boldsymbol{x} : 3 = \mathrm{argmax}_k\, \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$

Linear models for multiclass classification

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \underset{k \in [\mathsf{C}]}{\operatorname{argmax}} \ \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{D}} \right\}$$

Linear models for multiclass classification

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k \in [\mathsf{C}]} \; \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{D}} \right\}$$

$$= \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k \in [\mathsf{C}]} \, (\boldsymbol{W}\boldsymbol{x})_k \mid \boldsymbol{W} \in \mathbb{R}^{\mathsf{C} \times \mathsf{D}} \right\}$$

Linear models for multiclass classification

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \underset{k \in [\mathsf{C}]}{\operatorname{argmax}} \ \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{D}} \right\}$$
$$= \left\{ f(\boldsymbol{x}) = \underset{k \in [\mathsf{C}]}{\operatorname{argmax}} \ (\boldsymbol{W}\boldsymbol{x})_k \mid \boldsymbol{W} \in \mathbb{R}^{\mathsf{C} \times \mathsf{D}} \right\}$$

Step 2: *How do we generalize perceptron/hinge/logistic loss?*

Linear models for multiclass classification

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k \in [\mathsf{C}]} \; \boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{D}} \right\}$$
$$= \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k \in [\mathsf{C}]} \; (\boldsymbol{W}\boldsymbol{x})_k \mid \boldsymbol{W} \in \mathbb{R}^{\mathsf{C} \times \mathsf{D}} \right\}$$

Step 2: *How do we generalize perceptron/hinge/logistic loss?*

This lecture: focus on the more popular **logistic loss**

Multinomial logistic regression: a probabilistic view

Observe: for binary logistic regression, with $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$:

$$\mathbb{P}(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}} = \frac{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}}{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}} + e^{\boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}$$

Multinomial logistic regression: a probabilistic view

Observe: for binary logistic regression, with $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$:

$$\mathbb{P}(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}} = \frac{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}}{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}} + e^{\boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}$$

Naturally, for multiclass:

$$\mathbb{P}(y = k \mid \boldsymbol{x}; \boldsymbol{W}) = \frac{e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}}{\sum_{k' \in [\mathsf{C}]} e^{\boldsymbol{w}_{k'}^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$$

Multinomial logistic regression: a probabilistic view

Observe: for binary logistic regression, with $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$:

$$\mathbb{P}(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}} = \frac{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}}{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}} + e^{\boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}$$

Naturally, for multiclass:

$$\mathbb{P}(y = k \mid \boldsymbol{x}; \boldsymbol{W}) = \frac{e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}}{\sum_{k' \in [\mathsf{C}]} e^{\boldsymbol{w}_{k'}^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$$

Important operator: *softmax function* (or really, "softargmax")

For a vector $s \in \mathbb{R}^{\mathsf{C}}$, softmax$(s) = \left(\frac{e^{s_1}}{\sum_{k \in [\mathsf{C}]} e^{s_k}}, \cdots, \frac{e^{s_{\mathsf{C}}}}{\sum_{k \in [\mathsf{C}]} e^{s_k}} \right)$

Multinomial logistic regression: a probabilistic view

Observe: for binary logistic regression, with $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$:

$$\mathbb{P}(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}} = \frac{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}}{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}} + e^{\boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}$$

Naturally, for multiclass:

$$\mathsf{softmax}(\boldsymbol{W}\boldsymbol{x})_k = \mathbb{P}(y = k \mid \boldsymbol{x}; \boldsymbol{W}) = \frac{e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}}{\sum_{k' \in [\mathsf{C}]} e^{\boldsymbol{w}_{k'}^{\mathrm{T}}\boldsymbol{x}}} \propto e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}$$

Important operator: *softmax function* (or really, "softargmax")

For a vector $s \in \mathbb{R}^{\mathsf{C}}$, $\mathsf{softmax}(s) = \left(\frac{e^{s_1}}{\sum_{k \in [\mathsf{C}]} e^{s_k}}, \cdots, \frac{e^{s_{\mathsf{C}}}}{\sum_{k \in [\mathsf{C}]} e^{s_k}} \right)$

Applying MLE again

Maximize probability of seeing labels $y_1, \ldots, y_\mathsf{N}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\mathsf{N}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}$$

Applying MLE again

Maximize probability of seeing labels $y_1, \ldots, y_{\mathsf{N}}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\mathsf{N}}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}$$

By taking **negative log**, this is equivalent to minimizing

$$F(\boldsymbol{W}) = \sum_{n=1}^{\mathsf{N}} \ln \left(\frac{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}} \right)$$

Applying MLE again

Maximize probability of seeing labels $y_1, \ldots, y_\mathsf{N}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\mathsf{N}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}$$

By taking **negative log**, this is equivalent to minimizing

$$F(\boldsymbol{W}) = \sum_{n=1}^{\mathsf{N}} \ln \left(\frac{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}} \right) = \sum_{n=1}^{\mathsf{N}} \ln \left(1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n} \right)$$

Applying MLE again

Maximize probability of seeing labels $y_1, \ldots, y_{\mathsf{N}}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\mathsf{N}}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}$$

By taking **negative log**, this is equivalent to minimizing

$$F(\boldsymbol{W}) = \sum_{n=1}^{\mathsf{N}} \ln\left(\frac{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}\right) = \sum_{n=1}^{\mathsf{N}} \ln\left(1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}\right)$$

This is the *multiclass logistic loss*, a.k.a. *cross-entropy loss*.

Applying MLE again

Maximize probability of seeing labels $y_1, \ldots, y_\mathsf{N}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\mathsf{N}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}$$

By taking **negative log**, this is equivalent to minimizing

$$F(\boldsymbol{W}) = \sum_{n=1}^{\mathsf{N}} \ln \left(\frac{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}} \right) = \sum_{n=1}^{\mathsf{N}} \ln \left(1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n} \right)$$

This is the *multiclass logistic loss*, a.k.a. *cross-entropy loss*.

When $\mathsf{C} = 2$, this is the same as binary logistic loss.

Step 3: Optimization

Apply **SGD**: what is the gradient of

$$F_n(\boldsymbol{W}) = \ln\left(1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}\right)?$$

Step 3: Optimization

Apply **SGD**: what is the gradient of

$$F_n(\boldsymbol{W}) = \ln\left(1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}\right)?$$

It's a C $\times$ D matrix. Let's focus on the k-th row:

Step 3: Optimization

Apply **SGD**: what is the gradient of

$$F_n(\boldsymbol{W}) = \ln\left(1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)?$$

It's a C $\times$ D matrix. Let's focus on the k-th row:

If $k \neq y_n$:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}}$$

Step 3: Optimization

Apply **SGD**: what is the gradient of

$$F_n(\boldsymbol{W}) = \ln\left(1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}\right)?$$

It's a C $\times$ D matrix. Let's focus on the k-th row:

If $k \neq y_n$:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} = \mathbb{P}(k \mid \boldsymbol{x}_n; \boldsymbol{W}) \boldsymbol{x}_n^{\mathrm{T}}$$

Step 3: Optimization

Apply **SGD**: what is the gradient of

$$F_n(\boldsymbol{W}) = \ln\left(1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)?$$

It's a C $\times$ D matrix. Let's focus on the k-th row:

If $k \neq y_n$:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} = \mathbb{P}(k \mid \boldsymbol{x}_n; \boldsymbol{W})\boldsymbol{x}_n^{\mathrm{T}}$$

else:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{-\left(\sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}}$$

Step 3: Optimization

Apply **SGD**: what is the gradient of

$$F_n(\boldsymbol{W}) = \ln\left(1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)?$$

It's a C $\times$ D matrix. Let's focus on the k-th row:

If $k \neq y_n$:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} = \mathbb{P}(k \mid \boldsymbol{x}_n; \boldsymbol{W})\boldsymbol{x}_n^{\mathrm{T}}$$

else:

$$\nabla_{\boldsymbol{w}_k^{\mathrm{T}}} F_n(\boldsymbol{W}) = \frac{-\left(\sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)}{1 + \sum_{k' \neq y_n} e^{(\boldsymbol{w}_{k'} - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} = \left(\mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) - 1\right)\boldsymbol{x}_n^{\mathrm{T}}$$

SGD for multinomial logistic regression

Initialize $\boldsymbol{W} = \mathbf{0}$ (or randomly). Repeat:

1. pick $n \in [\mathsf{N}]$ uniformly at random
2. update the parameters

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \begin{pmatrix} \mathbb{P}(y = 1 \mid \boldsymbol{x}_n; \boldsymbol{W}) \\ \vdots \\ \mathbb{P}(y = y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) - 1 \\ \vdots \\ \mathbb{P}(y = \mathsf{C} \mid \boldsymbol{x}_n; \boldsymbol{W}) \end{pmatrix} \boldsymbol{x}_n^{\mathrm{T}}$$

SGD for multinomial logistic regression

Initialize $\boldsymbol{W} = \mathbf{0}$ (or randomly). Repeat:

1. pick $n \in [\mathsf{N}]$ uniformly at random
2. update the parameters

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \begin{pmatrix} \mathbb{P}(y = 1 \mid \boldsymbol{x}_n; \boldsymbol{W}) \\ \vdots \\ \mathbb{P}(y = y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) - 1 \\ \vdots \\ \mathbb{P}(y = \mathsf{C} \mid \boldsymbol{x}_n; \boldsymbol{W}) \end{pmatrix} \boldsymbol{x}_n^{\mathrm{T}}$$

Think about why the algorithm makes sense intuitively.

A note on prediction

Having learned $\boldsymbol{W}$, we can either

- make a *deterministic* prediction $\operatorname{argmax}_{k\in[\mathsf{C}]}\ \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$

A note on prediction

Having learned $\boldsymbol{W}$, we can either

- make a *deterministic* prediction $\operatorname{argmax}_{k\in[\mathsf{C}]}\ \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$
- make a *randomized* prediction drawn from $\mathrm{softmax}(\boldsymbol{W}\boldsymbol{x})$

Generalization of cross-entropy loss

Given a general model class:

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \underset{k\in[\mathsf{C}]}{\operatorname{argmax}}\ s(\boldsymbol{x})_k \right\}$$

where $s : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{\mathsf{C}}$ is a **"scoring" function**.

Generalization of cross-entropy loss

Given a general model class:

$$\mathcal{F} = \left\{ f(\boldsymbol{x}) = \underset{k \in [\mathsf{C}]}{\operatorname{argmax}} \; s(\boldsymbol{x})_k \right\}$$

where $s : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{\mathsf{C}}$ is a **"scoring" function**.

The *cross-entropy loss* of f for a training sample $(\boldsymbol{x}, y)$ is

$$-\ln\left(\text{softmax}(s(\boldsymbol{x}))_y\right) = -\ln\left(\frac{e^{s(\boldsymbol{x})_y}}{\sum_{k\in[\mathsf{C}]} e^{s(\boldsymbol{x})_k}}\right) = \ln\left(1 + \sum_{k\neq y} e^{s(\boldsymbol{x})_k - s(\boldsymbol{x})_y}\right)$$

Reduce multiclass to binary

Is there an *even more general and simpler approach* to derive multiclass classification algorithms?

Reduce multiclass to binary

Is there an *even more general and simpler approach* to derive multiclass classification algorithms?

Given a binary classification algorithm (*any one*, not just linear methods), can we turn it to a multiclass algorithm, *in a black-box manner*?

Reduce multiclass to binary

Is there an *even more general and simpler approach* to derive multiclass classification algorithms?

Given a binary classification algorithm (*any one*, not just linear methods), can we turn it to a multiclass algorithm, *in a black-box manner*?

Yes, there are in fact many ways to do it.

- **one-versus-all** (one-versus-rest, one-against-all, etc.)
- **one-versus-one** (all-versus-all, etc.)
- **tree-based reduction**

One-versus-all (OvA)

(picture credit: link)

Idea: train C binary classifiers to learn "**is class k or not?**" for each k.

One-versus-all (OvA)

(picture credit: link)

Idea: train C binary classifiers to learn "**is class k or not?**" for each k.

Training: for each class $k \in [\mathrm{C}]$,

- relabel examples with class k as $+1$, and all others as -1
- train a binary classifier h_k using this new dataset

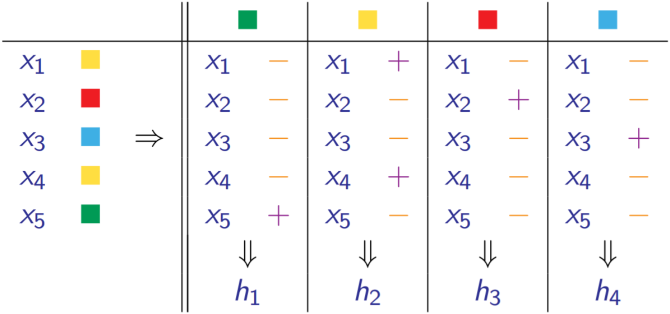
One-versus-all (OvA)

(picture credit: link)

Idea: train C binary classifiers to learn "**is class k or not?**" for each k.

Training: for each class $k \in [\mathrm{C}]$,

- relabel examples with class k as $+1$, and all others as -1
- train a binary classifier h_k using this new dataset



One-versus-all (OvA)

Prediction: for a new example $\boldsymbol{x}$

- ask each h_k: **does this belong to class k?** (i.e. $h_k(\boldsymbol{x})$)

One-versus-all (OvA)

Prediction: for a new example $\boldsymbol{x}$

- ask each h_k: **does this belong to class k?** (i.e. $h_k(\boldsymbol{x})$)
- randomly pick among all k's s.t. $h_k(\boldsymbol{x}) = +1$.

One-versus-all (OvA)

Prediction: for a new example $\boldsymbol{x}$

- ask each h_k: **does this belong to class k?** (i.e. $h_k(\boldsymbol{x})$)
- randomly pick among all k's s.t. $h_k(\boldsymbol{x}) = +1$.

Issue: will (probably) make a mistake *as long as one of h_k errs*.

One-versus-one (OvO)

(picture credit: link)

Idea: train $\binom{\mathsf{C}}{2}$ binary classifiers to learn "**is class k or k'?**".

One-versus-one (OvO)

(picture credit: link)

Idea: train $\binom{\mathsf{C}}{2}$ binary classifiers to learn "**is class k or k'?**".

Training: for each pair (k, k'),

- relabel examples with class k as $+1$ and examples with class k' as -1
- *discard all other examples*
- train a binary classifier $h_{(k,k')}$ using this new dataset

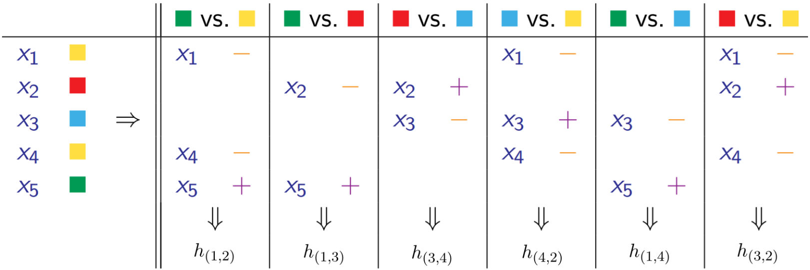
One-versus-one (OvO)

(picture credit: link)

Idea: train $\binom{C}{2}$ binary classifiers to learn "**is class k or k'?**".

Training: for each pair (k, k'),

- relabel examples with class k as $+1$ and examples with class k' as -1
- *discard all other examples*
- train a binary classifier $h_{(k,k')}$ using this new dataset



One-versus-one (OvO)

Prediction: for a new example $\boldsymbol{x}$

- ask each classifier $h_{(k,k')}$ to **vote for either class k or k'**

One-versus-one (OvO)

Prediction: for a new example $\boldsymbol{x}$

- ask each classifier $h_{(k,k')}$ to **vote for either class k or k'**
- predict the class with the most votes (break tie in some way)

One-versus-one (OvO)

Prediction: for a new example $\boldsymbol{x}$

- ask each classifier $h_{(k,k')}$ to **vote for either class k or k'**
- predict the class with the most votes (break tie in some way)

More robust than one-versus-all, but *slower* in prediction.

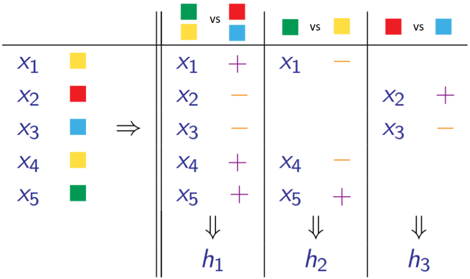
Tree based method

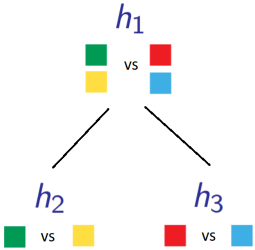
Idea: train $\approx$ C binary classifiers to learn "**belongs to which half?**".

Tree based method

Idea: train $\approx$ C binary classifiers to learn "**belongs to which half?**".

Training: see pictures

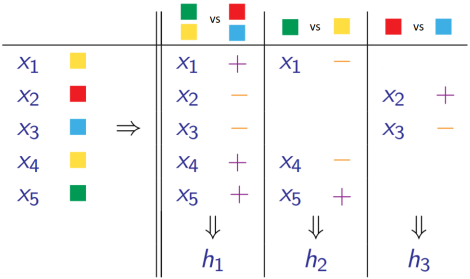


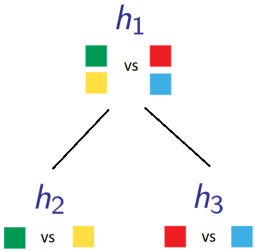


Tree based method

Idea: train $\approx$ C binary classifiers to learn "**belongs to which half?**".

Training: see pictures



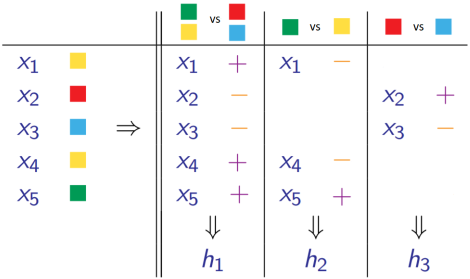


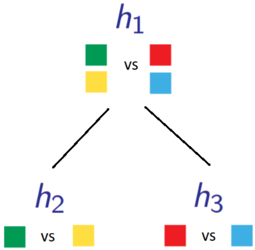
Prediction is also natural,

Tree based method

Idea: train $\approx$ C binary classifiers to learn "**belongs to which half?**".

Training: see pictures





Prediction is also natural, *but is very fast!* (think ImageNet where $C \approx 20K$)

Comparisons

Reduction	**training time**	**prediction time**	**remark**

training time: how many training points are created

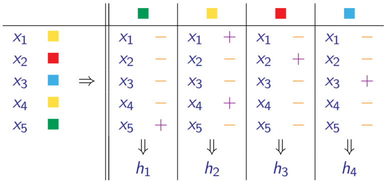
prediction time: how many binary predictions are made

Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA			

training time: how many training points are created

prediction time: how many binary predictions are made

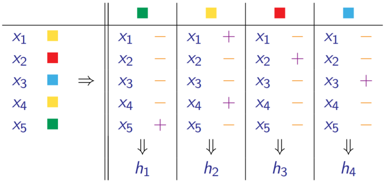


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN		

training time: how many training points are created

prediction time: how many binary predictions are made

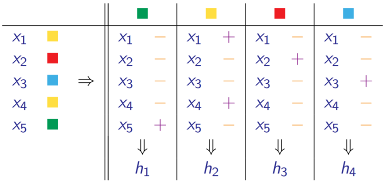


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	

training time: how many training points are created

prediction time: how many binary predictions are made

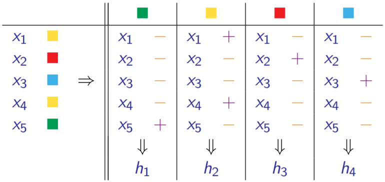


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust

training time: how many training points are created

prediction time: how many binary predictions are made

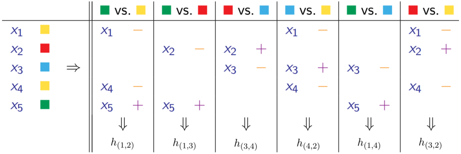


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO			

training time: how many training points are created

prediction time: how many binary predictions are made

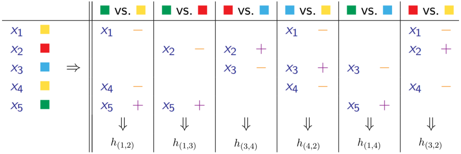


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(C-1)N$		

training time: how many training points are created

prediction time: how many binary predictions are made

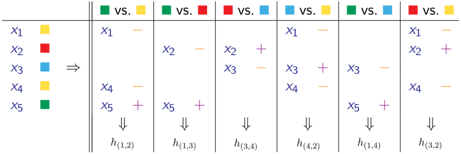


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(C-1)N$	$\mathcal{O}(C^2)$	

training time: how many training points are created

prediction time: how many binary predictions are made

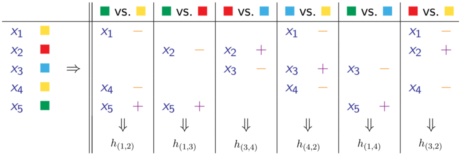


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(C-1)N$	$\mathcal{O}(C^2)$	can achieve very small training error

training time: how many training points are created

prediction time: how many binary predictions are made

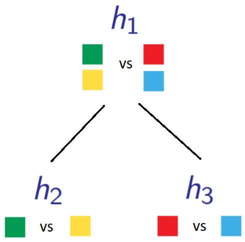


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(C-1)N$	$\mathcal{O}(C^2)$	can achieve very small training error
Tree			

training time: how many training points are created

prediction time: how many binary predictions are made

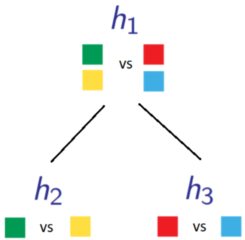


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(\mathsf{C}-1)\mathsf{N}$	$\mathcal{O}(\mathsf{C}^2)$	can achieve very small training error
Tree	$\mathcal{O}((\log_2 \mathsf{C})\mathsf{N})$		

training time: how many training points are created

prediction time: how many binary predictions are made

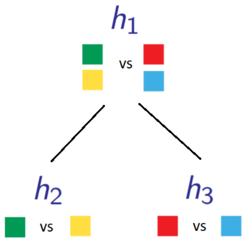


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(\mathsf{C}-1)\mathsf{N}$	$\mathcal{O}(\mathsf{C}^2)$	can achieve very small training error
Tree	$\mathcal{O}((\log_2 \mathsf{C})\mathsf{N})$	$\mathcal{O}(\log_2 \mathsf{C})$	

training time: how many training points are created

prediction time: how many binary predictions are made

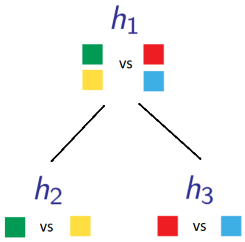


Comparisons

Reduction	**training time**	**prediction time**	**remark**
OvA	CN	C	not robust
OvO	$(\mathsf{C}-1)\mathsf{N}$	$\mathcal{O}(\mathsf{C}^2)$	can achieve very small training error
Tree	$\mathcal{O}((\log_2 \mathsf{C})\mathsf{N})$	$\mathcal{O}(\log_2 \mathsf{C})$	good for "extreme classification"

training time: how many training points are created

prediction time: how many binary predictions are made



Outline

1. Review of Last Lecture
2. Multiclass Classification
3. Kernel methods
   - Motivation
   - Example: Perceptron
   - Kernel Trick
   - Kernelized Perceptron

Motivation

Recall: when linear models are not good enough, we can use **a nonlinear feature map** $\phi : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ to transform all $\boldsymbol{x}$ to $\phi(\boldsymbol{x})$.

Motivation

Recall: when linear models are not good enough, we can use **a nonlinear feature map** $\phi : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ to transform all $\boldsymbol{x}$ to $\boldsymbol{\phi}(\boldsymbol{x})$.

Issue: what if M is huge, or even infinity?

Motivation

Recall: when linear models are not good enough, we can use **a nonlinear feature map** $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ to transform all $\boldsymbol{x}$ to $\boldsymbol{\phi}(\boldsymbol{x})$.

Issue: what if M is huge, or even infinity?

Solution: **kernel methods**

Case study: Perceptron for binary classification

Perceptron

Initialize $\boldsymbol{w} = \boldsymbol{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

Case study: Perceptron for binary classification

Perceptron

Initialize $\boldsymbol{w} = \mathbf{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n \boldsymbol{x}_n$

Observation: $\boldsymbol{w}$ is a **linear combination** of training data

$$\boldsymbol{w} = \sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_m$$

Case study: Perceptron for binary classification

Perceptron

Initialize $\boldsymbol{w} = \boldsymbol{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

Observation: $\boldsymbol{w}$ is a **linear combination** of training data

$$\boldsymbol{w} = \sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_m$$

where $\alpha_m = y_m \times$ number of times x_m has been misclassified

Dual form of Perceptron

Perceptron (primal form)

Initialize $\boldsymbol{w} = \mathbf{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

Dual form of Perceptron

Perceptron (primal form)

Initialize $\boldsymbol{w} = \mathbf{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

How to update $\alpha_1, \ldots, \alpha_{\mathsf{N}}$ so that $\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_m \equiv \boldsymbol{w}$ holds always?

Perceptron (dual form)

Dual form of Perceptron

Perceptron (primal form)

Initialize $\boldsymbol{w} = \mathbf{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

How to update $\alpha_1, \ldots, \alpha_{\mathsf{N}}$ so that $\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_m \equiv \boldsymbol{w}$ holds always?

Perceptron (dual form)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$

Dual form of Perceptron

Perceptron (primal form)

Initialize $\boldsymbol{w} = \mathbf{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

How to update $\alpha_1, \ldots, \alpha_{\mathsf{N}}$ so that $\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_m \equiv \boldsymbol{w}$ holds always?

Perceptron (dual form)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$

Repeat:

- Pick a data point index n uniformly at random

Dual form of Perceptron

Perceptron (primal form)

Initialize $\boldsymbol{w} = \boldsymbol{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

How to update $\alpha_1, \ldots, \alpha_{\mathsf{N}}$ so that $\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_m \equiv \boldsymbol{w}$ holds always?

Perceptron (dual form)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$

Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_{\boldsymbol{m}}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$,

Dual form of Perceptron

Perceptron (primal form)

Initialize $\boldsymbol{w} = \boldsymbol{0}$

Repeat:

- Pick a data point index n uniformly at random
- If $\mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$

How to update $\alpha_1, \ldots, \alpha_{\mathsf{N}}$ so that $\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_m \equiv \boldsymbol{w}$ holds always?

Perceptron (dual form)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$

Repeat:

- Pick a data point index n uniformly at random
- If $\mathrm{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{x}_{\boldsymbol{m}}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Applying a feature map

Perceptron (primal form with ϕ)

Initialize $\boldsymbol{w} = \mathbf{0} \in \mathbb{R}^{\mathsf{M}}$
Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$

Applying a feature map

Perceptron (primal form with ϕ) *issue: time/space linear in* M

Initialize $\boldsymbol{w} = \mathbf{0} \in \mathbb{R}^{\mathsf{M}}$
Repeat:

- Pick a data point index n uniformly at random
- If sgn$(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$

Applying a feature map

Perceptron (primal form with $\boldsymbol{\phi}$) *issue: time/space linear in* M

Initialize $\boldsymbol{w} = \mathbf{0} \in \mathbb{R}^{\mathsf{M}}$

Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$

Perceptron (dual form with $\boldsymbol{\phi}$)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$

Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Applying a feature map

Perceptron (primal form with $\boldsymbol{\phi}$) *issue: time/space linear in* M

Initialize $\boldsymbol{w} = \mathbf{0} \in \mathbb{R}^{\mathsf{M}}$
Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{\phi}(\boldsymbol{x}_n)$

Perceptron (dual form with $\boldsymbol{\phi}$)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

If we can compute $\boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_n)$ without explicitly evaluating $\boldsymbol{\phi}(\boldsymbol{x}_m)$ and $\boldsymbol{\phi}(\boldsymbol{x}_n)$, then time/space is independent of M!

Example

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

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

Example

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 $\boldsymbol{\phi}(\boldsymbol{x}')$?

$$\boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}') = {x_1}^2{x_1'}^2 + 2x_1x_2x_1'x_2' + {x_2}^2{x_2'}^2$$

Example

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 $\boldsymbol{\phi}(\boldsymbol{x}')$?

$$\begin{aligned} \boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}') &= {x_1}^2{x_1'}^2 + 2x_1x_2x_1'x_2' + {x_2}^2{x_2'}^2 \\ &= (x_1x_1' + x_2x_2')^2 \end{aligned}$$

Example

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 $\boldsymbol{\phi}(\boldsymbol{x}')$?

$$\begin{aligned} \boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}') &= {x_1}^2{x_1'}^2 + 2x_1x_2x_1'x_2' + {x_2}^2{x_2'}^2 \\ &= (x_1x_1' + x_2x_2')^2 = (\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}')^2 \end{aligned}$$

Example

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

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

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

$$\begin{aligned} \boldsymbol{\phi}(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}') &= {x_1}^2 {x_1'}^2 + 2 x_1 x_2 x_1' x_2' + {x_2}^2 {x_2'}^2 \\ &= (x_1 x_1' + x_2 x_2')^2 = (\boldsymbol{x}^{\mathrm{T}} \boldsymbol{x}')^2 \end{aligned}$$

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

Another example

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

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

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 $\boldsymbol{\phi}_\theta(\boldsymbol{x}')$?

$$\boldsymbol{\phi}_\theta(\boldsymbol{x})^{\mathrm{T}} \boldsymbol{\phi}_\theta(\boldsymbol{x}') = \sum_{d=1}^{\mathsf{D}} \cos(\theta x_d) \cos(\theta x'_d) + \sin(\theta x_d) \sin(\theta x'_d)$$

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 $\boldsymbol{\phi}_\theta(\boldsymbol{x}')$?

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

Another example

$\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{2\mathsf{D}}$ 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 $\boldsymbol{\phi}_\theta(\boldsymbol{x}')$?

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

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 ϕ_θ, define $\phi_L : \mathbb{R}^{\mathsf{D}} \to \mathbb{R}^{2\mathsf{D}(L+1)}$ for some integer L:

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

More complicated example

Based on ϕ_θ, define $\phi_L : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{2\mathsf{D}(L+1)}$ for some integer L:

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

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

When $L \to \infty$, even if we cannot compute $\phi(x)$, a vector of *infinite dimension*, we can still compute inner product:

Infinite dimensional mapping

When $L \to \infty$, even if we cannot compute $\phi(x)$, a vector of *infinite dimension*, we can still compute 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.

Infinite dimensional mapping

When $L \to \infty$, even if we cannot compute $\phi(x)$, a vector of *infinite dimension*, we can still compute 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 classification, we are *learning a weight $\boldsymbol{w}$ with infinite dimension!*

Kernel functions

Definition: a function $k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}$ is called a *kernel function* if there exists a function $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ 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

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

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

Can be seen as a kind of similarity measure.

Kernel functions

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

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

Can be seen as a kind of similarity measure.

Examples we have seen

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

$$k(\boldsymbol{x}, \boldsymbol{x}') = \sum_{d=1}^{\mathsf{D}} \frac{\sin(2\pi(x_d - x'_d))}{x_d - x'_d}$$

Composing kernels

Creating more kernel functions using the following rules:

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

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

- **conical combination**: $\alpha k_1(\cdot, \cdot) + \beta k_2(\cdot, \cdot)$ if $\alpha, \beta \geq 0$

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

- **conical combination**: $\alpha k_1(\cdot,\cdot) + \beta k_2(\cdot,\cdot)$ if $\alpha, \beta \geq 0$
- **product**: $k_1(\cdot,\cdot)k_2(\cdot,\cdot)$

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

- **conical 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**: $e^{k(\cdot,\cdot)}$

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

- **conical 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**: $e^{k(\cdot, \cdot)}$
- $\cdots$

Verify using the definition of kernel!

Common 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.

Common 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 $\sigma > 0$.

Common 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.

Back to Perceptron

Perceptron (dual form with $\boldsymbol{\phi}$)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Back to Perceptron

Perceptron (dual form with $\boldsymbol{\phi}$)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Instead of choosing $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ explicitly, we choose a kernel function k.

Back to Perceptron

Perceptron (dual form with $\boldsymbol{\phi}$)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Instead of choosing $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ explicitly, we choose a kernel function k.

Kernelized Perceptron

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\operatorname{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m k(\boldsymbol{x}_m, \boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Back to Perceptron

Perceptron (dual form with ϕ)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Instead of choosing $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ explicitly, we choose a kernel function k.

Kernelized Perceptron

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$
Repeat:

- Pick a data point index n uniformly at random
- If $\text{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m k(\boldsymbol{x}_m, \boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Completely M-*independent*,

Back to Perceptron

Perceptron (dual form with $\boldsymbol{\phi}$)

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$

Repeat:

- Pick a data point index n uniformly at random
- If $\mathrm{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m \boldsymbol{\phi}(\boldsymbol{x}_m)^{\mathrm{T}} \boldsymbol{\phi}(\boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Instead of choosing $\boldsymbol{\phi} : \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}^{\mathsf{M}}$ explicitly, we choose a kernel function k.

Kernelized Perceptron

Initialize $\alpha_m = 0$ for all $m \in [\mathsf{N}]$

Repeat:

- Pick a data point index n uniformly at random
- If $\mathrm{sgn}(\sum_{m=1}^{\mathsf{N}} \alpha_m k(\boldsymbol{x}_m, \boldsymbol{x}_n)) \neq y_n$, update $\alpha_n \leftarrow \alpha_n + y_n$

Completely M-*independent*, becomes a *non-parametric* method

Gram/kernel matrix

When N is small, can precompute all inner products as a **Gram matrix**

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

Gram/kernel matrix

When N is small, can precompute all inner products as a **Gram matrix**

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

Recall: $\boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}} \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{N \times M}$

Gram/kernel matrix

When N is small, can precompute all inner products as a **Gram matrix**

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

Recall: $\boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\phi}(\boldsymbol{x}_1)^{\mathrm{T}} \\ \boldsymbol{\phi}(\boldsymbol{x}_2)^{\mathrm{T}} \\ \vdots \\ \boldsymbol{\phi}(\boldsymbol{x}_N)^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{N \times M}$

Gram matrix vs covariance matrix

	dimensions	entry (i, j)	property
$\mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}$			
$\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}$			

Gram matrix vs covariance matrix

	dimensions	**entry (i, j)**	**property**
$\boldsymbol{\Phi\Phi}^{\mathrm{T}}$	$\mathsf{N} \times \mathsf{N}$		
$\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$			

Gram matrix vs covariance matrix

	dimensions	**entry (i,j)**	**property**
$\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$	N $\times$ N		
$\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$	M $\times$ M		

Gram matrix vs covariance matrix

	dimensions	entry (i, j)	property
$\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$	N $\times$ N	$\boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)$	
$\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$	M $\times$ M		

Gram matrix vs covariance matrix

	dimensions	**entry** (i,j)	**property**
$\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$	N $\times$ N	$\boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)$	
$\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$	M $\times$ M	$\sum_{n=1}^{N}\phi(\boldsymbol{x}_n)_i\phi(\boldsymbol{x}_n)_j$	

Gram matrix vs covariance matrix

	dimensions	entry (i, j)	property
$\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$	N × N	$\boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)$	both are **symmetric** and **positive semidefinite**
$\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$	M × M	$\sum_{n=1}^{N} \phi(\boldsymbol{x}_n)_i \phi(\boldsymbol{x}_n)_j$	

Mercer Theorem

$k : \mathbb{R}^\mathsf{D} \times \mathbb{R}^\mathsf{D} \rightarrow \mathbb{R}$ is a kernel function if and only if the Gram matrix $\boldsymbol{K}$ for *any N and any $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, . . ., $\boldsymbol{x}_N$* is positive semidefinite.

Mercer Theorem

$k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}$ is a kernel function if and only if the Gram matrix $\boldsymbol{K}$ for *any N and any $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., $\boldsymbol{x}_N$* is positive semidefinite.

- useful for showing that a function is not a kernel

Mercer Theorem

$k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}$ is a kernel function if and only if the Gram matrix $\boldsymbol{K}$ for *any N and any $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, ..., $\boldsymbol{x}_N$* is positive semidefinite.

- useful for showing that a function is not a kernel

Example:

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

is *not a kernel*, why?

Mercer Theorem

$k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \to \mathbb{R}$ is a kernel function if and only if the Gram matrix $\boldsymbol{K}$ for *any* N *and any* $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$ is positive semidefinite.

- useful for showing that a function is not a kernel

Example:

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

is *not a kernel*, why?

If it is a kernel, the kernel matrix for two data points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$:

$$\boldsymbol{K} = \begin{pmatrix} 0 & \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 \\ \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 & 0 \end{pmatrix}$$

must be positive semidefinite,

Mercer Theorem

$k : \mathbb{R}^{\mathsf{D}} \times \mathbb{R}^{\mathsf{D}} \rightarrow \mathbb{R}$ is a kernel function if and only if the Gram matrix $\boldsymbol{K}$ for *any N and any $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$* is positive semidefinite.

- useful for showing that a function is not a kernel

Example:

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

is *not a kernel*, why?

If it is a kernel, the kernel matrix for two data points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$:

$$\boldsymbol{K} = \begin{pmatrix} 0 & \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 \\ \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2^2 & 0 \end{pmatrix}$$

must be positive semidefinite, but it is not (contradiction).

Kernelizing ML algorithms

Many other ML algorithms can be **kernelized**:

- nearest neighbor classifier
- linear regression
- logistic regression
- SVM
- $\cdots$

Kernelizing ML algorithms

Many other ML algorithms can be **kernelized**:

- nearest neighbor classifier
- linear regression
- logistic regression
- SVM
- $\cdots$

Key idea: rewrite the algorithm so that its dependence on the transformed dataset $\mathbf{\Phi}$ is only through the Gram matrix $\boldsymbol{K} = \mathbf{\Phi}\mathbf{\Phi}^{\mathrm{T}}$.