Machine Learning

CSCI 5622 Fall 2020

Prof. Claire Monteleoni

Today

- Discriminative learning II
  - Support vector machine (SVM), continued
  - Kernels
  - Kernel Perceptron

with much credit to S. Dasgupta and T. Jaakkola

Hard-margin SVM (first version we saw)

- Several desirable properties
  - maximizes the margin on the training set ($\approx$ good generalization)
  - the solution is unique and sparse ($\approx$ good generalization)

- But...
  - the solution is very sensitive to outliers, and labeling errors, as they may drastically change the resulting max-margin boundary
  - if the training set is not linearly separable, there's no solution!

Soft-margin SVM

- We relax the optimization problem by adding slack variables
- Now, not all the constraints need to be met
- The solution therefore need not:
  - Classify all training points with a margin
  - Correctly classify all training points
- The margin is still the region within $\frac{1}{\|\underline{\theta}^*\|}$ of the decision boundary

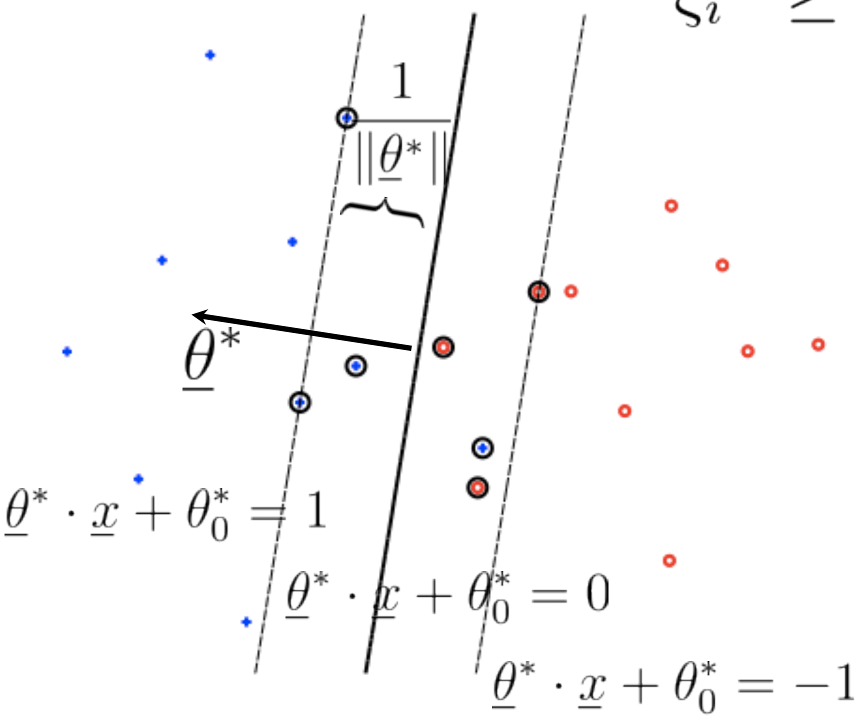
Soft-margin SVM

- Relaxed quadratic optimization problem

$$\text{minimize} \quad \frac{1}{2}\|\underline{\theta}\|^2 \quad + \quad C\sum_{i=1}^{n}\xi_i \quad \text{subject to}$$

$$y_i(\underline{\theta}\cdot\underline{x}_i+\theta_0) \quad \geq \quad 1-\xi_i, \quad i=1,\ldots,n$$

$$\xi_i \quad \geq \quad 0, \quad i=1,\ldots,n$$

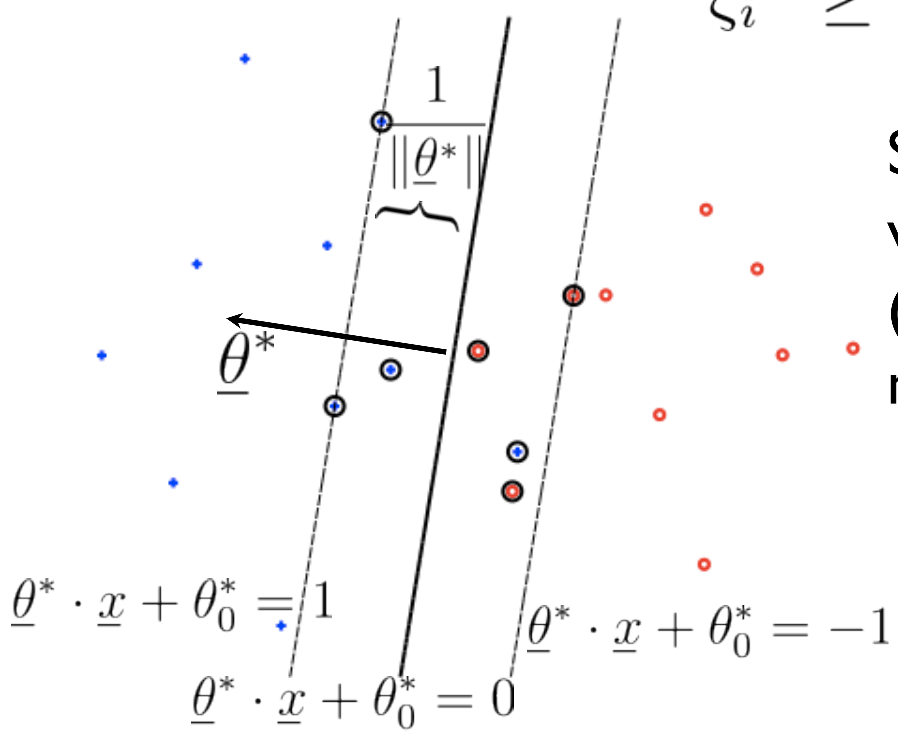


Soft-margin SVM

- Relaxed quadratic programming problem

$$\underset{\underline{\theta}, \theta_0, \xi}{\text{minimize}} \quad \frac{1}{2}\|\underline{\theta}\|^2 \quad + \quad C \sum_{i=1}^{n} \xi_i \quad \text{subject to}$$

$$\begin{aligned} y_i(\underline{\theta} \cdot \underline{x}_i + \theta_0) \quad &\geq \quad 1 - \xi_i, \quad i = 1, \ldots, n \\ \xi_i \quad &\geq \quad 0, \quad i = 1, \ldots, n \end{aligned}$$

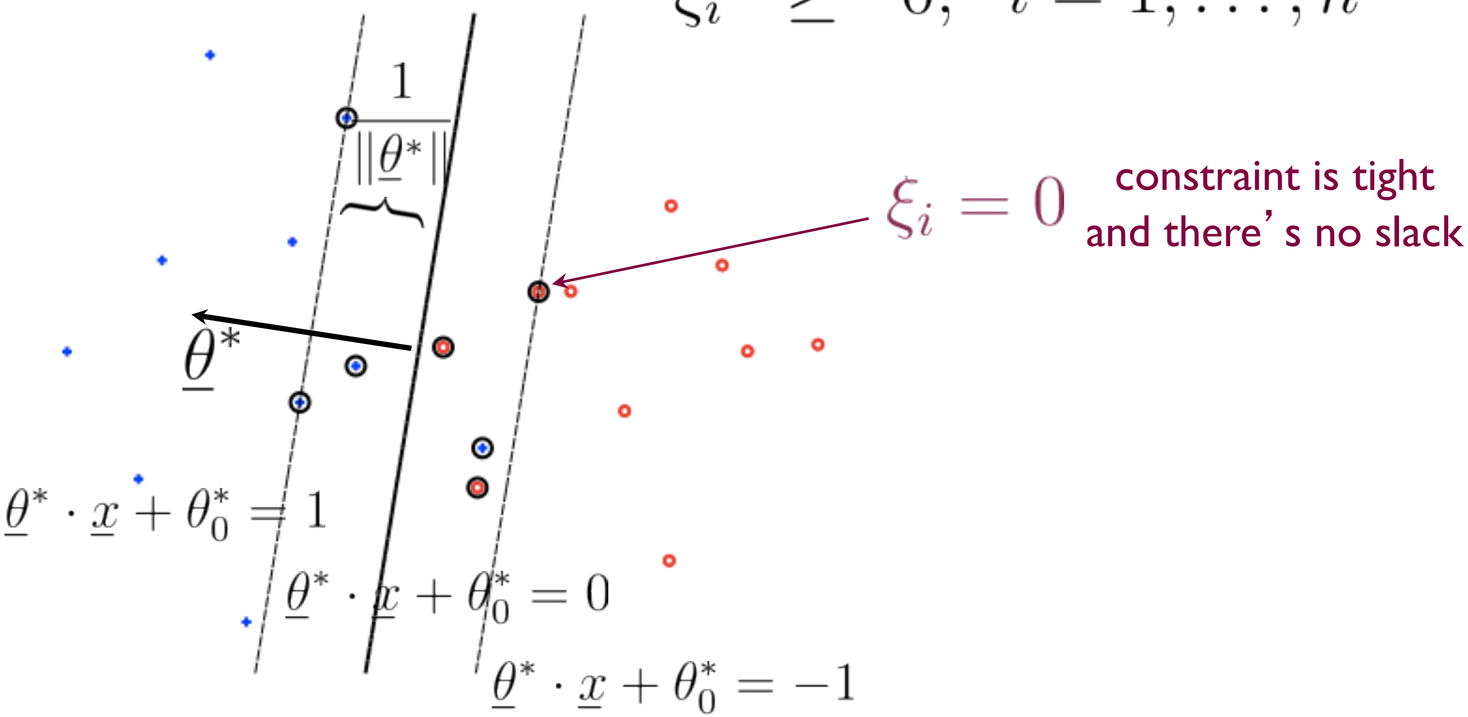


Some of the points may now violate the margin constraints (positive slack) or even be misclassified

Soft-margin SVM

- The solution now has three types of support vectors

$$
\begin{aligned}
\text{minimize} \quad \frac{1}{2}\|\underline{\theta}\|^2 \quad + \quad C\sum_{i=1}^{n}\xi_i \quad \text{subject to}\\
y_i(\underline{\theta}\cdot\underline{x}_i+\theta_0) \quad &\geq \quad 1-\xi_i, \quad i=1,\ldots,n\\
\xi_i \quad &\geq \quad 0, \quad i=1,\ldots,n
\end{aligned}
$$



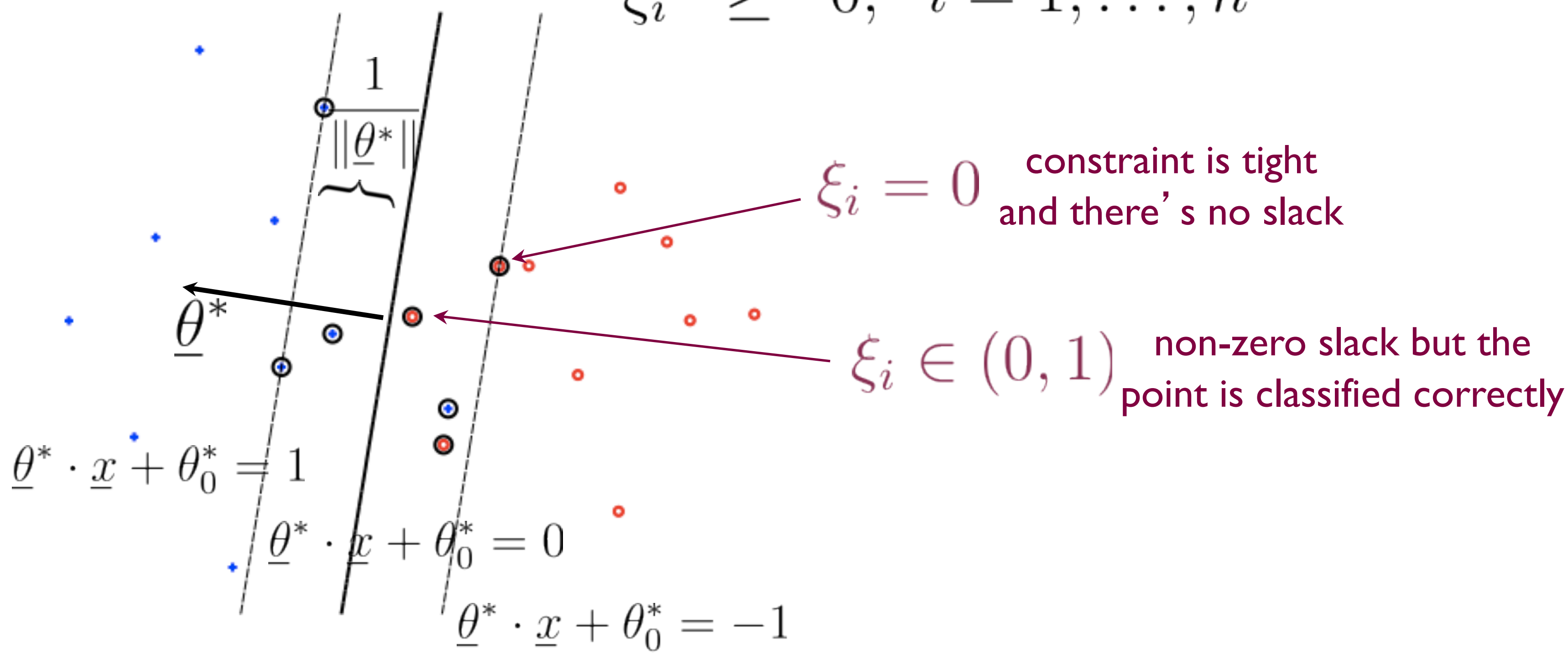
Soft-margin SVM

- The solution now has three types of support vectors

$$\text{minimize} \quad \frac{1}{2}\|\underline{\theta}\|^2 \quad + \quad C\sum_{i=1}^{n}\xi_i \quad \text{subject to}$$

$$y_i(\underline{\theta}\cdot\underline{x}_i+\theta_0) \;\geq\; 1-\xi_i, \quad i=1,\ldots,n$$

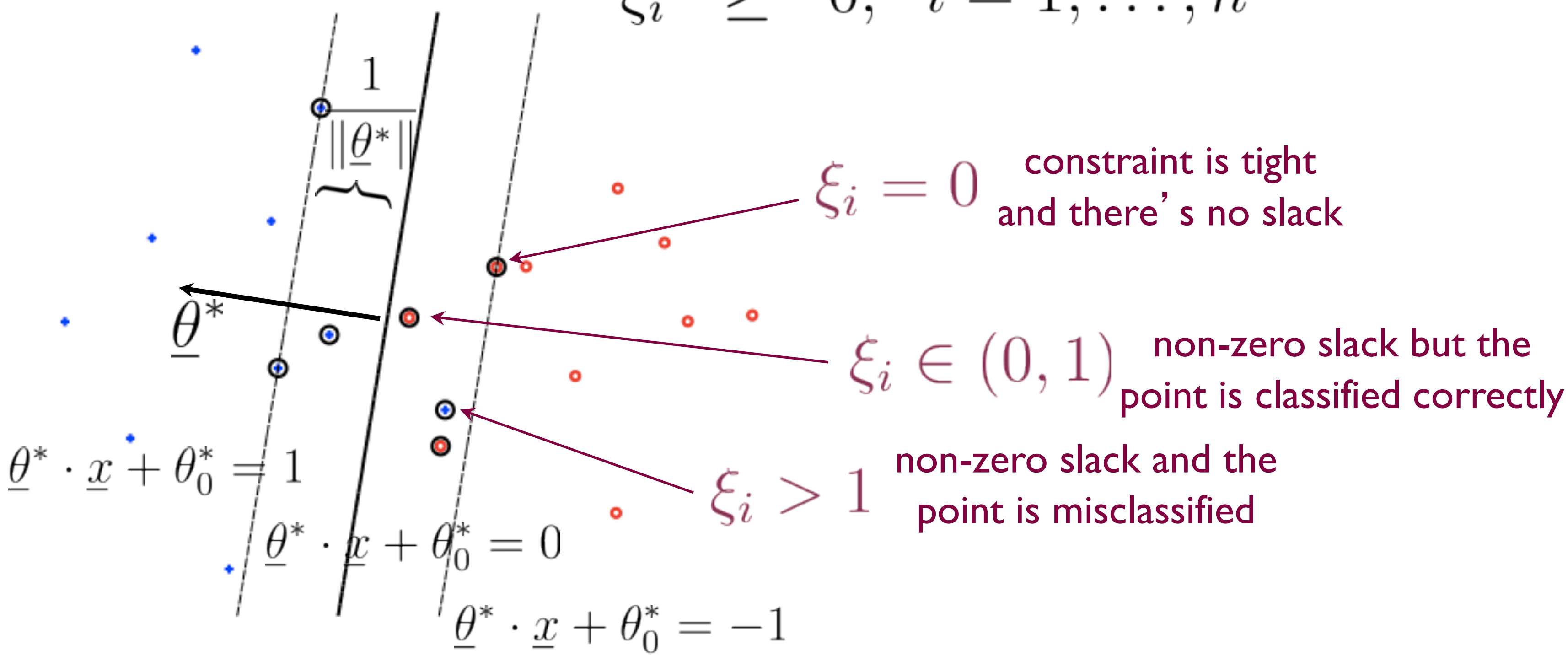
$$\xi_i \;\geq\; 0, \quad i=1,\ldots,n$$



Soft-margin SVM

- The solution now has three types of support vectors

$$\begin{aligned} \text{minimize} \quad \frac{1}{2}\|\underline{\theta}\|^2 \;\; &+ \;\; C\sum_{i=1}^{n}\xi_i \quad \text{subject to} \\ y_i(\underline{\theta}\cdot\underline{x}_i+\theta_0) \;\; &\geq \;\; 1-\xi_i, \quad i=1,\ldots,n \\ \xi_i \;\; &\geq \;\; 0, \quad i=1,\ldots,n \end{aligned}$$

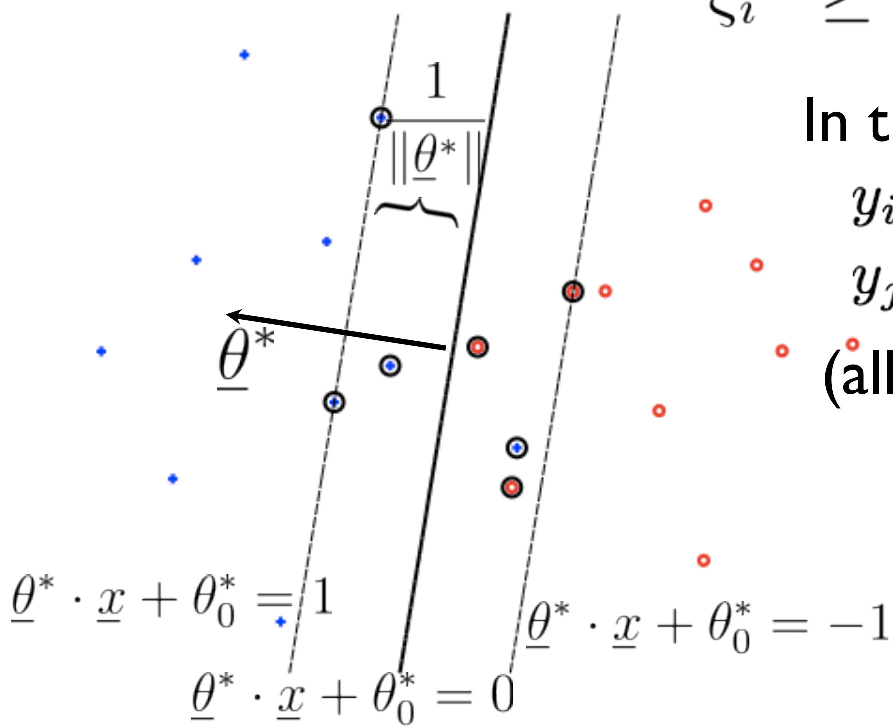


Soft-margin SVM

- Relaxed quadratic programming problem

$$\underset{\underline{\theta}, \theta_0, \xi}{\text{minimize}} \quad \frac{1}{2}\|\underline{\theta}\|^2 \quad + \quad C\sum_{i=1}^{n} \xi_i \quad \text{subject to}$$

$$\begin{aligned} y_i(\underline{\theta} \cdot \underline{x}_i + \theta_0) &\geq 1 - \xi_i, \quad i = 1, \ldots, n \\ \xi_i &\geq 0, \quad i = 1, \ldots, n \end{aligned}$$



In the solution, we will have either

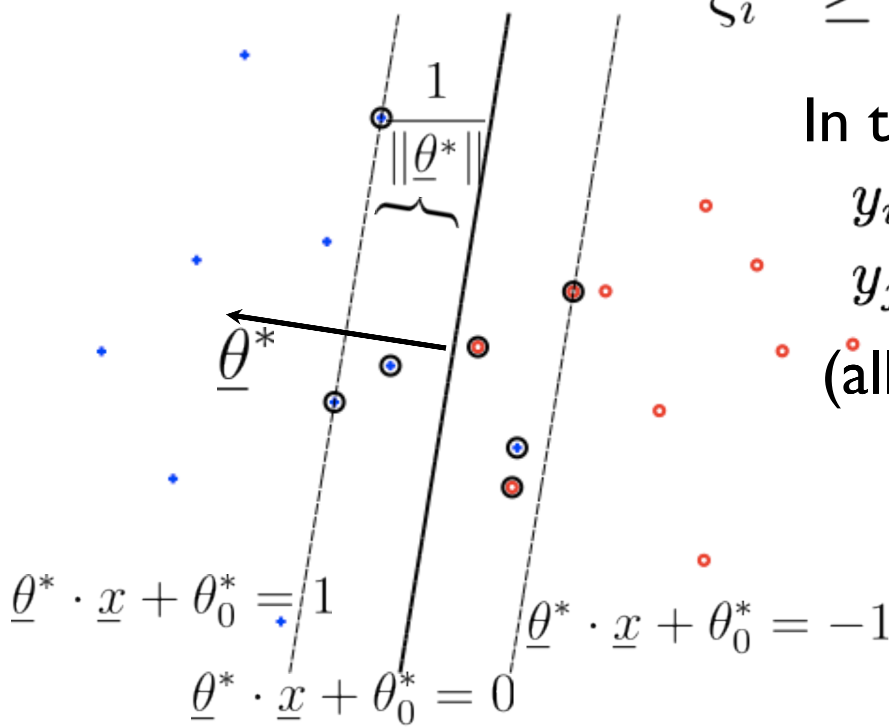
$$\begin{aligned} y_i(\underline{\theta}^* \cdot \underline{x}_i + \theta_0^*) &= 1 - \xi_i^*, \quad \xi_i^* \geq 0 \\ y_j(\underline{\theta}^* \cdot \underline{x}_j + \theta_0^*) &> 1, \quad \xi_j^* = 0 \end{aligned}$$

(all the active constraints are SVs)

Soft-margin SVM

- Relaxed quadratic programming problem

$$\begin{aligned} \underset{\underline{\theta}, \theta_0, \xi}{\text{minimize}} \quad \frac{1}{2}\|\underline{\theta}\|^2 \quad + \quad C\sum_{i=1}^{n} \xi_i \quad \text{subject to} \\ y_i(\underline{\theta} \cdot \underline{x}_i + \theta_0) \quad &\geq \quad 1 - \xi_i, \quad i = 1, \ldots, n \\ \xi_i \quad &\geq \quad 0, \quad i = 1, \ldots, n \end{aligned}$$



In the solution, we will have either

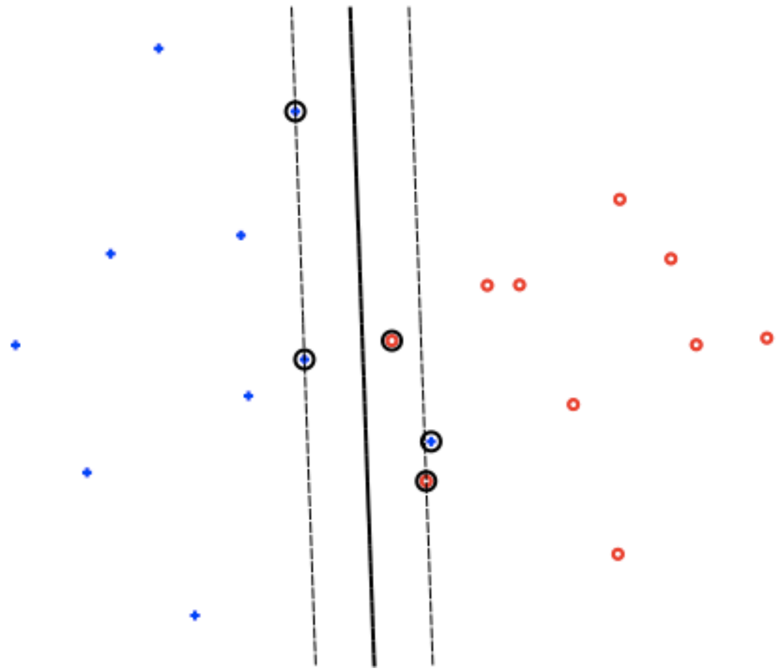
$$\begin{aligned} y_i(\underline{\theta}^* \cdot \underline{x}_i + \theta_0^*) \quad &= 1 - \xi_i^*, \quad \xi_i^* \geq 0 \\ y_j(\underline{\theta}^* \cdot \underline{x}_j + \theta_0^*) \quad &> 1, \qquad\quad \xi_j^* = 0 \end{aligned}$$

(all the active constraints are SVs)

The solution need not be unique in terms of θ_0, ξ

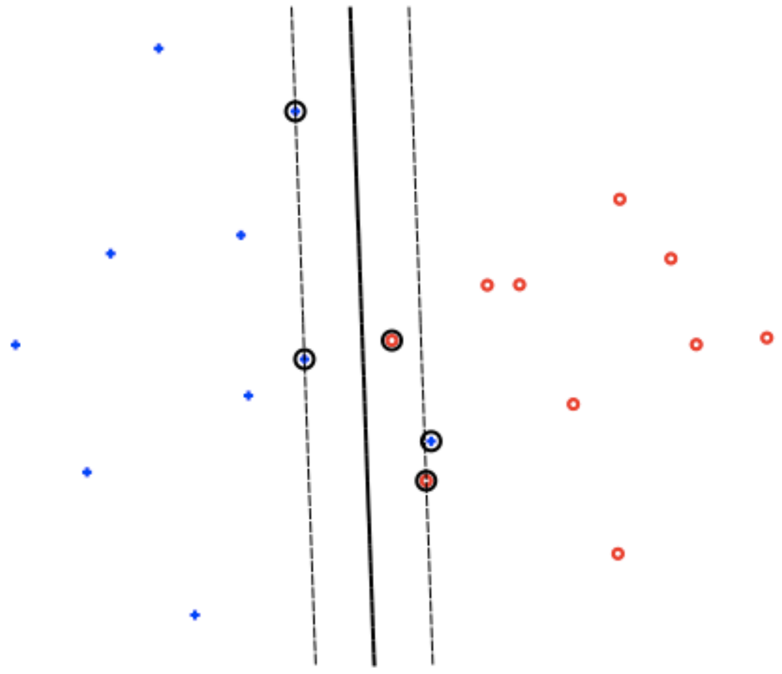
Soft-margin SVM: Examples

- C=100



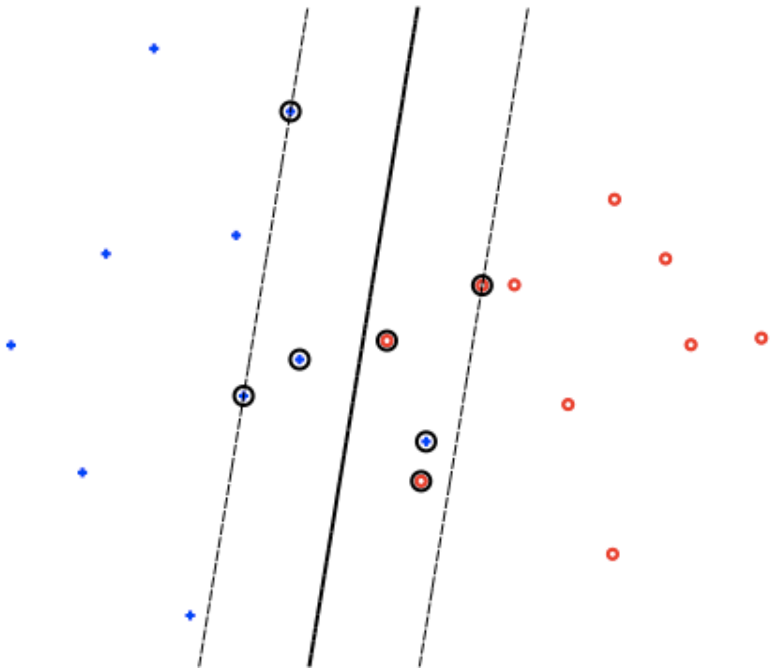
Soft-margin SVM: Examples

- C=10



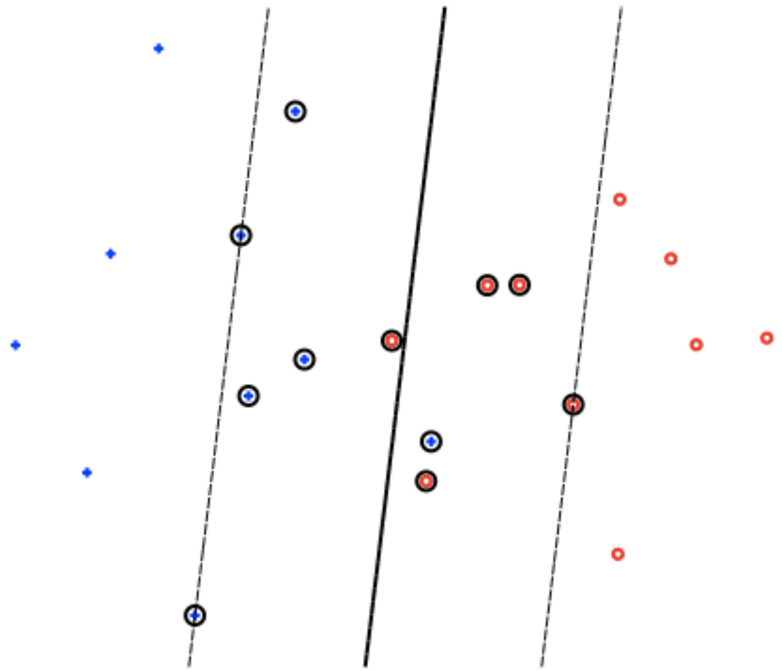
Soft-margin SVM: Examples

- C=1



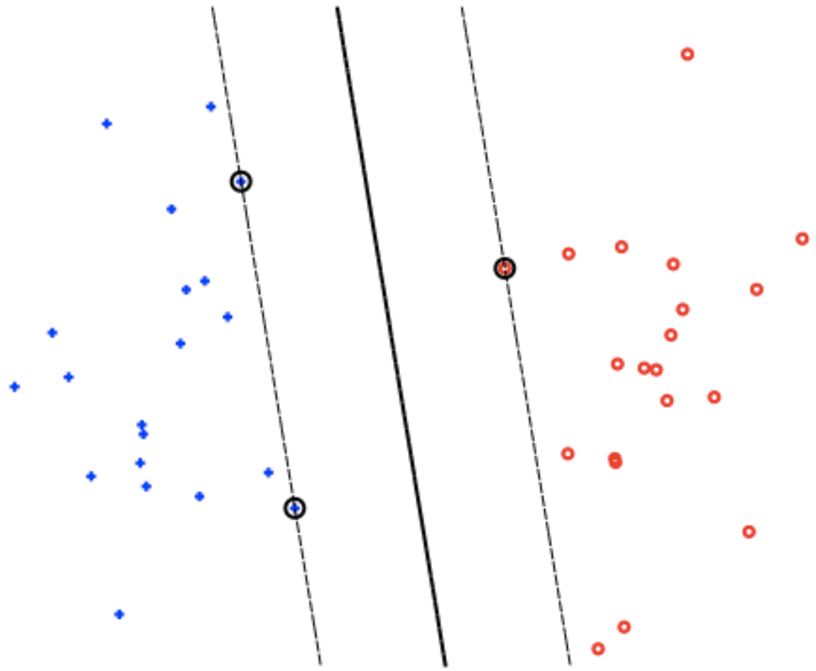
Soft-margin SVM: Examples

- C=0.1



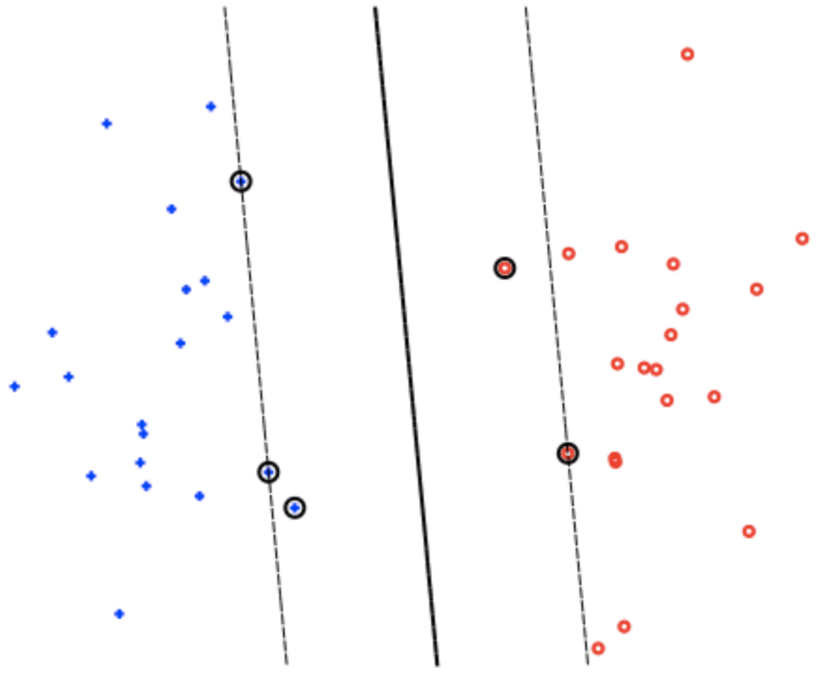
Soft-margin SVM: Examples

- C potentially affects the solution even in the separable case
- C = 1



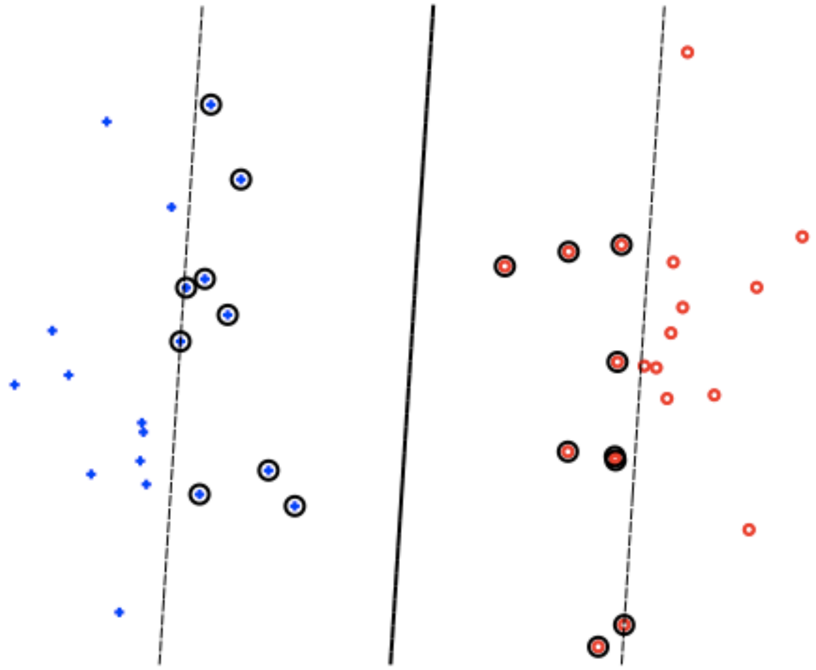
Soft-margin SVM: Examples

- C potentially affects the solution even in the separable case
- C = 0.1



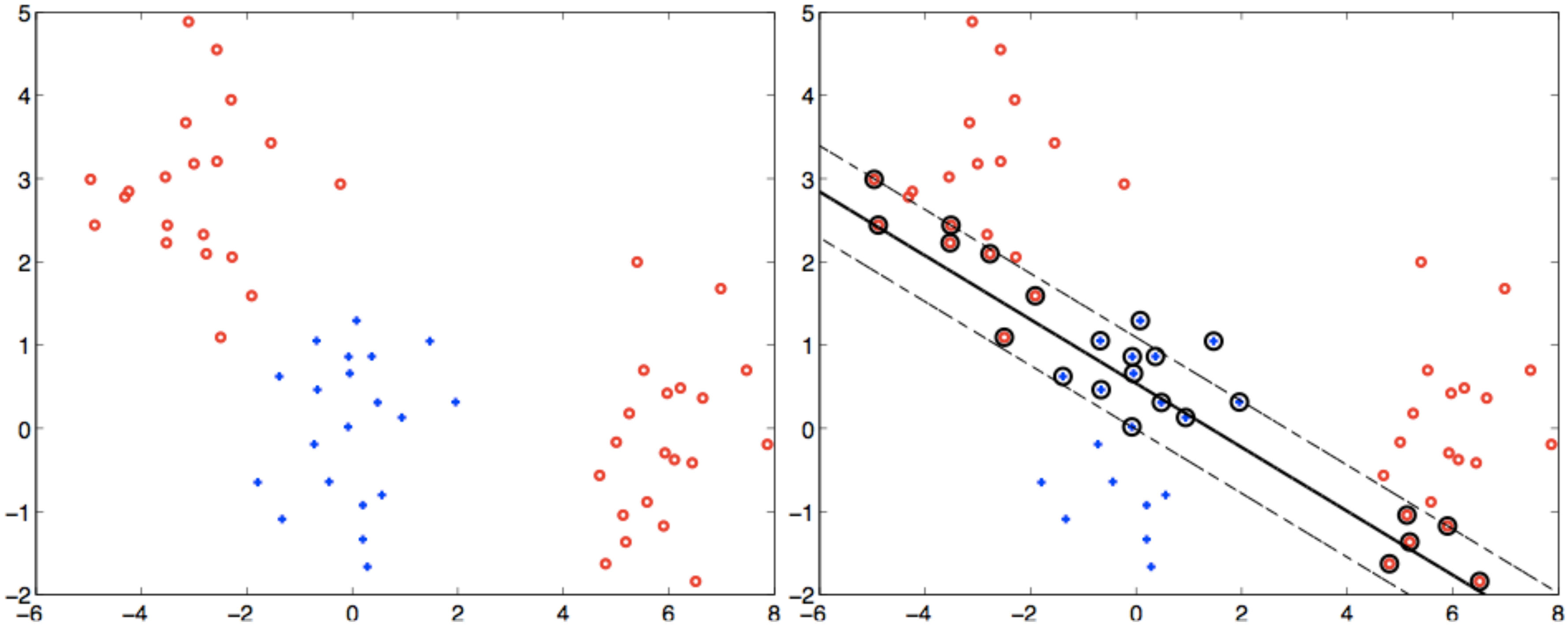
Soft-margin SVM: Examples

- C potentially affects the solution even in the separable case
- C = 0.01



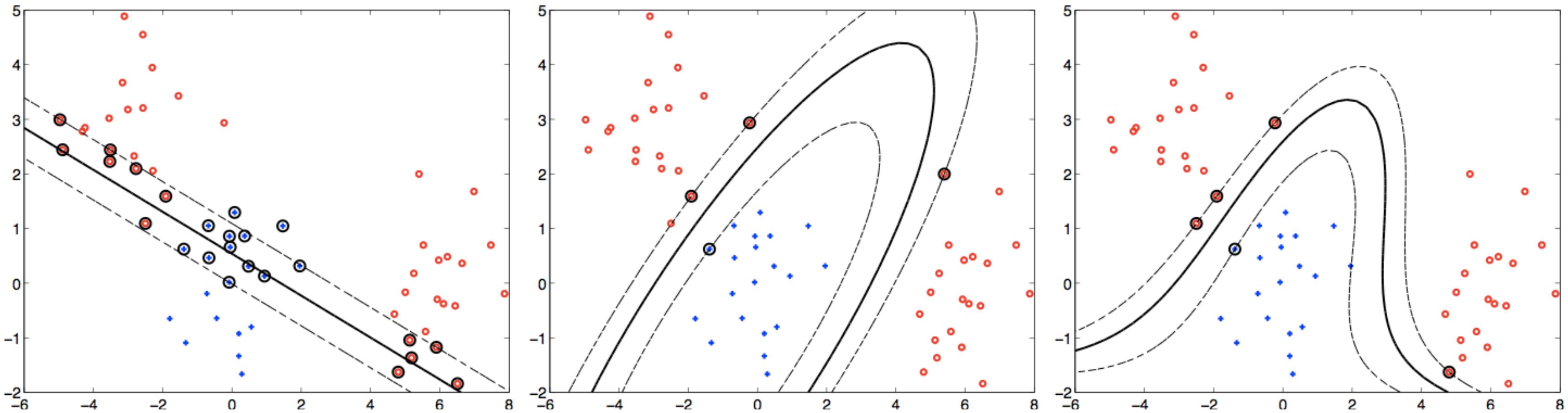
Beyond linear classifiers...

- Many problems are not solved well by a linear classifier even if we allow misclassified examples (SVM with slack)
- E.g., data from experiments typically involve "clusters" of different types of examples



Non-linear classifiers

- Many (low dimensional) problems are not solved well by a linear classifier even with slack
- By mapping examples to feature vectors, and maximizing a linear margin in the feature space, we obtain non-linear margin curves in the original space
- By using non-linear feature mappings we get more powerful sets of classifiers



linear features 2nd order features 3rd order features

Non-linear feature mappings

- The easiest way to make the classifier more powerful is to add non-linear coordinates to the feature vectors
- The classifier is still linear in the parameters, $\underline{\vartheta}$, not inputs, $\underline{x}$

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \longrightarrow \quad \underline{\phi}(\underline{x}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ \sqrt{2}x_1 x_2 \\ x_2^2 \end{bmatrix}$$

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{x} + \theta_0\big)$$

linear classifier

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{\phi}(\underline{x}) + \theta_0\big)$$

non-linear classifier

Non-linear feature mappings

- The easiest way to make the classifier more powerful is to add non-linear coordinates to the feature vectors
- The classifier is still linear in the parameters, $\underline{\vartheta}$, not inputs, $\underline{x}$

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \longrightarrow \quad \phi(\underline{x}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$$

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{x} + \theta_0\big)$$

linear classifier

$$\underline{\theta} \cdot \underline{x} + \theta_0 = 0$$

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{\phi}(\underline{x}) + \theta_0\big)$$

non-linear classifier

Non-linear feature mappings

- The easiest way to make the classifier more powerful is to add non-linear coordinates to the feature vectors
- The classifier is still linear in the parameters, $\underline{\vartheta}$, not inputs, $\underline{x}$

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \longrightarrow \quad \phi(\underline{x}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$$

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{x} + \theta_0\big)$$

linear classifier

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \phi(\underline{x}) + \theta_0\big)$$

non-linear classifier

$$\underline{\theta} \cdot \underline{x} + \theta_0 = 0$$

$$\theta_1 x_1 + \theta_2 x_2 + \theta_0 = 0$$

linear decision boundary

Non-linear feature mappings

- The easiest way to make the classifier more powerful is to add non-linear coordinates to the feature vectors
- The classifier is still linear in the parameters, $\underline{\vartheta}$, not inputs, $\underline{x}$

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \longrightarrow \quad \underline{\phi}(\underline{x}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$$

$$f(\underline{x};\underline{\theta},\theta_0) = \text{sign}\big(\underline{\theta}\cdot\underline{x}+\theta_0\big)$$

linear classifier

$$f(\underline{x};\underline{\theta},\theta_0) = \text{sign}\big(\underline{\theta}\cdot\underline{\phi}(\underline{x})+\theta_0\big)$$

non-linear classifier

$$\underline{\theta}\cdot\underline{\phi}(\underline{x})+\theta_0 = 0$$

Non-linear feature mappings

- The easiest way to make the classifier more powerful is to add non-linear coordinates to the feature vectors
- The classifier is still linear in the parameters, $\underline{\vartheta}$, not inputs, $\underline{x}$

$$\underline{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \longrightarrow \quad \phi(\underline{x}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}$$

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{x} + \theta_0\big)$$

linear classifier

$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{\phi}(\underline{x}) + \theta_0\big)$$

non-linear classifier

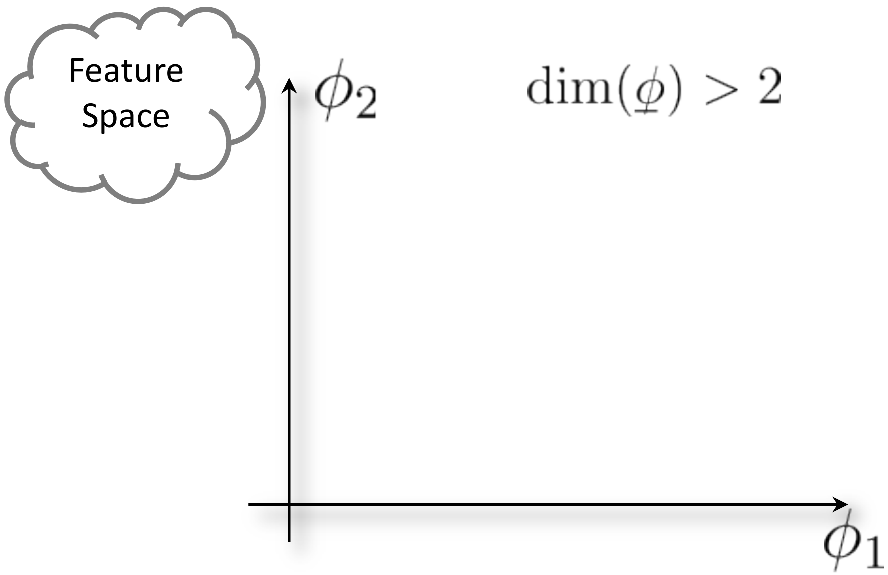
$$\underline{\theta} \cdot \underline{\phi}(\underline{x}) + \theta_0 = 0$$

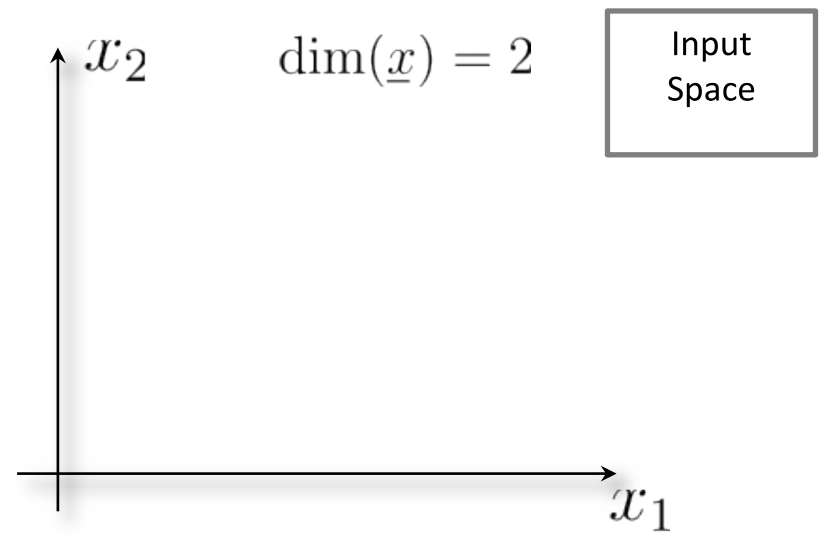
$$\theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 \sqrt{2} x_1 x_2 + \theta_5 x_2^2 + \theta_0 = 0$$

non-linear decision boundary

Non-linear feature mappings

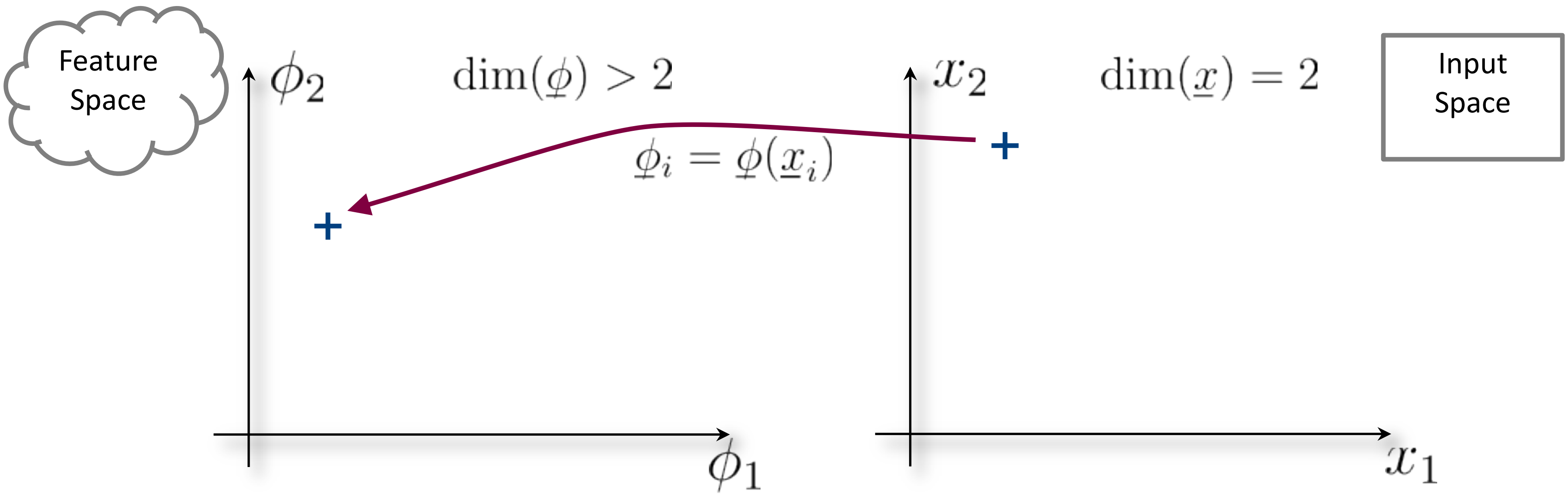
- By expanding the feature coordinates, we still have a linear classifier in the new feature coordinates but a non-linear classifier in the original coordinates





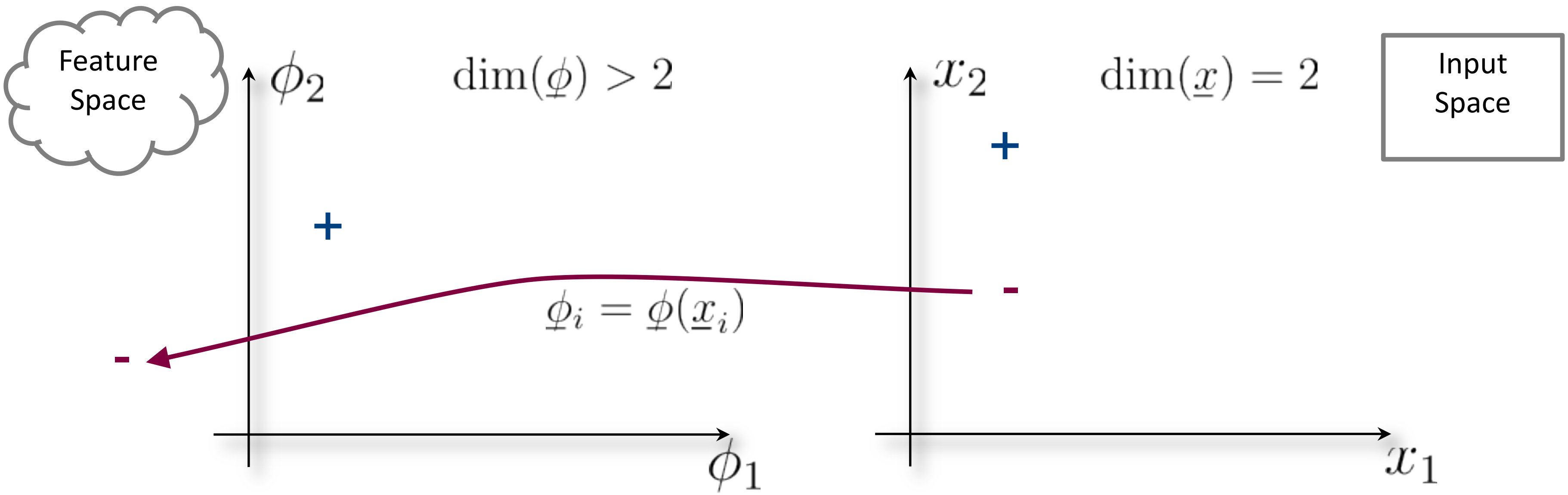
Non-linear feature mappings

- By expanding the feature coordinates, we still have a linear classifier in the new feature coordinates but a non-linear classifier in the original coordinates



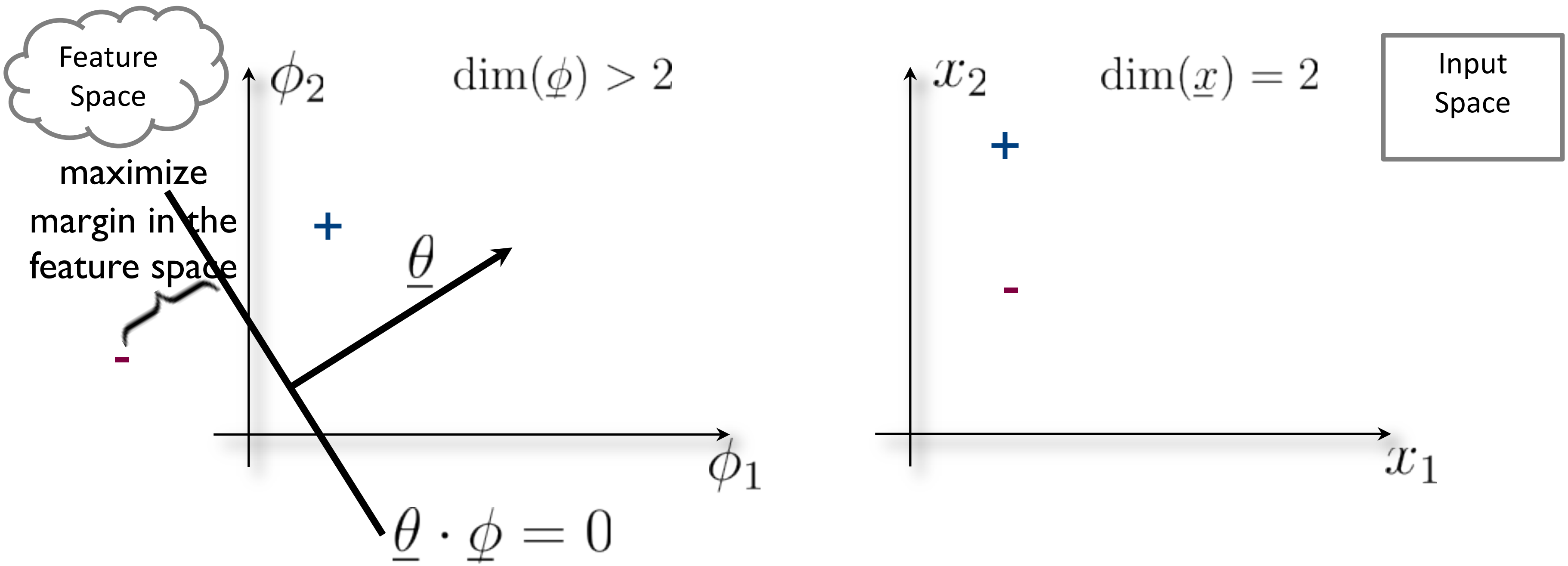
Non-linear feature mappings

- By expanding the feature coordinates, we still have a linear classifier in the new feature coordinates but a non-linear classifier in the original coordinates



Non-linear feature mappings

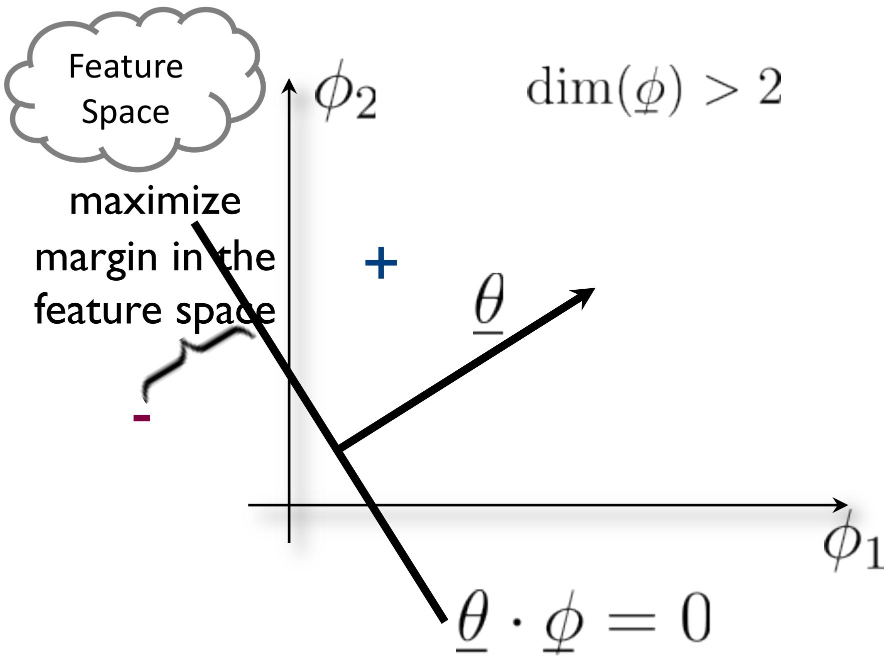
- By expanding the feature coordinates, we still have a linear classifier in the new feature coordinates but a non-linear classifier in the original coordinates

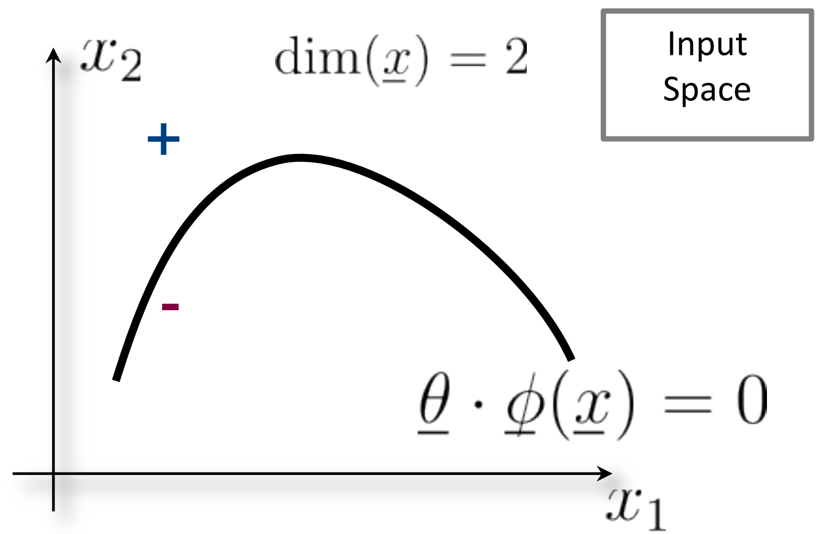


$$f(\underline{\phi}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{\phi} + \theta_0\big)$$

Non-linear feature mappings

- By expanding the feature coordinates, we still have a linear classifier in the new feature coordinates but a non-linear classifier in the original coordinates





$$f(\underline{x}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{\phi}(\underline{x}) + \theta_0\big)$$

$$f(\underline{\phi}; \underline{\theta}, \theta_0) = \text{sign}\big(\underline{\theta} \cdot \underline{\phi} + \theta_0\big)$$

Learning non-linear classifiers

- We can apply the same SVM formulation, just replacing the input examples, x_i, with (higher dimensional) feature vectors, $\Phi(x_i)$.

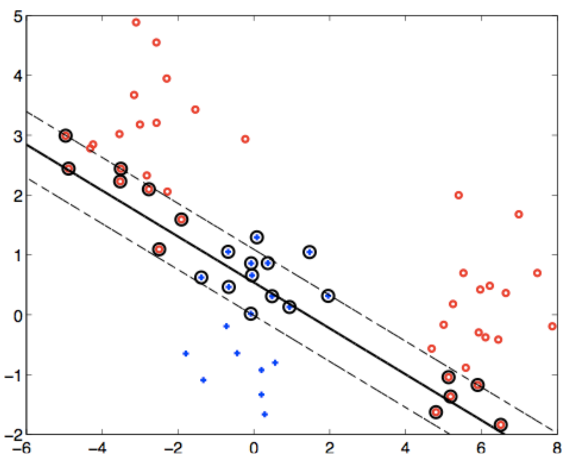
$$\text{minimize} \quad \frac{1}{2}\|\underline{\theta}\|^2 \quad + \quad C\sum_{i=1}^{n}\xi_i \quad \text{subject to}$$

$$\begin{aligned} y_i(\underline{\theta} \cdot \underline{\phi(\underline{x}_i)} + \theta_0) &\geq 1 - \xi_i, \quad i = 1, \ldots, n \\ \xi_i &\geq 0, \quad i = 1, \ldots, n \end{aligned}$$

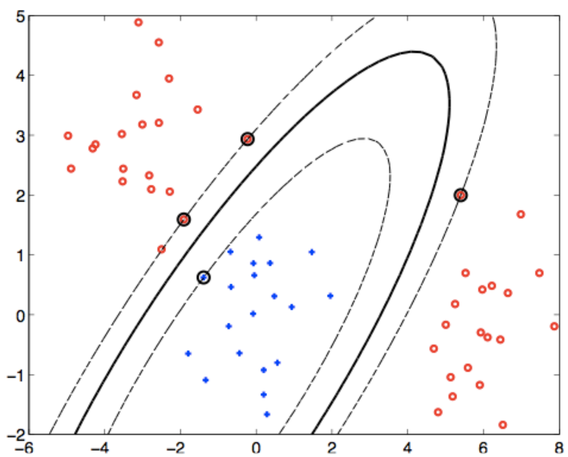
- Note that the cost of solving this quadratic programming problem increases with the dimension of the feature vectors (BUT, we will avoid this problem by solving the dual instead – still only *n* constraints.)

Problems to resolve

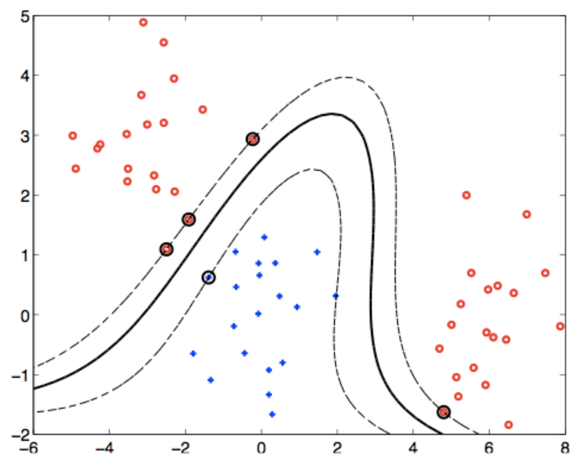
- By using non-linear feature mappings we get more powerful sets of classifiers
- Computational efficiency?
  - the cost of using higher dimensional feature vectors (seems to) increase with the dimension
- Model selection?
  - how do we choose among different feature mappings?



linear features

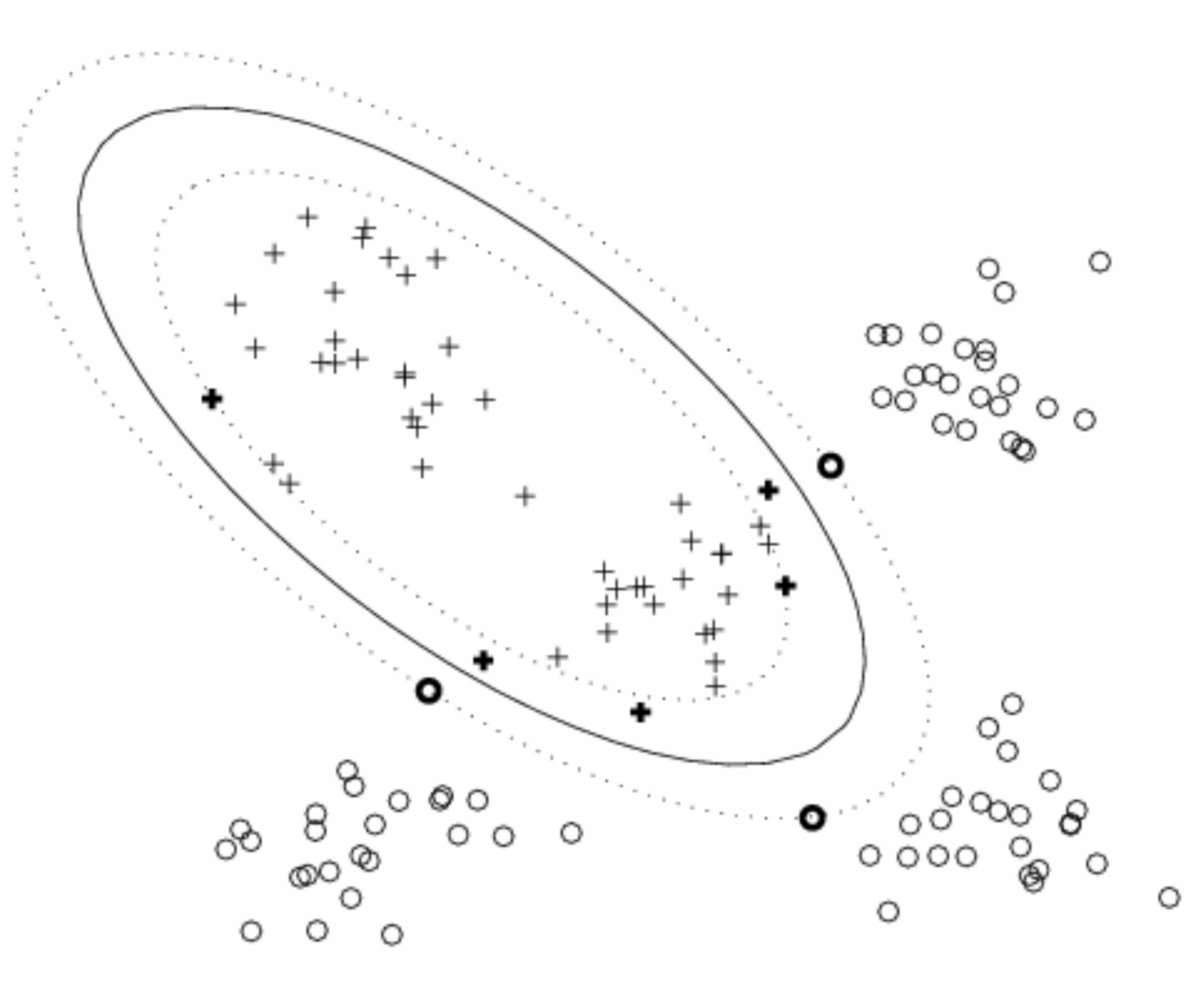


2nd order features

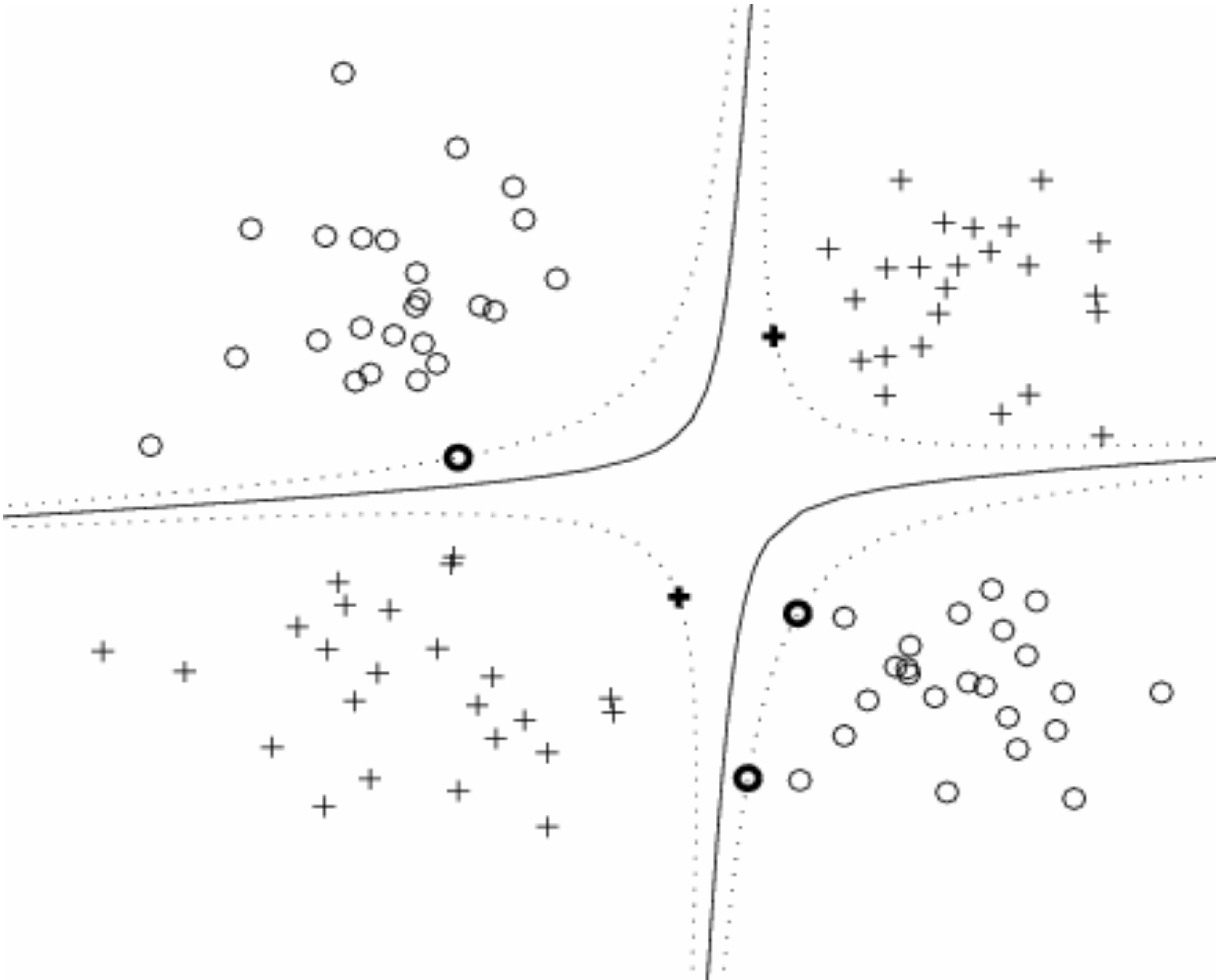


3rd order features

Quadratic kernel



Quadratic kernel



Non-linear perceptron, kernels

- Non-linear feature mappings can be dealt with more efficiently through their inner products or "kernels"
- We will begin by turning the perceptron classifier with non-linear features into a "kernel perceptron"
- For simplicity, we drop the offset parameter

$$f(\underline{x};\underline{\theta}) = \text{sign}\big(\underline{\theta} \cdot \underline{\phi}(\underline{x})\big)$$

$$\text{Initialize: } \underline{\theta} = 0$$

$$\text{For } t = 1, 2, \ldots$$

(applied in a sequence or repeatedly over a fixed training set)

$$\text{if } y_t(\underline{\theta} \cdot \underline{\phi}(\underline{x}_t)) \leq 0 \text{ (mistake)}$$

$$\underline{\theta} \leftarrow \underline{\theta} + y_t \underline{\phi}(\underline{x}_t)$$

On perceptron updates

- Each update adds $y_t \underline{\phi}(\underline{x}_t)$ to the parameter vector
- Repeated updates on the same example simply result in adding the same term multiple times
- We can therefore write the current perceptron solution as a function of how many times we performed an update on each training example

$$\underline{\theta} = \sum_{i=1}^{n} \alpha_i \, y_i \underline{\phi}(\underline{x}_i)$$

$$\alpha_i \in \{0, 1, \ldots\}, \;\; \sum_{i=1}^{n} \alpha_i = \# \text{ of mistakes}$$

where α_i is the number of mistakes made on example *i*.

Kernel perceptron

- By subbing in this "count" representation of $\underline{\vartheta}$, we can write the perceptron algorithm entirely in terms of inner products between the feature vectors

$$f(\underline{x};\underline{\theta}) = \text{sign}\big(\underline{\theta}\cdot\underline{\phi}(\underline{x})\big) = \text{sign}\Big(\sum_{i=1}^{n}\alpha_i y_i[\underline{\phi}(\underline{x}_i)\cdot\underline{\phi}(\underline{x})]\Big)$$

$$\text{Initialize: } \alpha_i = 0,\ i = 1,\ldots,n$$

$$\text{Repeat for } t = 1,\ldots,n$$

$$\text{if } y_t\Big(\sum_{i=1}^{n}\alpha_i y_i[\underline{\phi}(\underline{x}_i)\cdot\underline{\phi}(\underline{x}_t)]\Big) \le 0 \ \text{(mistake)}$$

$$\alpha_t \leftarrow \alpha_t + 1$$

Kernel perceptron

- By subbing in this "count" representation of $\underline{\vartheta}$, we can write the perceptron algorithm entirely in terms of inner products between the feature vectors

$$f(\underline{x};\underline{\theta}) = \text{sign}\big(\underline{\theta}\cdot\underline{\phi}(\underline{x})\big) = \text{sign}\Big(\sum_{i=1}^{n}\alpha_i y_i[\underline{\phi}(\underline{x}_i)\cdot\underline{\phi}(\underline{x})]\Big)$$

$$\text{Initialize: } \alpha_i = 0,\ i = 1,\ldots,n$$

$$\text{Repeat for } t = 1,\ldots,n$$

$$\text{if } y_t\big(\textstyle\sum_{i=1}^{n}\alpha_i y_i[\underline{\phi}(\underline{x}_i)\cdot\underline{\phi}(\underline{x}_t)]\big) \le 0 \ \text{(mistake)}$$

$$\alpha_t \leftarrow \alpha_t + 1$$

Why inner products?

- For some feature mappings, the inner products can be computed efficiently, without expanding the feature vectors!

$$\underline{\phi}(\underline{x}) \cdot \underline{\phi}(\underline{x}') = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \cdot \begin{bmatrix} x_1' \\ x_2' \\ x_1'^{\,2} \\ \sqrt{2}x_1'x_2' \\ x_2'^{\,2} \end{bmatrix}$$

$$= (x_1x_1') + (x_2x_2') + (x_1x_1')^2 + 2(x_1x_1')(x_2x_2') + (x_2x_2')^2$$

$$= (x_1x_1' + x_2x_2') + (x_1x_1' + x_2x_2')^2$$

$$= (\underline{x} \cdot \underline{x}') + (\underline{x} \cdot \underline{x}')^2$$

Why inner products?

- Instead of explicitly constructing feature vectors, we can try to evaluate their inner product or "kernel" directly.

$$\underline{\phi}(\underline{x}) \cdot \underline{\phi}(\underline{x}') = \begin{bmatrix} \\ ? \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \\ ? \\ \\ \end{bmatrix}$$

$$= (\underline{x} \cdot \underline{x}') + (\underline{x} \cdot \underline{x}')^2$$

- What is $\underline{\phi}(\underline{x})$ such that the above holds?

Why inner products?

- Instead of explicitly constructing feature vectors, we can try to explicate their inner product or "kernel"

$$\underline{\phi}(\underline{x}) \cdot \underline{\phi}(\underline{x}') = \begin{bmatrix} \\ ? \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \\ ? \\ \\ \end{bmatrix}$$

$$= (\underline{x} \cdot \underline{x}') + (\underline{x} \cdot \underline{x}')^2 + (\underline{x} \cdot \underline{x}')^3 + (\underline{x} \cdot \underline{x}')^4$$

- What is $\underline{\phi}(\underline{x})$ now? Does it even exist?

Feature mappings and kernels

- In the kernel perceptron algorithm, the feature vectors appear only as inner products
- Instead of explicitly constructing feature vectors, we can try to evaluate their inner product or kernel

- $K : \mathcal{R}^d \times \mathcal{R}^d \rightarrow \mathcal{R}$ is a kernel function if there exists a feature mapping such that:

$$K(\underline{x}, \underline{x}') = \underline{\phi}(\underline{x}) \cdot \underline{\phi}(\underline{x}')$$

Feature mappings and kernels

- In the kernel perceptron algorithm, the feature vectors appear only as inner products
- Instead of explicitly constructing feature vectors, we can try to explicate their inner product or kernel
- $K : \mathcal{R}^d \times \mathcal{R}^d \rightarrow \mathcal{R}$ is a kernel function if there exists a feature mapping, *Φ(x)*, such that

$$K(\underline{x}, \underline{x}') = \underline{\phi}(\underline{x}) \cdot \underline{\phi}(\underline{x}')$$

- Examples of polynomial kernels

$$\begin{aligned}
K(\underline{x}, \underline{x}') &= (\underline{x} \cdot \underline{x}') \\
K(\underline{x}, \underline{x}') &= (\underline{x} \cdot \underline{x}') + (\underline{x} \cdot \underline{x}')^2 \\
K(\underline{x}, \underline{x}') &= (\underline{x} \cdot \underline{x}') + (\underline{x} \cdot \underline{x}')^2 + (\underline{x} \cdot \underline{x}')^3 \\
K(\underline{x}, \underline{x}') &= (1 + \underline{x} \cdot \underline{x}')^p, \quad p = 1, 2, \ldots
\end{aligned}$$

Polynomial decision surfaces

To get a decision surface which is an arbitrary polynomial of order p:

Let $\Phi(\mathbf{x})$ consist of all terms of order $\leq$ $\mathbf{p}$, such as $\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3^{p-3}$.

$\mathbf{k(x,z) = \Phi(x) \cdot \Phi(z) = (1 + x \cdot z)^p}$

Kernel Perceptron recap

Learning in the higher-dimensional feature space:

```
w = 0
while some  y(w·Φ(x)) ≤ 0:
      w = w + y Φ(x)
```

Everything works as before; final `w` is a weighted sum of various `Φ(x)`.

Problem: number of features has now increased dramatically.
OCR data: from 784 to 307,720!

The kernel trick

[Aizenman, Braverman, Rozonoer, 1964]

No need to explicitly write out $\Phi(\mathbf{x})$!

The only time we ever access it is to compute a dot product $\mathbf{w} \cdot \Phi(\mathbf{x})$.

If $\mathbf{w} = a_1 \Phi(\mathbf{x}^{(1)}) + a_2 \Phi(\mathbf{x}^{(2)}) + a_3 \Phi(\mathbf{x}^{(3)})$ then $\mathbf{w} \cdot \Phi(\mathbf{x})$ = (weighted) sum of dot products, each of the form $\Phi(\mathbf{x}) \cdot \Phi(\mathbf{x}^{(i)})$.

Can we compute such dot products without writing out the $\Phi(\mathbf{x})$'s?

The kernel trick

Polynomial kernel, p=2:

In 2-d:

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{z})$$
$$= (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2) \cdot (1, \sqrt{2}z_1, \sqrt{2}z_2, z_1^2, z_2^2, \sqrt{2}z_1z_2)$$
$$= 1 + 2x_1z_1 + 2x_2z_2 + x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2$$
$$= (1 + x_1z_1 + x_2z_2)^2$$
$$= (1 + \mathbf{x} \cdot \mathbf{z})^2$$

In d dimensions:

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{z})$$
$$= (1, \sqrt{2}x_1, \ldots, \sqrt{2}x_d, x_1^2, \ldots, x_d^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \ldots, \sqrt{2}x_{d-1}x_d) \cdot$$
$$(1, \sqrt{2}z_1, \ldots, \sqrt{2}z_d, z_1^2, \ldots, z_d^2, \sqrt{2}z_1z_2, \sqrt{2}z_1z_3, \ldots, \sqrt{2}z_{d-1}z_d)$$
$$= (1 + x_1z_1 + x_2z_2 + \ldots + x_dz_d)^2$$
$$= (1 + \mathbf{x} \cdot \mathbf{z})^2$$

Computing dot products in the 307,720-dimensional feature space takes time proportional to just 784, the original dimension!

Never need to write out $\Phi(\mathbf{x})$.

Need $\mathbf{w}$ – but since it's a linear combination of (kernelized) data points, just store the coefficients.

Kernel trick

Why does it work?

1. The only time we ever use the data is to compute dot products $\mathbf{w} \cdot \Phi(\mathbf{x})$.

2. And $\mathbf{w}$ itself is a linear combination of $\Phi(\mathbf{x})$'s. If $\mathbf{w} = a_1\, \Phi(\mathbf{x}^{(1)}) + a_{22}\, \Phi(\mathbf{x}^{(22)}) + a_{37}\, \Phi(\mathbf{x}^{(37)})$ store it as $[(1,a_1),\ (22,a_{22}),\ (37,a_{37})]$

3. Dot products $\Phi(\mathbf{x}) \cdot \Phi(\mathbf{z})$ can be computed very efficiently.

Valid kernels: composition rules

- We can construct valid kernels from simple components
- For any function $f : R^d \rightarrow R$, if K_1 is a kernel, then so is

1) $K(\underline{x}, \underline{x}') = f(\underline{x}) K_1(\underline{x}, \underline{x}') f(\underline{x}')$

- The set of kernel functions is closed under addition and multiplication: if K_1 and K_2 are kernels, then so are

2) $K(\underline{x}, \underline{x}') = K_1(\underline{x}, \underline{x}') + K_2(\underline{x}, \underline{x}')$

3) $K(\underline{x}, \underline{x}') = K_1(\underline{x}, \underline{x}') K_2(\underline{x}, \underline{x}')$

- The composition rules are also helpful in verifying that a kernel is valid (i.e., corresponds to an inner product of some feature vectors)

Radial basis kernel

- The feature "vectors" corresponding to kernels may also be infinite dimensional (i.e., functions)
- This is the case, e.g., for the radial basis kernel

$$K(\underline{x}, \underline{x}') = \exp\left(-\beta \|\underline{x} - \underline{x}'\|^2 \right), \quad \beta > 0$$

- Any distinct set of training points, regardless of their labels, are separable using this kernel function!

Radial basis function (RBF) kernel

- The feature "vectors" corresponding to kernels may also be infinite dimensional (i.e., functions)
- This is the case, e.g., for the radial basis kernel

$$K(\underline{x}, \underline{x}') = \exp\left(-\beta \|\underline{x} - \underline{x}'\|^2 \right), \quad \beta > 0$$

- Any distinct set of training points, regardless of their labels, are separable using this kernel function!
- We can use the composition rules to show that this is indeed a valid kernel

$$\begin{aligned}
\exp\{-\beta\|\underline{x} - \underline{x}'\|^2\} &= \exp\{-\beta\underline{x}\cdot\underline{x} + 2\beta\underline{x}\cdot\underline{x}' - \beta\underline{x}'\cdot\underline{x}'\} \\
&= \overbrace{\exp\{-\beta\underline{x}\cdot\underline{x}\}}^{f(\underline{x})} \exp\{2\beta\underline{x}\cdot\underline{x}'\} \overbrace{\exp\{-\beta\underline{x}'\cdot\underline{x}'\}}^{f(\underline{x}')} \\
&= f(\underline{x}) \left(1 + 2\beta(\underline{x}\cdot\underline{x}') + \ldots \right) f(\underline{x}')
\end{aligned}$$

Valid kernels

- A kernel function is valid (is a kernel) if there exists some feature mapping such that

$$K(\underline{x}, \underline{x}') \;=\; \underline{\phi}(\underline{x}) \cdot \underline{\phi}(\underline{x}')$$

- We can verify this, e.g., via the composition rules
- Equivalently, a kernel is valid if it is symmetric and for all training sets, the Gram matrix:

$$\begin{bmatrix} K(\underline{x}_1, \underline{x}_1) & \cdots & K(\underline{x}_1, \underline{x}_n) \\ \cdots & \cdots & \cdots \\ K(\underline{x}_n, \underline{x}_1) & \cdots & K(\underline{x}_n, \underline{x}_n) \end{bmatrix}$$

is positive semi-definite.

Kernel functions

As one varies Φ, what kinds of similarity measures $\mathbf{K}$ are possible?

Any $\mathbf{K}$ which satisfies a technical condition (positive semi-definiteness) will correspond to some embedding $\Phi(\mathbf{x})$.

So: don't worry about Φ and just pick a similarity measure $\mathbf{K}$ which suits the data at hand.

Popular choice: *Gaussian kernel* (typical choice for RBF)

$$k(x,x') = \exp(-\|x - x'\|^2/s^2)$$

Kernel perceptron revisited

- We can now apply the kernel perceptron algorithm without ever expanding the feature vectors!

$$f(\underline{x};\alpha) = \mathrm{sign}\Big(\sum_{i=1}^{n} \alpha_i y_i \underline{K(\underline{x}_i, \underline{x})} \Big)$$

Initialize: $\alpha_i = 0,\ i = 1, \ldots, n$

Repeat for $t = 1, \ldots, n$

$\quad$ if $y_t\big(\sum_{i=1}^{n} \alpha_i y_i \underline{K(\underline{x}_i, \underline{x}_t)} \big) \leq 0$ (mistake)

$\quad\quad \alpha_t \leftarrow \alpha_t + 1$

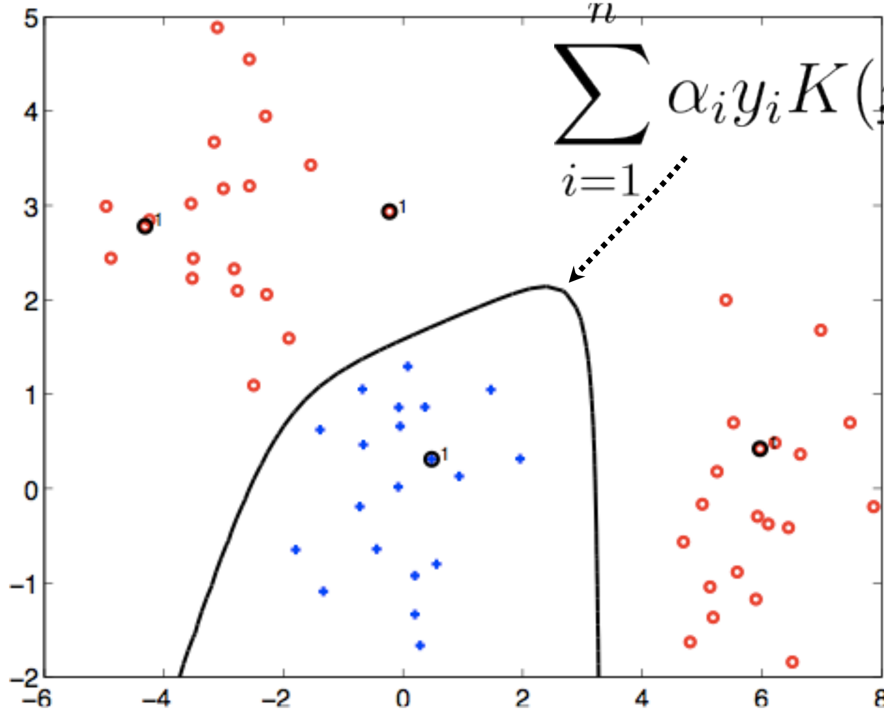
Kernel perceptron: example

- With a radial basis kernel

$$f(\underline{x};\alpha) = \mathrm{sign}\Big(\sum_{i=1}^{n} \alpha_i y_i K(\underline{x}_i, \underline{x}) \Big)$$

Decision surface:

$$\sum_{i=1}^{n} \alpha_i y_i K(\underline{x}_i, \underline{x}) = 0$$



Only 4 mistakes!

Kernel SVM

- Kernel SVM: implicitly find the max-margin linear separator in the feature space, e.g., corresponding to the radial basis kernel

$$f(\underline{x};\alpha) = \text{sign}\Big(\sum_{i=1}^{n} \alpha_i y_i K(\underline{x}_i, \underline{x}) + \theta_0 \Big)$$