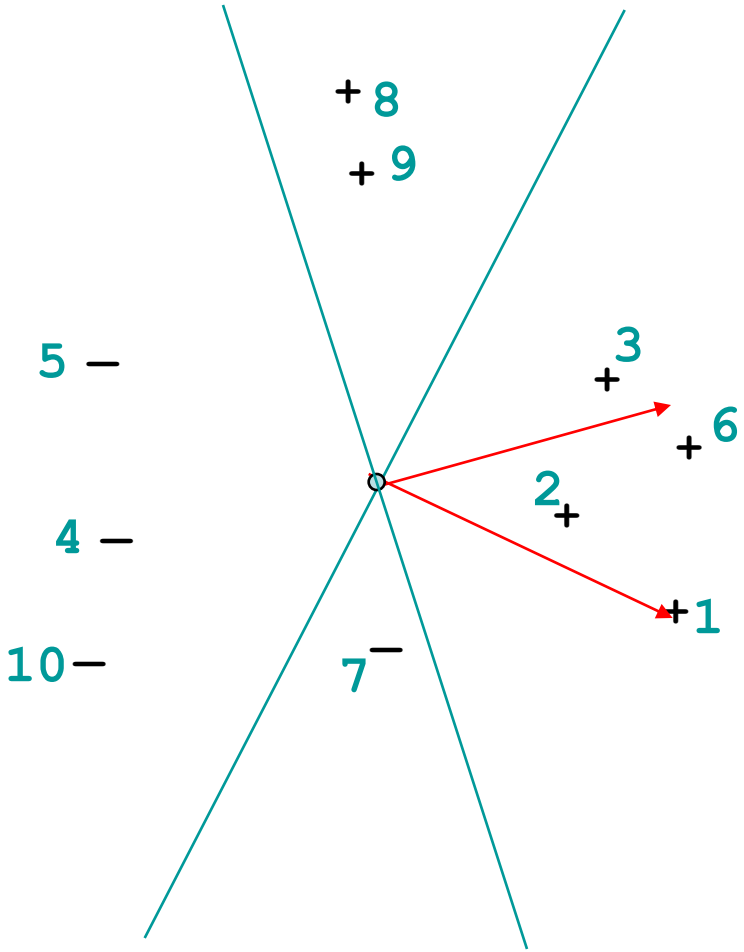
Kernels

Perceptron in action




```
w = 0
while some (x,y) is
  misclassified:
      w = w + yx
```

$$w = x^{(1)} - x^{(7)}$$

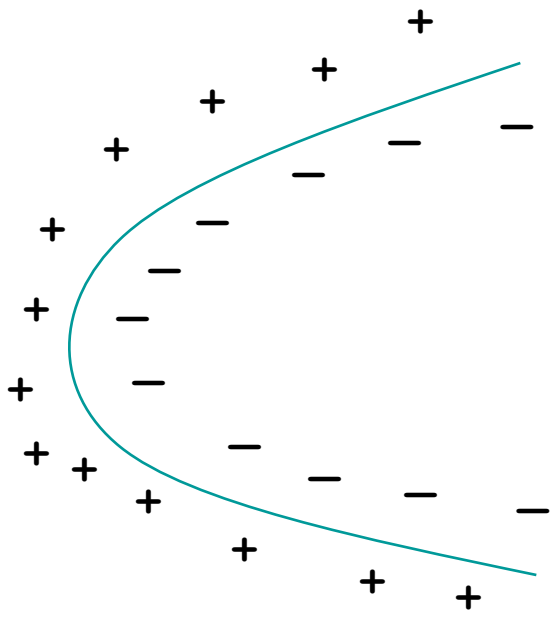
Sparse representation:

[(1,1), (7,-1)]

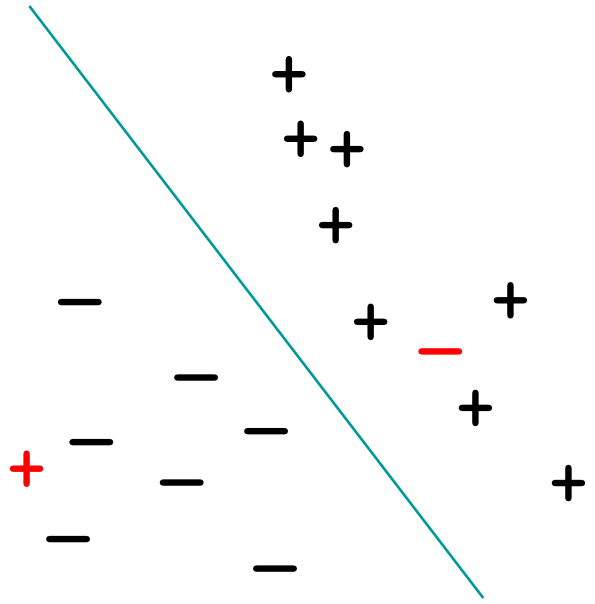
When does this fail?

When data is not *linearly separable*.

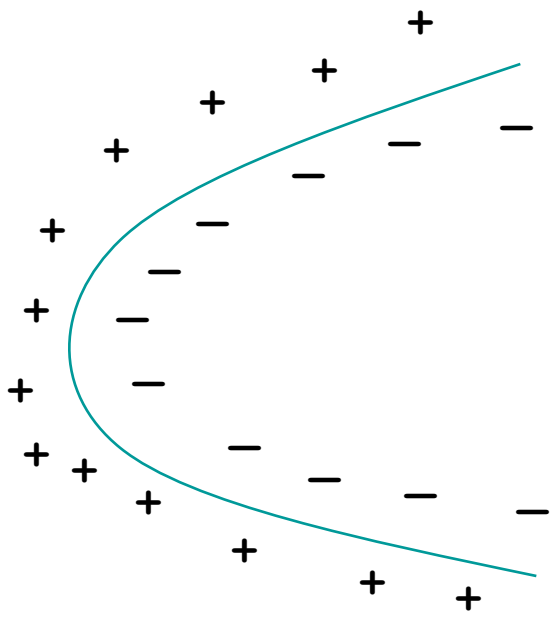
[1] Systematic deviation



[2] Noise



Systematic inseparability



Actual decision boundary is *quadratic*.

Quick fix: in addition to regular features

$$\mathbf{x} = (x_1, x_2, \ldots, x_d),$$

add in extra features

$$x_1^2, x_2^2, \ldots, x_d^2,$$

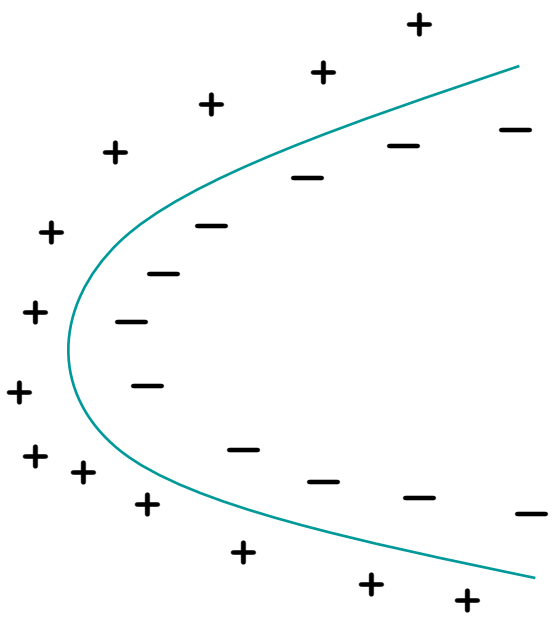
$$x_1x_2, x_1x_3, \ldots, x_{d-1}x_d$$

The new, enhanced data vectors are of the form

$$\Phi(\mathbf{x}) =$$

$$(1, \sqrt{2}x_1, \ldots, \sqrt{2}x_d, x_1^2, \ldots, x_d^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \ldots, \sqrt{2}x_{d-1}x_d)$$

Adding new features



Boundary is something like

$$x_1 = x_2^2 + 5.$$

This is *quadratic* in

$$\mathbf{x} = (1, x_1, x_2)$$

but *linear* in

$$\Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

By embedding the data in a higher-dimensional feature space, we can keep using a linear classifier!

Perceptron revisited

Learning in the higher-dimensional feature space: map each $\mathbf{x}$ onto $\Phi(\mathbf{x})$ and then run the regular perceptron; that is:

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

Everything works as before; final $\mathbf{w}$ is a weighted sum of various $\Phi(\mathbf{x})$.

Problem: number of features has now increased dramatically.

For instance, if $\mathbf{x} \in R^{1,000}$ then $\Phi(\mathbf{x}) \in R^{1,000,000}$!

The kernel trick

[Aizenman, Braverman, Rozonoer, 1964]

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

[1] Keep w in sparse form

If $\mathbf{w}$ = (say) $a_1\ \Phi(\mathbf{x}^{(1)}) + a_2\ \Phi(\mathbf{x}^{(20)}) + a_3\ \Phi(\mathbf{x}^{(31)})$,

store it as a list, $[(1,a_1),\ (20,a_2),\ (31,a_3)]$.

[2] When do we ever access $\Phi(\mathbf{x})$?

Only to compute a dot product $\mathbf{w}\cdot\Phi(\mathbf{x})$ (during training or future evaluation).

In above example $\mathbf{w}\cdot\Phi(\mathbf{x})$ is

$a_1\ \Phi(\mathbf{x}^{(1)})\cdot\Phi(\mathbf{x}) + a_2\ \Phi(\mathbf{x}^{(20)})\cdot\Phi(\mathbf{x}) + a_3\ \Phi(\mathbf{x}^{(31)})\cdot\Phi(\mathbf{x})$

In general, $\mathbf{w}\cdot\Phi(\mathbf{x})$ is a (weighted) sum of dot products $\Phi(\mathbf{x}^{(i)})\cdot\Phi(\mathbf{x})$.

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

Kernel trick, cont'd

In 2-d:

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

In d dimensions:

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

Computing dot products in the 1,000,000-dimensional feature space takes time proportional to just 1000, the original dimension!

Never need to write out $\Phi(x)$!

Kernel trick

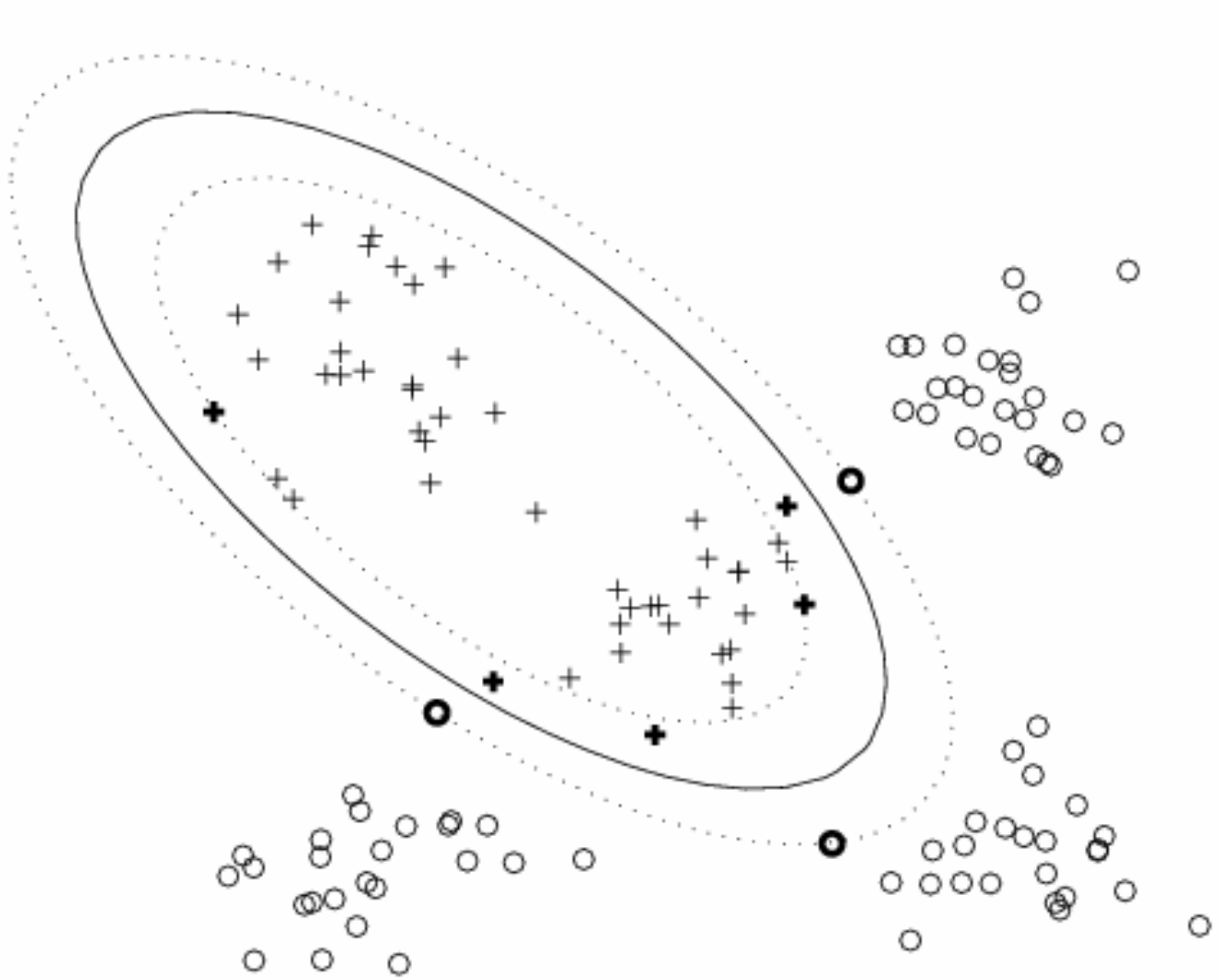
Why does it work?

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

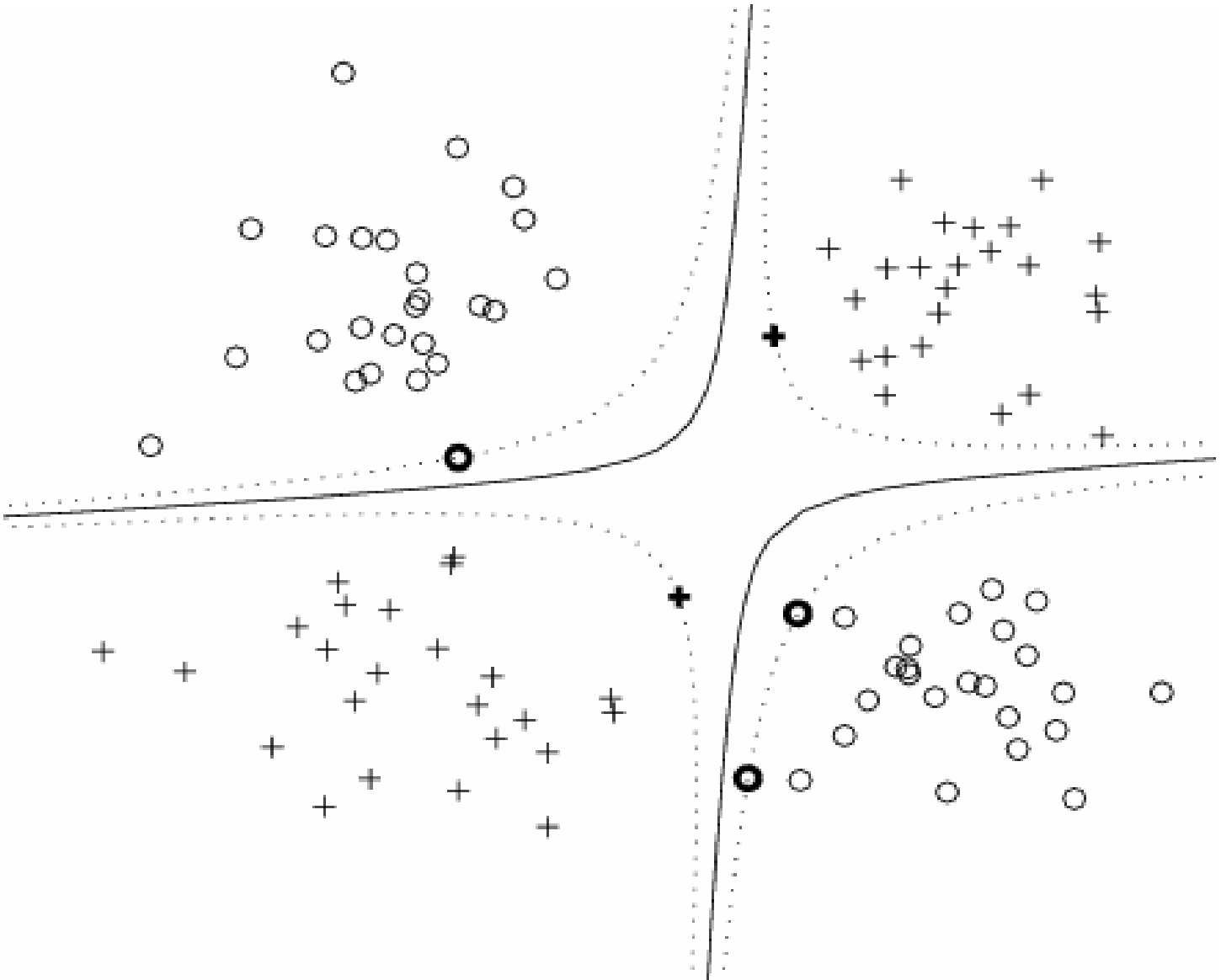
2. And $\mathbf{w}$ itself is a linear combination of $\Phi(\mathbf{x})$'s. If $\mathbf{w} = a_1 \Phi(\mathbf{x}^{(1)}) + a_{22} \Phi(\mathbf{x}^{(22)}) + a_{37} \Phi(\mathbf{x}^{(37)})$ store it as $[(1, a_1), (22, a_{22}), (37, a_{37})]$

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

Quadratic kernel

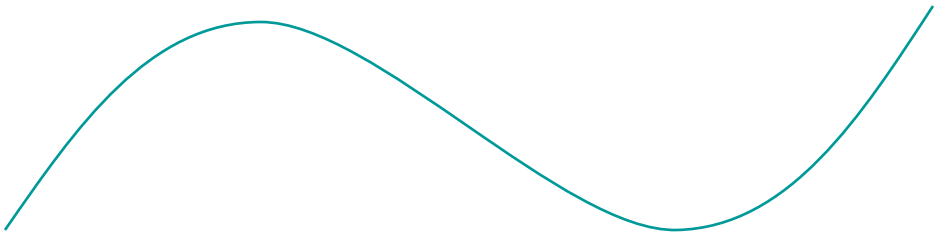


Quadratic kernel



Polynomial decision surfaces

To get a decision surface which is a polynomial of order d:



Let $\Phi(\mathbf{x})$ consist of all terms of order $\leq d$, e.g. $x_1 x_2 x_3^{d-3}$

If $\mathbf{x}$ has p coords, then $\Phi(\mathbf{x})$ has about p^d coords.

Same trick works: $\Phi(\mathbf{x}) \cdot \Phi(\mathbf{z}) = (1 + \mathbf{x} \cdot \mathbf{z})^d$!

The data is accessed only through the *kernel function*

$$k(\mathbf{x},\mathbf{z}) = \Phi(\mathbf{x}) \cdot \Phi(\mathbf{z}) .$$

This is a measure of similarity.

String kernels

Sequence data: eg.

text documents

speech signals

protein sequences

Each data point is a *sequence*.

Input space

`X = {A,C,G,T}*`

`X = {English words}*`

Different data points have different lengths.

String kernels

For each substring **s**, define *feature*

$\Phi_s(x)$ = # of times substring **s** appears in **x**

and

$\Phi(x) = (\Phi_s(x)$: all strings **s**)

(vector with a coordinate for each string **s** in the alphabet)

Infinite-dimensional embedding!

Eg.

Φ_{ar}(aardvark) = 2

Φ_{th}(aardvark) = 0

Linear classifier with such features is potentially very powerful.

String kernels

We can compute dot products quickly!

To compute $k(x,z) = \Phi(x) \cdot \Phi(z)$:

```
for each substring s of x
    count how often s appears in z
```

With dynamic programming, this just takes time proportional to product of the lengths of x, z.

Postscript: kernels over other interesting input spaces: parse trees, unordered collections of objects,…

Kernel function

Now, shift attention:

away from embedding $\Phi(x)$

[which we never explicitly construct anyway],

towards the *similarity measure* $k(x,z)$

[the thing we actually use].

Let's rewrite everything in terms of k.

Suppose the final linear separator is

$$w = a_1 y_1 \Phi(x^{(1)}) + \ldots + a_m y_m \Phi(x^{(m)}).$$

Then the final classifier can also be written

$$F(x) = \mathrm{sgn}\{a_1 y_1 k(x^{(1)},x) + \ldots + a_m y_m k(x^{(m)},x)\},$$

a weighted vote.

Kernel function

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

Any $\mathbf{k}$ satisfying a technical condition ("positive definiteness") corresponds 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*.

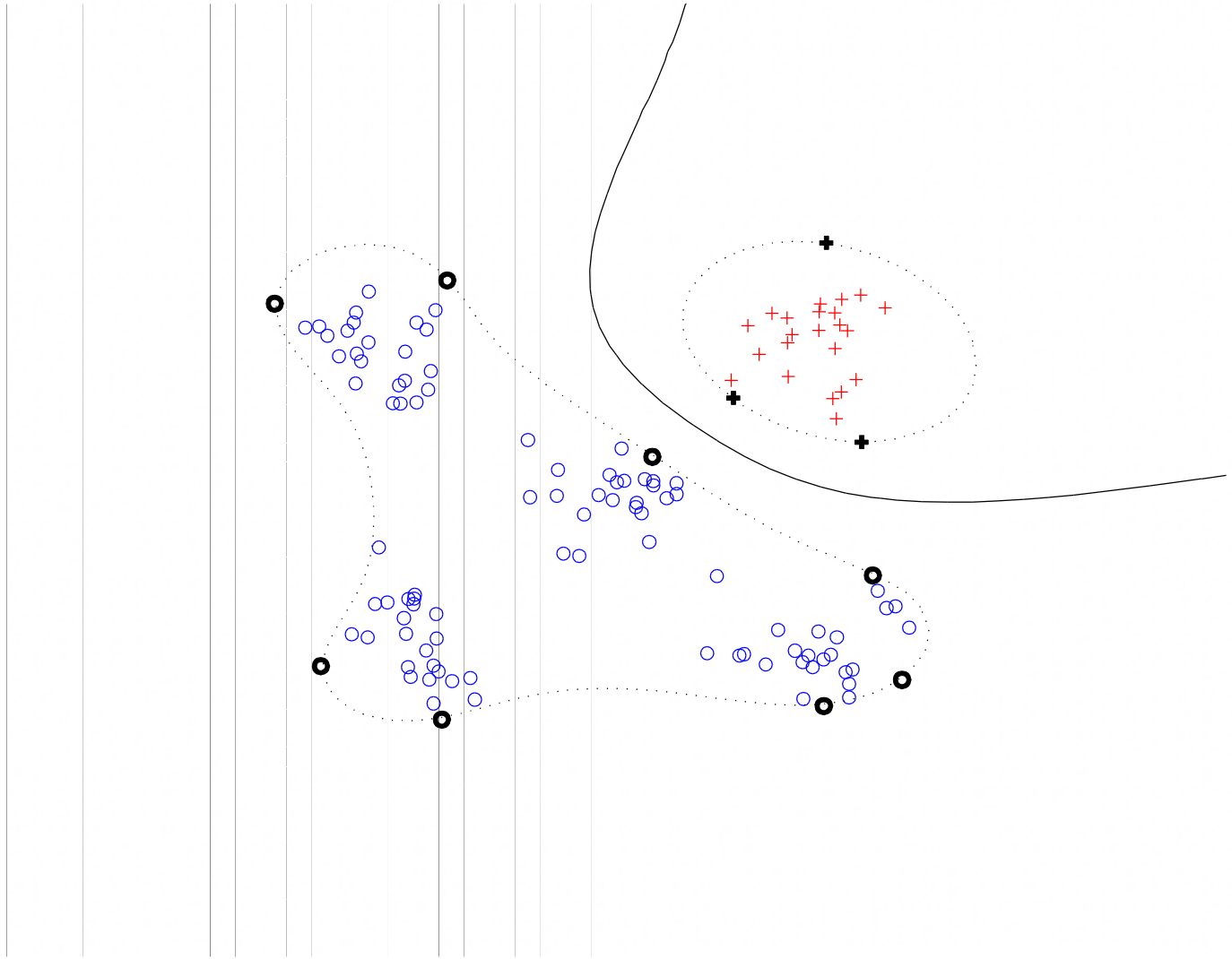
$$\mathbf{k(x,x')} = \exp(-\|\mathbf{x} - \mathbf{x'}\|^2/\sigma^2)$$

The resulting classifiers

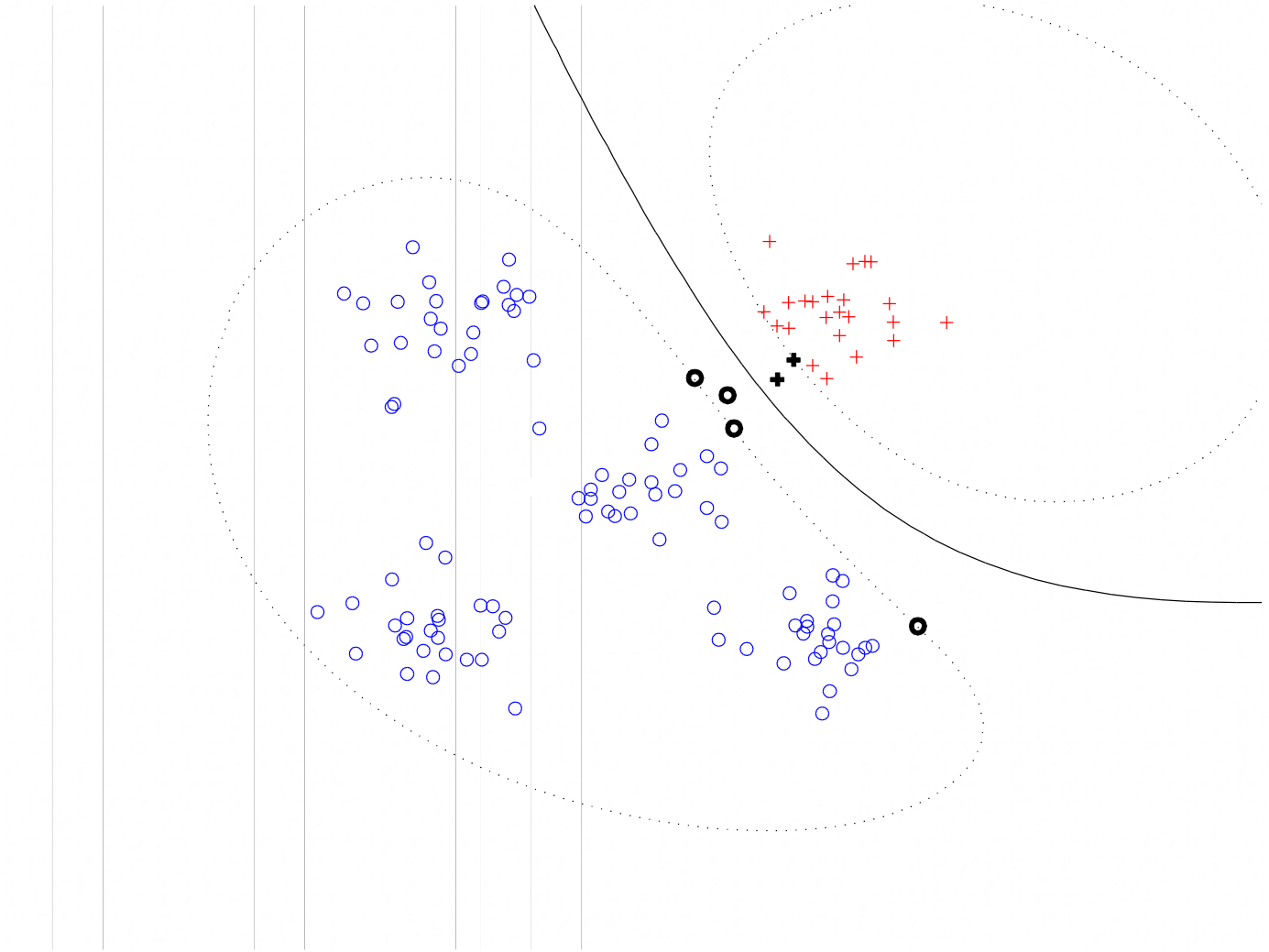
$$\mathbf{F(x)} = \mathrm{sgn}\{a_1 y_1 \mathbf{k}(\mathbf{x}^{(1)},\mathbf{x}) + \ldots + a_m y_m \mathbf{k}(\mathbf{x}^{(m)},\mathbf{x})\}$$

are closely related to nearest neighbors.

Gaussian kernel



Gaussian kernel



Generalization

With a powerful kernel: can fit any data set.

Danger: *overfitting* – the resulting classifier might not perform well on future instances.

e.g. given 100 data points in $\mathbf{R}^{101}$.

Can always find a linear classifier to fit them.

But prediction on a future point is arbitrary.

Margins

All this becomes a lot more manageable if there is a substantial *margin* between the two classes.

Performance of the perceptron depends only on the margin m of the classifier, not the dimension!

[Number of updates = $1/m^2$]

So even infinite dimensions are fine!