Chapter 6
Linear Models

supplementary slides to
Machine Learning Fundamentals

published by Cambridge University Press

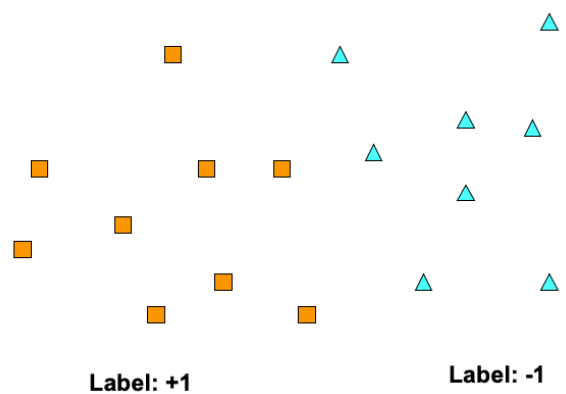
August 2020

CAMBRIDGE
UNIVERSITY PRESS

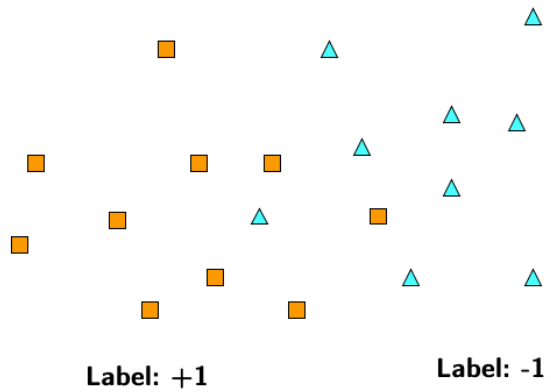
Outline

Linear Models

- linear models: $\mathbf{x} \in \mathbb{R}^d \longmapsto y \in \mathbb{R}$
  - linear functions: $y = \mathbf{w}^\intercal \mathbf{x} \quad (\mathbf{w} \in \mathbb{R}^d)$
  - affine functions: $y = \mathbf{w}^\intercal \mathbf{x} + b \quad (\mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R})$
- binary classification problems: $\mathcal{D} = \left\{ (\mathbf{x}_i, y_i) \mid i = 1, 2, \cdots N \right\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ and binary label $y_i \in \{+1, -1\}$
- distinguish linearly *separable* vs. *non-separable* cases



(a) linearly separable



(b) linearly nonseparable

Perceptron

Use a linear model for 2-class problems (Rosenblatt, 1957):

$$y = \text{sign}(\mathbf{w}^{\mathsf{T}}\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w}^{\mathsf{T}}\mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

Perceptron Algorithm

initialize $\mathbf{w}^{(0)} = 0$, $n = 0$

loop

  randomly choose a sample $(\mathbf{x}_i, y_i)$ in $\mathcal{D}$

  calculate the actual output $h_i = \text{sign}(\mathbf{w}^{(n)\mathsf{T}}\mathbf{x}_i)$

  if upon a mistake: $h_i \neq y_i$ **then**

    $\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + y_i\mathbf{x}_i$

    $n = n + 1$

  else if no mistake is found **then**

    return $\mathbf{w}^{(n)}$ and terminate

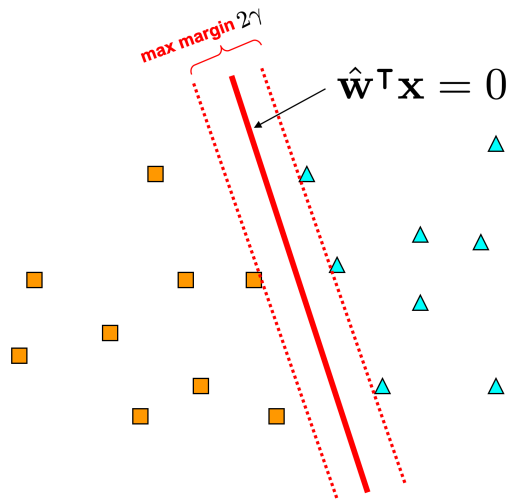
  end if

end loop

Perceptron: Convergence

- assume training set $\mathcal{D}$ is *linearly separable*
- normalizing all input vectors in $\mathcal{D}$:
  $\|\mathbf{x}_i\| \leq 1 \quad \forall i = \{1, 2, \cdots, N\}$
- separation margin (scaling $\|\hat{\mathbf{w}}\| = 1$):

$$\gamma = \min_{\mathbf{x}_i \in \mathcal{D}} \frac{|\hat{\mathbf{w}}^{\mathsf{T}} \mathbf{x}_i|}{\|\hat{\mathbf{w}}\|} = \min_{\mathbf{x}_i \in \mathcal{D}} |\hat{\mathbf{w}}^{\mathsf{T}} \mathbf{x}_i|$$



Theorem 1 (Convergence of Perceptron Algorithm)

If the perceptron algorithm is run on a linearly separable training set $\mathcal{D}$, the number of mistakes made is at most $1/\gamma^2$. In other words, the perceptron algorithm will terminate after at most $\lceil 1/\gamma^2 \rceil$ updates and return a hyperplane that perfectly separates $\mathcal{D}$.

Perceptron: Proof Convergence

proof sketch:

1. margin definition: $|\hat{\mathbf{w}}^{\mathsf{T}}\mathbf{x}_i| \geq \gamma \implies y_i\hat{\mathbf{w}}^{\mathsf{T}}\mathbf{x}_i \geq \gamma \quad \left(\forall(\mathbf{x}_i, y_i) \in \mathcal{D}\right)$
2. record all M mistakes: $\mathcal{M} = \{(\mathbf{x}^{(1)}, y^{(1)}), \cdots, (\mathbf{x}^{(M)}, y^{(M)})\}$, then $\sum_{n\in\mathcal{M}} y^{(n)}\hat{\mathbf{w}}^{\mathsf{T}}\mathbf{x}^{(n)} \geq M \cdot \gamma$
3. use Cauchy–Schwarz ineq. and $\|\hat{\mathbf{w}}\| = 1$: $\sum_{n\in\mathcal{M}} y^{(n)}\hat{\mathbf{w}}^{\mathsf{T}}\mathbf{x}^{(n)} \leq \|\hat{\mathbf{w}}\| \cdot \left\|\sum_{n\in\mathcal{M}} y^{(n)}\mathbf{x}^{(n)}\right\| = \left\|\sum_{n\in\mathcal{M}} y^{(n)}\mathbf{x}^{(n)}\right\|$
4. $\left\|\sum_{n\in\mathcal{M}} y^{(n)}\mathbf{x}^{(n)}\right\| = \left\|\sum_{n\in\mathcal{M}} \left(\mathbf{w}^{(n+1)} - \mathbf{w}^{(n)}\right)\right\| = \left\|\mathbf{w}^{(M+1)}\right\| = \sqrt{\|\mathbf{w}^{(M+1)}\|^2} = \sqrt{\sum_{n\in\mathcal{M}} \left(\|\mathbf{w}^{(n+1)}\|^2 - \|\mathbf{w}^{(n)}\|^2\right)}$
5. $\|\mathbf{w}^{(n+1)}\|^2 - \|\mathbf{w}^{(n)}\|^2 = \|\mathbf{w}^{(n)} + y^{(n)}\mathbf{x}^{(n)}\|^2 - \|\mathbf{w}^{(n)}\|^2 =$ $\cancel{\|\mathbf{w}^{(n)}\|^2} + \overbrace{2y^{(n)}\mathbf{w}^{(n)\mathsf{T}}\mathbf{x}^{(n)}}^{<0} + \cancel{(y^{(n)})^2}^{(\pm 1)^2}\|\mathbf{x}^{(n)}\|^2 - \cancel{\|\mathbf{w}^{(n)}\|^2} < \|\mathbf{x}^{(n)}\|^2 \leq 1$
6. $M \cdot \gamma \leq \left\|\sum_{n\in\mathcal{M}} y^{(n)}\mathbf{x}^{(n)}\right\| < \sqrt{M} \implies M < (1/\gamma)^2$ ■

Linear Regression (I)

- how about *linearly non-separable* cases?
- find a linear mapping $y = \mathbf{w}^{\mathsf{T}}\mathbf{x}$ to fit all data in $\mathcal{D}$:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^{\mathsf{T}} \\ \mathbf{x}_2^{\mathsf{T}} \\ \vdots \\ \mathbf{x}_N^{\mathsf{T}} \end{bmatrix}_{N \times d} \longmapsto \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}_{N \times 1}$$

- the least square error:

$$E(\mathbf{w}) = \sum_{i=1}^{N} \left(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i - y_i \right)^2 = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$$

- linear regression:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \; E(\mathbf{w})$$

Linear Regression (II)

- compute the gradient:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = 2\mathbf{X}^{\mathsf{T}}\mathbf{X}\mathbf{w} - 2\mathbf{X}^{\mathsf{T}}\mathbf{y}$$

- derive the closed-form solution by vanishing the gradient:

$$\mathbf{w}^* = \left(\mathbf{X}^{\mathsf{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}$$

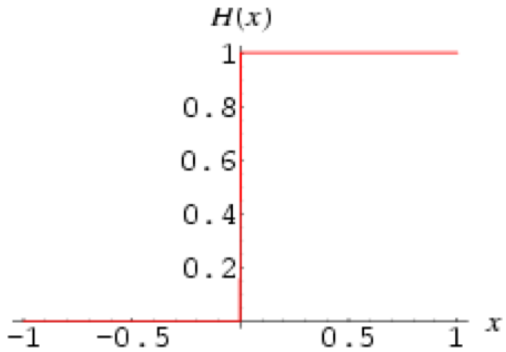
  with $\mathbf{X}^{\mathsf{T}}\mathbf{X} \in \mathbb{R}^{d\times d}$
- may alternatively use gradient descent to derive $\mathbf{w}^*$ iteratively to avoid the matrix inversion
- assign a new input $\mathbf{x}$ in the test stage:

$$y = \text{sign}(\mathbf{w}^{*\mathsf{T}}\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w}^{*\mathsf{T}}\mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

Minimum Classification Error (I)

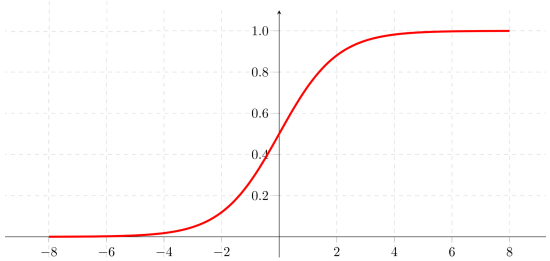
- ERM minimizes 0-1 classification errors in $\mathcal{D}$
- for each training sample $(\mathbf{x}_i, y_i)$ in $\mathcal{D}$

$$-y_i \mathbf{w}^{\mathsf{T}} \mathbf{x}_i = \begin{cases} > 0 & \Longrightarrow \text{ mis-classification} \\ < 0 & \Longrightarrow \text{ correct classification} \end{cases}$$



- count the 0-1 training errors:

$$E_0(\mathbf{w}) = \sum_{i=1}^{N} H(-y_i \mathbf{w}^{\mathsf{T}} \mathbf{x}_i)$$



- smooth approximation by *sigmoid* $l(x)$:

$$E_1(\mathbf{w}) = \sum_{i=1}^{N} l(-y_i \mathbf{w}^{\mathsf{T}} \mathbf{x}_i)$$

$$l(x) = \frac{1}{1 + e^{-x}}$$

Minimum Classification Error (II)

- minimum classification error (MCE): learn $\mathbf{w}$ by minimizing the sum of smooth errors $E_1(\mathbf{w})$

$$\mathbf{w}_{\text{MCE}} = \arg\min_{\mathbf{w}} \; E_1(\mathbf{w}) = \arg\min_{\mathbf{w}} \; \sum_{i=1}^{N} l(-y_i \mathbf{w}^{\mathsf{T}} \mathbf{x}_i)$$

- MCE does not have a closed-form solution
- rely on a gradient descent or SGD method:

$$\frac{\partial E_1(\mathbf{w})}{\partial \mathbf{w}} = \sum_{i=1}^{N} y_i \, l(y_i \mathbf{w}^{\mathsf{T}} \mathbf{x}_i) \Big(1 - l(y_i \mathbf{w}^{\mathsf{T}} \mathbf{x}_i)\Big) \mathbf{x}_i$$

where $\frac{dl(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = l(x)\big(1 - l(x)\big)$

Logistic Regression (I)

- for any $(\mathbf{x}_i, y_i)$ in $\mathcal{D}$, the quantity $l(-y_i\mathbf{w}^\intercal\mathbf{x}_i) \in (0,1)$:
  - MCE interprets it as a soft error to misclassify $(\mathbf{x}_i, y_i)$
  - it can be viewed as a probability to misclassify $(\mathbf{x}_i, y_i)$
  - the probability to correctly classify $(\mathbf{x}_i, y_i)$: $1 - l(-y_i\mathbf{w}^\intercal\mathbf{x}_i) = l(y_i\mathbf{w}^\intercal\mathbf{x}_i)$
- logistic regression learns $\mathbf{w}$ by maximizing the probability to correctly classify all samples in $\mathcal{D}$

$$\begin{aligned}\mathbf{w}_{\text{LR}} &= \arg\max_{\mathbf{w}} L(\mathbf{w}) = \arg\max_{\mathbf{w}} \prod_{i=1}^{N} l(y_i\mathbf{w}^\intercal\mathbf{x}_i) \\ &= \arg\max_{\mathbf{w}} \ln L(\mathbf{w}) = \arg\max_{\mathbf{w}} \sum_{i=1}^{N} \ln l(y_i\mathbf{w}^\intercal\mathbf{x}_i)\end{aligned}$$

Logistic Regression (II)

- rely on gradient descent or SGD:

$$\frac{\partial \ln L(\mathbf{w})}{\partial \mathbf{w}} = \sum_{i=1}^{N} y_i \Big(1 - l(y_i \mathbf{w}^{\mathsf{T}} \mathbf{x}_i)\Big) \mathbf{x}_i$$

- logistic regression vs. MCE
  - MCE learning focuses more on the boundary cases
  - logistic regression generates significant gradients for all mis-classified samples: prone to outliers but faster convergence
- can be extended to multi-class using the *softmax* function

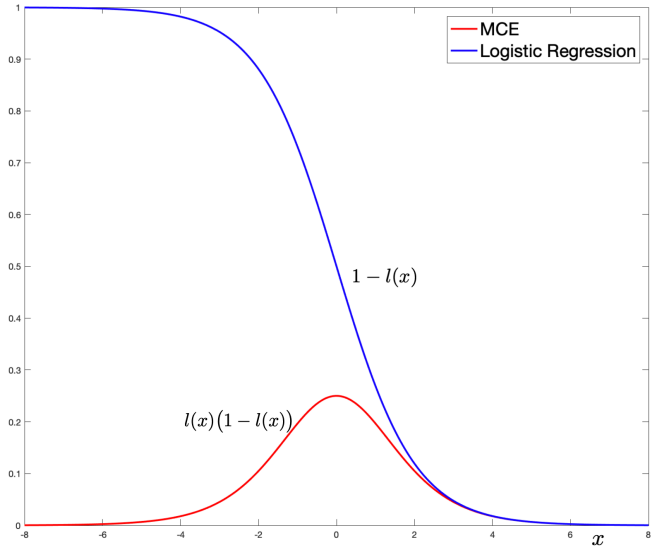


Figure: Comparison of the gradient weights of MCE and logistic regression

Support Vector Machines (SVMs)

- linear SVMs
- soft SVMs
- nonlinear SVMs
- solving quadratic programming
- multi-class SVMs

Linear SVMs (1): SVM0

- assume the data $\mathcal{D}$ is linearly separable
- how to find the max-margin hyperplane?
- margin: $\gamma = \min_{\mathbf{x}_i \in \mathcal{D}_N} \frac{y_i(\mathbf{w}^T\mathbf{x}_i + b)}{||\mathbf{w}||}$
- linear SVM $\Longrightarrow$ max margin hyperplane:

$$\{\mathbf{w}^*, b^*\} = \arg\max_{\mathbf{w},b} \ \gamma$$

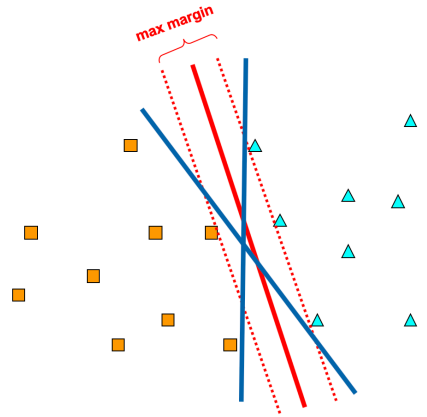


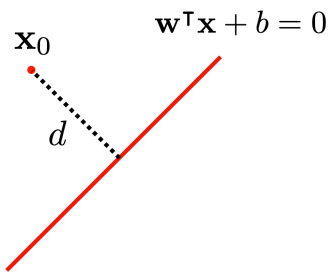
Figure: max margin hyperplane

SVM0

$$\max_{\gamma,\mathbf{w},b} \quad \gamma$$

subject to

$$\frac{y_i(\mathbf{w}^T\mathbf{x}_i + b)}{||\mathbf{w}||} \geq \gamma \quad \forall i \in \{1, 2, \cdots, N\}$$



$$d = \frac{|\mathbf{w}^T\mathbf{x}_0 + b|}{\|\mathbf{w}\|}$$

Linear SVMs (2): SVM1

- scale $\mathbf{w}, b$ to normalize numerators $\Longrightarrow$ margin $\gamma = \frac{1}{\|\mathbf{w}\|}$
- max margin $\max \gamma \iff \min \|\mathbf{w}\|^2$
- an equivalent formulation of linear SVMs:

SVM1

$$\min_{\mathbf{w},b} \quad \frac{1}{2}\mathbf{w}^{\mathsf{T}}\mathbf{w}$$

subject to

$$y_i(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i + b) \geq 1 \quad \forall i \in \{1, 2, \cdots, N\}$$

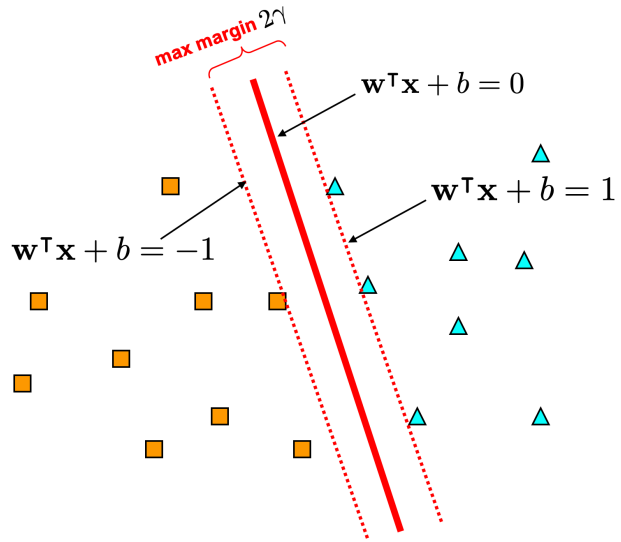


Figure: scaling $\mathbf{w}, b$ does not change the location of the hyperplane $\mathbf{w}^{\mathsf{T}}\mathbf{x} + b = 0$

Linear SVMs (3): Lagrange Duality

- use the KKT conditions to derive the Lagrange duality
  1. introduce Lagrange multipliers $\alpha_i \geq 0$ to derive the Lagrangian

$$L\big(\mathbf{w}, b, \{\alpha_i\}\big) = \frac{1}{2}\mathbf{w}^{\mathsf{T}}\mathbf{w} + \sum_{i=1}^{N} \alpha_i \Big(1 - y_i(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i + b)\Big)$$

  2. the Lagrange dual function:

$$\begin{aligned} L^*(\{\alpha_i\}) &= \inf_{\mathbf{w},b} \; L\big(\mathbf{w}, b, \{\alpha_i\}\big) \\ &= \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j y_i y_j \mathbf{x}_i^{\mathsf{T}}\mathbf{x}_j \end{aligned}$$

  3. subject to the constraint: $\sum_{i=1}^{N} \alpha_i y_i = 0$

Linear SVMs (4): Dual Problem

- linear SVMs are convex optimizatoin $\Longrightarrow$ strong duality
- the unique solution, i.e. $\left(\mathbf{w}^*, b^*, \{\alpha_i^*\}\right)$, is a saddle point

$$\underbrace{\text{SVM1}}_{\text{prime problem}} \iff \underbrace{\max_{\{\alpha_i\}} L^*\left(\alpha_i\right) \text{ s.t. } \sum_{i=1}^{N} \alpha_i y_i = 0}_{\text{dual problem}}$$

- solving the dual problem yields $\{\alpha_i^*\}$
- linear SVMs: $\mathbf{w}^* = \sum_{i=1}^{N} \alpha_i^* y_i \mathbf{x}_i$ and $b^* = y_i - \mathbf{w}^{*\mathsf{T}} \mathbf{x}_i$

Linear SVMs (5): SVM2

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix}_{N\times 1} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}_{N\times 1} \quad \mathbf{1} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{N\times 1}$$

$$\mathbf{Q} = \begin{bmatrix} Q_{ij} \end{bmatrix}_{N\times N} = \begin{bmatrix} \mathbf{y}\mathbf{y}^{\mathsf{T}} \end{bmatrix}_{N\times N} \odot \begin{bmatrix} \mathbf{x}_1^{\mathsf{T}}\mathbf{x}_1 & \cdots & \mathbf{x}_1^{\mathsf{T}}\mathbf{x}_N \\ \vdots & \mathbf{x}_i^{\mathsf{T}}\mathbf{x}_j & \vdots \\ \mathbf{x}_N^{\mathsf{T}}\mathbf{x}_1 & \cdots & \mathbf{x}_N^{\mathsf{T}}\mathbf{x}_N \end{bmatrix}_{N\times N}$$

SVM2: Quadratic Programming

$$\max_{\boldsymbol{\alpha}} \quad \mathbf{1}^{\mathsf{T}}\boldsymbol{\alpha} - \frac{1}{2}\boldsymbol{\alpha}^{\mathsf{T}}\mathbf{Q}\boldsymbol{\alpha}$$

subject to

$$\mathbf{y}^{\mathsf{T}}\boldsymbol{\alpha} = 0$$

$$\boldsymbol{\alpha} \geq 0$$

Soft SVMs

- how about linearly non-separable cases?
- allow non-negative error terms ξ
- soft SVMs: aim to minimize the *soft margin*

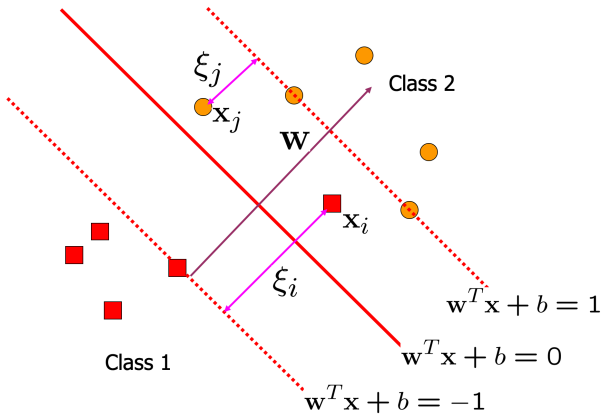
SVM3

$$\min_{\mathbf{w},b,\xi_i} \quad \frac{1}{2}\mathbf{w}^{\mathsf{T}}\mathbf{w} + C\sum_{i=1}^{N}\xi_i$$

subject to

$$y_i(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i \in \{1, 2, \cdots, N\}$$

$$\xi_i \geq 0 \quad \forall i \in \{1, 2, \cdots, N\}$$



soft margin =
margin + linear errors

Soft SVMs: Dual Problem

- use the same technique of Lagrangian
- derive the Lagrange dual function

SVM4

$$\max_{\boldsymbol{\alpha}} \quad \mathbf{1}^{\intercal}\boldsymbol{\alpha} - \frac{1}{2}\boldsymbol{\alpha}^{\intercal}\mathbf{Q}\boldsymbol{\alpha}$$

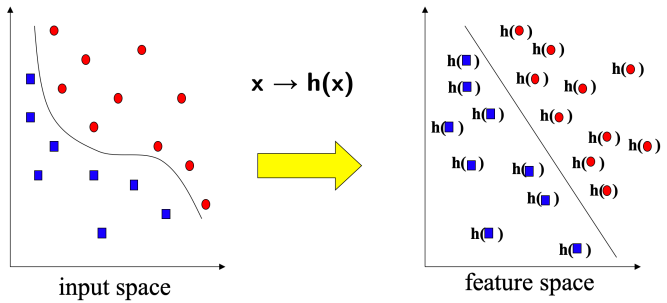
subject to

$$\mathbf{y}^{\intercal}\boldsymbol{\alpha} = 0$$

$$0 \leq \boldsymbol{\alpha} \leq C$$

Nonlinear SVMs

- apply a mapping function to a higher dimensional feature space: $\mathbf{x} \longmapsto h(\mathbf{x})$
- construct linear SVMs in the feature space
- linear SVMs only depend on $h^{\mathsf{T}}(\mathbf{x}_i)h(\mathbf{x}_j)$
- define the so-called kernel function $\Phi(\mathbf{x}_i, \mathbf{x}_j) = h^{\mathsf{T}}(\mathbf{x}_i)h(\mathbf{x}_j)$
- choose any kernel function based on the Mercer's condition
  - polynomial kernel: $\Phi(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^{\mathsf{T}}\mathbf{x}_j + 1)^p$
  - Gaussian (or RBF) kernel: $\Phi(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$



$h(\mathbf{x})$ vs. $\Phi(\mathbf{x}_i, \mathbf{x}_j)$

- $h(\mathbf{x})$: $\mathbb{R}^m \rightarrow \mathbb{R}^n$
- $\Phi(\mathbf{x}_i, \mathbf{x}_j)$: $\mathbb{R}^m + \mathbb{R}^m \rightarrow \mathbb{R}$

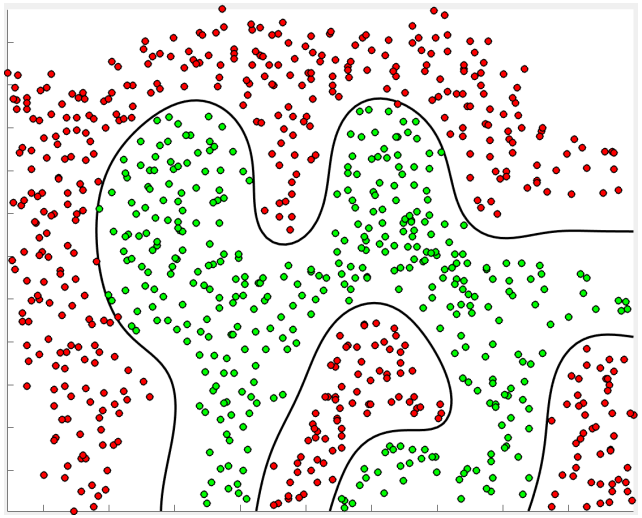
Nonlinear SVMs: Kernel Trick

- nonlinear SVMs: also solve SVM4 but using

$$\mathbf{Q} = \Big[Q_{ij}\Big]_{N\times N} = \Big[\mathbf{y}\mathbf{y}^{\mathsf{T}}\Big] \odot \begin{bmatrix} \Phi(\mathbf{x}_1,\mathbf{x}_1) & \cdots & \Phi(\mathbf{x}_1,\mathbf{x}_N) \\ \vdots & \Phi(\mathbf{x}_i,\mathbf{x}_j) & \vdots \\ \Phi(\mathbf{x}_N,\mathbf{x}_1) & \cdots & \Phi(\mathbf{x}_N,\mathbf{x}_N) \end{bmatrix}$$

- construct the nonlinear SVM:

$$y = \text{sign}\Big(\sum_{i=1}^{N} \alpha_i^* y_i \Phi(\mathbf{x}_i,\mathbf{x}) + b^*\Big)$$



- nonlinear SVMs are very powerful machine learning models
  - e.g. a proper RBF kernel is used
- *kernel tricks* in machine learning

Solving Quadratic Programming in SVMs

Dual problem of all SVMs is a dense quadratic programming:

$$\min_{\boldsymbol{\alpha}} \overbrace{\frac{1}{2}\boldsymbol{\alpha}^{\mathsf{T}}\mathbf{Q}\boldsymbol{\alpha} - \mathbf{1}^{\mathsf{T}}\boldsymbol{\alpha}}^{L(\boldsymbol{\alpha})}$$

subject to $\mathbf{y}^{\mathsf{T}}\boldsymbol{\alpha} = 0$ and $0 \leq \boldsymbol{\alpha} \leq C$, where

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_T \end{bmatrix}_{T\times 1} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_T \end{bmatrix}_{T\times 1} \quad \mathbf{1} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{T\times 1}$$

$$\mathbf{Q} = \Big[\, Q_{ij} \,\Big]_{T\times T} = \Big[\, \mathbf{y}\mathbf{y}^{\mathsf{T}} \,\Big]_{T\times T} \odot \Big[\, \Phi(\mathbf{x}_i, \mathbf{x}_j) \,\Big]_{T\times T}$$

Projected Gradient Descent for SVMs

Projected Gradient Descent Algorithm

initialize $\boldsymbol{\alpha}^{(0)} = 0$, and set $n = 0$

while not converged **do**

(1) compute the gradient: $\nabla L(\boldsymbol{\alpha}^{(n)}) = \mathbf{Q}\boldsymbol{\alpha}^{(n)} - \mathbf{1}$

(2) project the gradient to the hyperplane $\mathbf{y}^{\intercal}\boldsymbol{\alpha} = 0$:

$$\tilde{\nabla} L(\boldsymbol{\alpha}^{(n)}) = \nabla L(\boldsymbol{\alpha}^{(n)}) - \frac{\mathbf{y}^{\intercal}\nabla L(\boldsymbol{\alpha}^{(n)})}{||\mathbf{y}||^2}\mathbf{y}$$

(3) projected gradient descent: $\boldsymbol{\alpha}^{(n+1)} = \boldsymbol{\alpha}^{(n)} - \eta_n \cdot \tilde{\nabla} L(\boldsymbol{\alpha}^{(n)})$

(4) limit $\boldsymbol{\alpha}^{(n+1)}$ to $[0, C]$

(5) $n = n + 1$

end while

- projected gradient descent: intuitive but inefficient in memory use
- e.g. sequential minimization optimization (SMO) method

Multi-Class SVM

- SVMs are good for binary classification problems
- how about multi-class problems?
- build multiple binary SVMs:
  - *one-vs-one* strategy: a binary SVM to separate each pair of classes
  - *one-vs-all* strategy: a binary SVM to separate each class from all others
  - use majority-voting in the test stage
- directly formulate multi-class SVMs

SVM Summary

SVM Learning Procedure (in a nutshell)

Given a training set as $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_N, y_N)\}$

1. choose a kernel function $\Phi(\mathbf{x}_i, \mathbf{x}_j)$
2. build the matrices $\mathbf{Q}$, $\mathbf{y}$ and $\mathbf{1}$ from $\mathcal{D}$ and $\Phi(\mathbf{x}_i, \mathbf{x}_j)$
3. solve the quadratic programming problem to get:

$$\boldsymbol{\alpha}^* = [\alpha_1^*, \cdots \alpha_N^*]^\mathsf{T} \text{ and } b^*$$

4. evaluate the learned model as:

$$y = \text{sign}\left(\sum_{i=1}^{N} \alpha_i^* y_i \Phi(\mathbf{x}_i, \mathbf{x}) + b^* \right)$$