

- May 1, **CODY**, Error Backpropagation, Bishop 5.3, and Support Vector Machines (SVM) Bishop Ch 7.
- May 3, **Class HW** Bishop Ch 6-7, RVM, PCA dimensionality reduction
- May 8, **CODY** Machine Learning for finding oil, focusing on 1) robust seismic denoising/interpolation using structured matrix approximation 2) seismic image clustering and classification, using t-SNE(t-distributed stochastic neighbor embedding) and CNN. **Weichang Li**, Goup Leader Aramco, Houston.
- May 10, **Class HW First distribution of final projects**. K-means. Dictionary learning, **Mike Bianco** (half class) Bishop Ch 9
- May 15, **CODY** Seismology and Machine Learning, **Daniel Trugman** (half class), ch 9
- May 17, **Class HW Ocean acoustic source tracking. Final projects. The main goal in the last 3 weeks is the Final project.**
- May 22,
- May 24, Graphical models Bishop Ch 8
- May 31, **No Class Workshop, Big Data and The Earth Sciences: Grand Challenges Workshop**
- June 5, Discuss workshop.
- June 7, **Workshop report.** Importance of feature extraction, **Aaron Thode** (half class)
- June 16 **Final report delivered**

# Solving a Rank-Deficient System

If  $A$  is  $m$ -by- $n$  with  $m > n$  and full rank  $n$ , each of the three statements

$$x = A \backslash b$$

$$x = \text{pinv}(A) * b$$

$$x = \text{inv}(A' * A) * A' * b$$

theoretically computes the same least-squares solution  $x$ , although the backslash operator does it faster.

However, if  $A$  does not have full rank, the solution to the least-squares problem is not unique. There are many vectors  $x$  that minimize  $\text{norm}(A * x - b)$

The solution computed by  $x = A \backslash b$  is a basic solution; it has at most  $r$  nonzero components, where  $r$  is the rank of  $A$ . The solution computed by  $x = \text{pinv}(A) * b$  is the minimal norm solution because it minimizes  $\text{norm}(x)$ . An attempt to compute a solution with  $x = \text{inv}(A' * A) * A' * b$  fails because  $A' * A$  is singular.

# Error back propagation

- To train a NN we need to update weights  $\mathbf{w}$ . The size of  $\mathbf{w}$  is HUGE. Using gradient search or higher order methods.
- We have  $N$  observations  $\{x_n, t_n\}$
- For each observation we can compute  $\frac{dE(y_n, t_n)}{d\mathbf{w}}$
- The gradient is then computed as  $\sum_n^{batch} \frac{dE(y_n, t_n)}{d\mathbf{w}}$

# Error backpropagation

- To update weights we need derivatives for either gradient method or higher order methods.
- We have a set of  $\{\mathbf{x}_n, \mathbf{t}_n\}$ .
- For each set we can compute  $\frac{dE(\mathbf{x}_n, \mathbf{t}_n)}{d\mathbf{w}}$

# Gaussian Kernels

- Gaussian Kernel

$$k(\mathbf{x}, \mathbf{x}') = \exp \left( -\frac{1}{2} (\mathbf{x} - \mathbf{x}')^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \mathbf{x}') \right)$$

Diagonal  $\boldsymbol{\Sigma}$ : (this gives ARD)

$$k(\mathbf{x}, \mathbf{x}') = \exp \left( -\frac{1}{2} \sum_i^N \frac{(x_i - x'_i)^2}{\sigma_i^2} \right)$$

Isotropic  $\sigma_i^2$  gives an RBF

$$k(\mathbf{x}, \mathbf{x}') = \exp \left( -\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\sigma^2} \right)$$

# Commonly used kernels

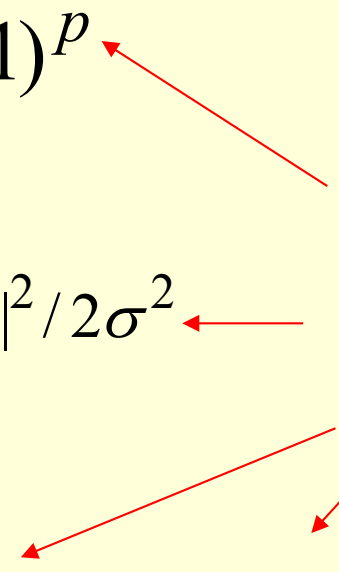
Polynomial:  $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^p$

Gaussian  
radial basis  
function

$$K(\mathbf{x}, \mathbf{y}) = e^{-\|\mathbf{x} - \mathbf{y}\|^2 / 2\sigma^2}$$

Neural net:  $K(\mathbf{x}, \mathbf{y}) = \tanh(k \mathbf{x} \cdot \mathbf{y} - \delta)$

Parameters  
that the user  
must choose



For the neural network kernel, there is one “hidden unit” per support vector, so the process of fitting the maximum margin hyperplane decides how many hidden units to use. Also, it may violate Mercer’s condition.

## Dual representation, Sec 6.2

$$E = \frac{1}{2} \sum_n^N \{ \mathbf{w}^T \mathbf{x}_n - t_n \}^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Solution

$$\begin{aligned} \mathbf{w} &= \mathbf{X}^+ \mathbf{t} \\ &= (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I}_M)^{-1} \mathbf{X}^T \mathbf{t} \\ &= \mathbf{X}^T (\mathbf{X} \mathbf{X}^T + \lambda \mathbf{I}_N)^{-1} \mathbf{t} \\ &= \mathbf{X}^T (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \mathbf{t} \\ &= \mathbf{X}^T \mathbf{a} \end{aligned}$$

Prediction

$$y = \mathbf{w}^T \mathbf{x} = \mathbf{a}^T \mathbf{X} \mathbf{x} = \sum_n^N a_n \mathbf{x}_n^T \mathbf{x} = \sum_n^N a_n k(\mathbf{x}_n, \mathbf{x})$$

Only kernels, no feature vectors

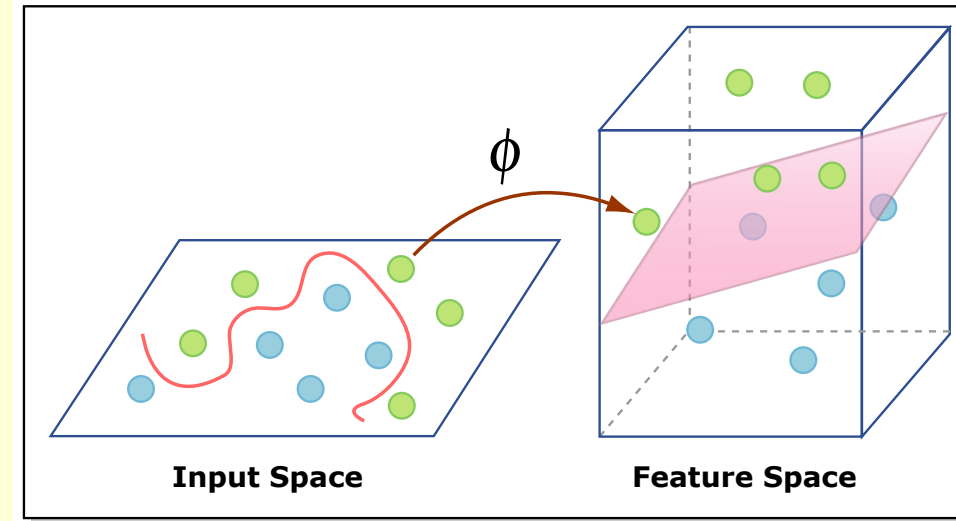
# Kernels

We might want to consider something more complicated than a linear model:

**Example 1:**  $[x^{(1)}, x^{(2)}] \rightarrow \Phi([x^{(1)}, x^{(2)}]) = [x^{(1)2}, x^{(2)2}, x^{(1)}x^{(2)}]$

Information unchanged, but now we have a **linear** classifier on the transformed points.

With the kernel trick, we just need kernel  
 $k(\mathbf{a}, \mathbf{b}) = \Phi(\mathbf{a})^T \Phi(\mathbf{b})$





### Example 4:

$$\begin{aligned}k(\mathbf{x}, \mathbf{z}) &= (\mathbf{x}^T \mathbf{z} + c)^2 = \left( \sum_{j=1}^n x^{(j)} z^{(j)} + c \right) \left( \sum_{\ell=1}^n x^{(\ell)} z^{(\ell)} + c \right) \\&= \sum_{j=1}^n \sum_{\ell=1}^n x^{(j)} x^{(\ell)} z^{(j)} z^{(\ell)} + 2c \sum_{j=1}^n x^{(j)} z^{(j)} + c^2 \\&= \sum_{j,\ell=1}^n (x^{(j)} x^{(\ell)}) (z^{(j)} z^{(\ell)}) + \sum_{j=1}^n (\sqrt{2c} x^{(j)}) (\sqrt{2c} z^{(j)}) + c^2,\end{aligned}$$

and in  $n = 3$  dimensions, one possible feature map is:

$$\Phi(\mathbf{x}) = [x^{(1)2}, x^{(1)}x^{(2)}, \dots, x^{(3)2}, \sqrt{2c}x^{(1)}, \sqrt{2c}x^{(2)}, \sqrt{2c}x^{(3)}, c]$$

and  $c$  controls the relative weight of the linear and quadratic terms in the inner product.

Even more generally, if you wanted to, you could choose the kernel to be any higher power of the regular inner product.

# Lecture 9

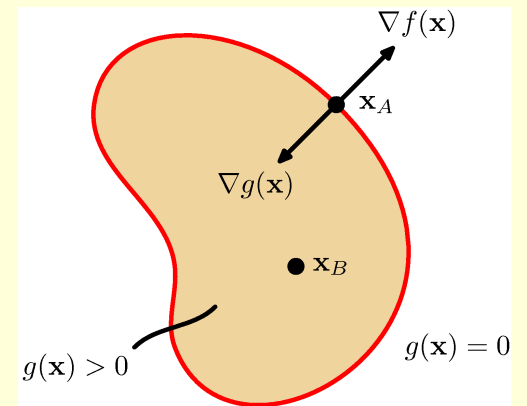
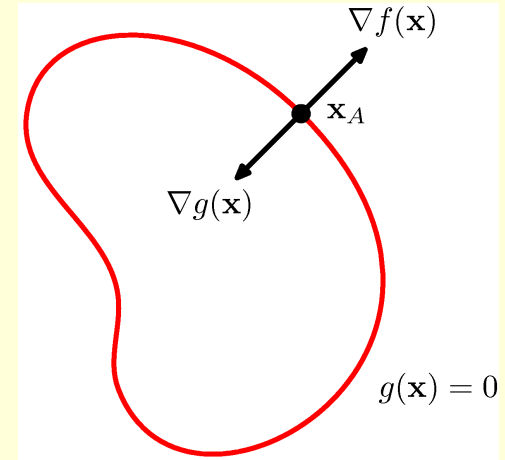
## Support Vector Machines

Non Bayesian!

# Lagrange multiplier

$$\max(f(x)) \text{ subject to } g(x) = 0$$

$$L(x, \lambda) = f(x) + \lambda g(x)$$



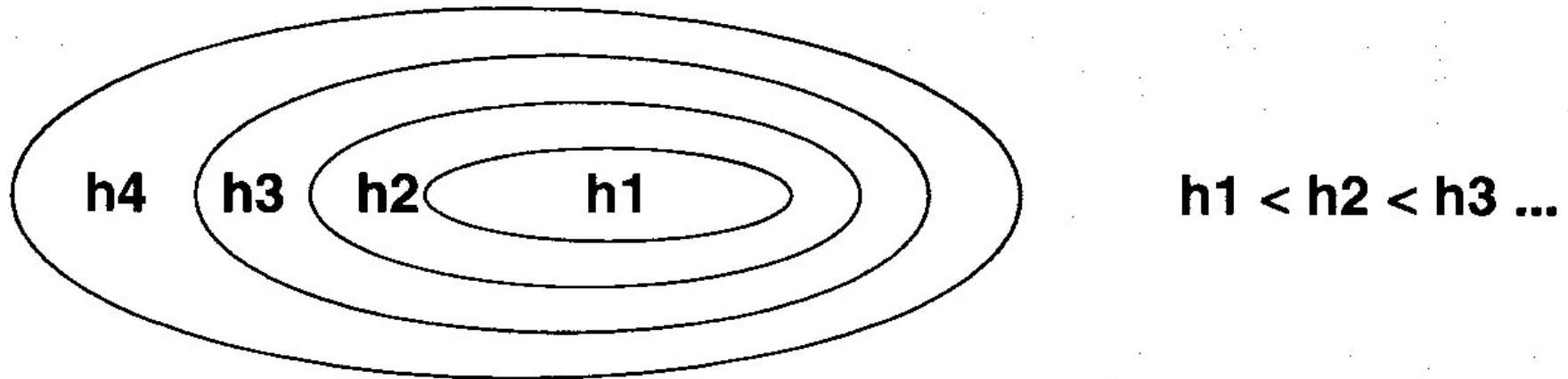
## Preprocessing the input vectors

- Instead predicting the answer directly from the raw inputs we could start by extracting a layer of “features”.
  - Sensible if certain combinations of input values would be useful (e.g. edges or corners in an image).
- Instead of learning the features we could design them by hand.
  - The hand-coded features are equivalent to a layer of non-linear neurons with no need to be learned.
  - Using a big set of features for a two-class problem, the classes will almost certainly be linearly separable.
    - But surely the linear separator gives poor generalization.

## Is preprocessing cheating?

- Its cheating if using carefully designed set of task-specific, hand-coded features and claim that the learning algorithm solved the whole problem.
  - The really hard bit is designing the features.
- Its not cheating if we **learn** the non-linear preprocessing.
  - This makes learning more difficult and more interesting (e.g. backpropagation after pre-training)
- Its not cheating if we use a very big set of non-linear features that is task-independent.
  - Support Vector Machines do this.
  - They prevent overfitting (first half of lecture)
  - They use a huge number of features without requiring as much computation as seems to be necessary (second half).

# A hierarchy of model classes



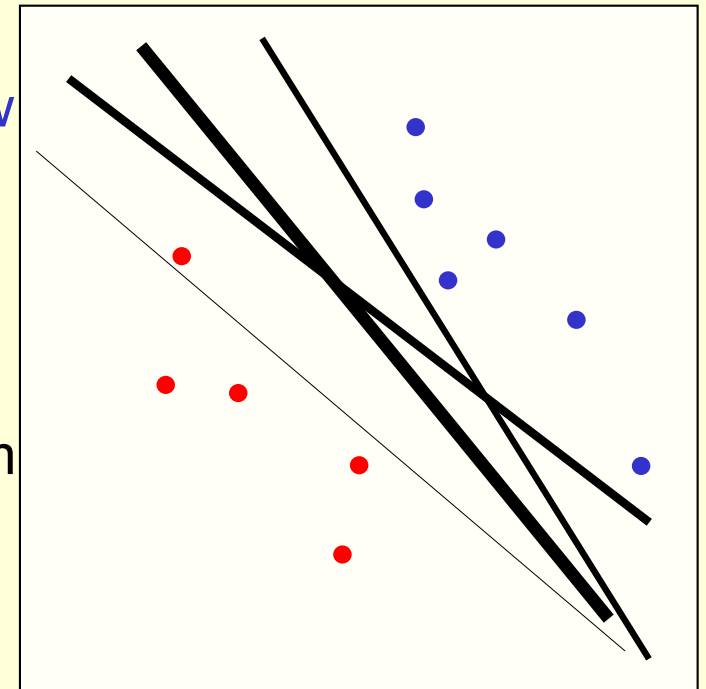
- Some model classes can be arranged in a hierarchy of increasing complexity.
- How to pick the best level in the hierarchy for modeling a given dataset?

## A way to choose a model class

- A low error rate on unseen data.
  - This is called “structural risk minimization”
- A guarantee of the following form is helpful:  
Test error rate  $\leq$  train error rate +  $f(N, h, p)$   
Where  $N$  = size of training set,  
     $h$  = measure of the model complexity,  
     $p$  = the probability that this bound fails  
    We need  $p$  to allow for really unlucky test sets.
- Then we choose the model complexity that minimizes the bound on the test error rate.

# Preventing overfitting when using big sets of features

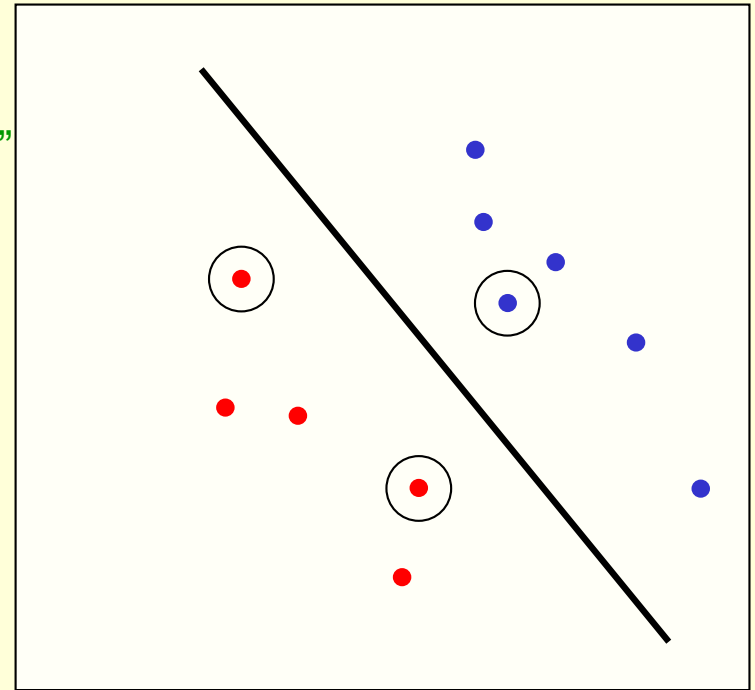
- Suppose we use a big set of features to ensure that two classes are linearly separable. What is the best separating line?
- The Bayesian answer is to use them all (including ones that do not separate the data.)
- Weight each line by its posterior probability (how well it fits the data and how well it fits the prior).
- Is there an efficient way to approximate the correct Bayesian answer?
- **A Bayesian Interpretation:** Using the maximum margin separator often gives a pretty good approximation to using all separators weighted by their posterior probabilities.





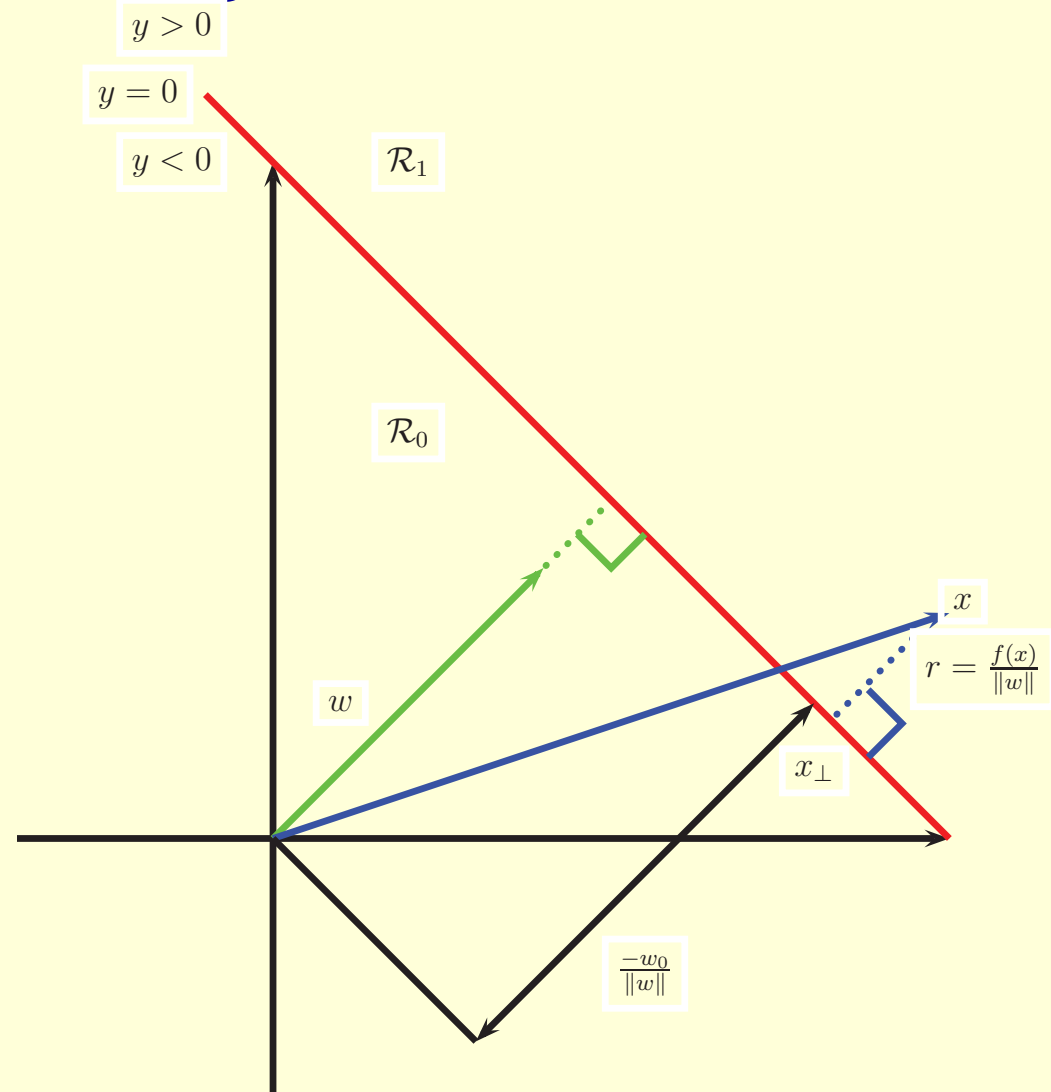
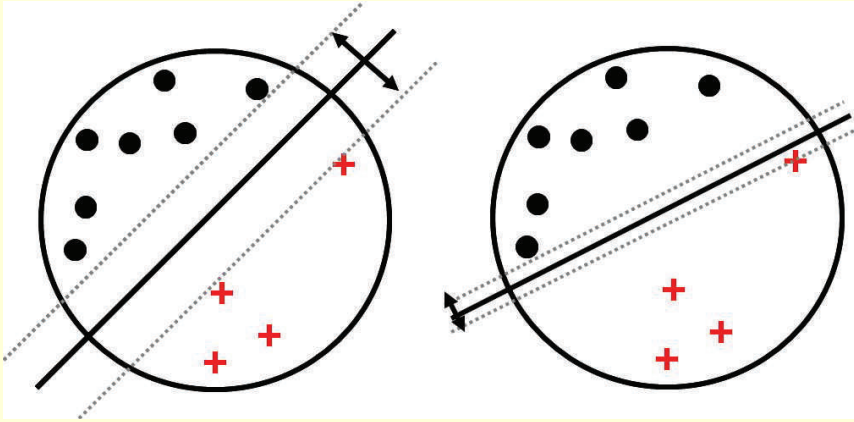
# Support Vector Machines

- The line that maximizes the minimum margin is a good bet.
  - The model class of “hyper-planes with a margin  $m$ ” has a low VC dimension if  $m$  is big.
- This maximum-margin separator is determined by a subset of the datapoints.
  - Datapoints in this subset are called “support vectors”.
  - It is useful computationally if only few datapoints are support vectors, because the support vectors decide which side of the separator a test case is on.



The support vectors are indicated by the circles around them.

# Large margin



## Training a linear SVM

- To find the maximum margin separator, we have to solve the following optimization problem:

$$\mathbf{w} \cdot \mathbf{x}^c + b > +1 \quad \text{for positive cases}$$

$$\mathbf{w} \cdot \mathbf{x}^c + b < -1 \quad \text{for negative cases}$$

$$\text{and } \|\mathbf{w}\|^2 \text{ is as small as possible}$$

- This is tricky but it's a convex problem. There is only one optimum and we can find it without fiddling with learning rates or weight decay or early stopping.
  - Don't worry about the optimization problem. It has been solved. Its called quadratic programming.
  - It takes time proportional to  $N^2$  which is bad for big datasets
    - so for big datasets we end up doing approximate optimization!

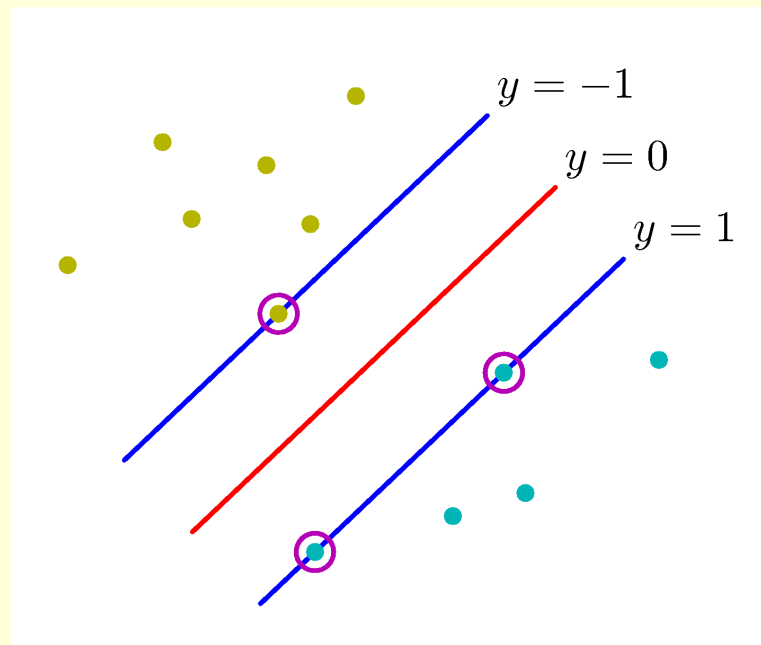
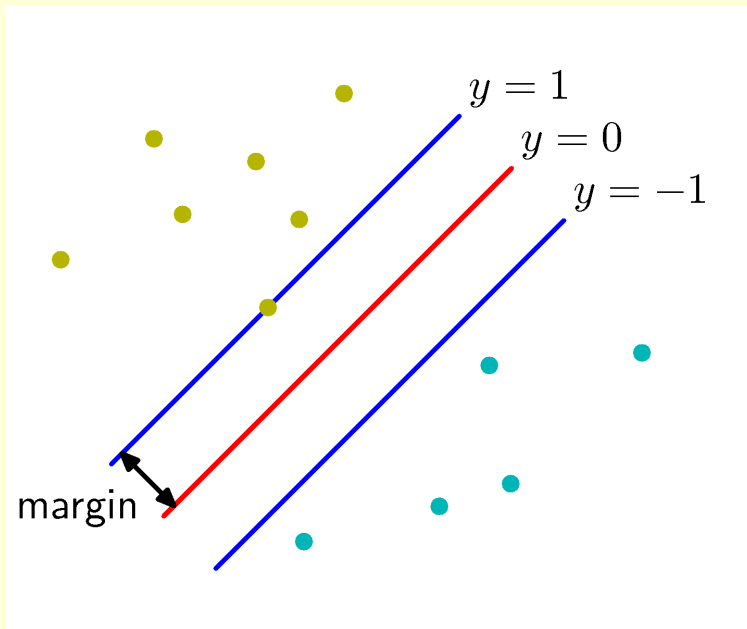
## Testing a linear SVM

- The separator is defined as the set of points for which:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

*so if  $\mathbf{w} \cdot \mathbf{x}^c + b > 0$  say its a positive case*

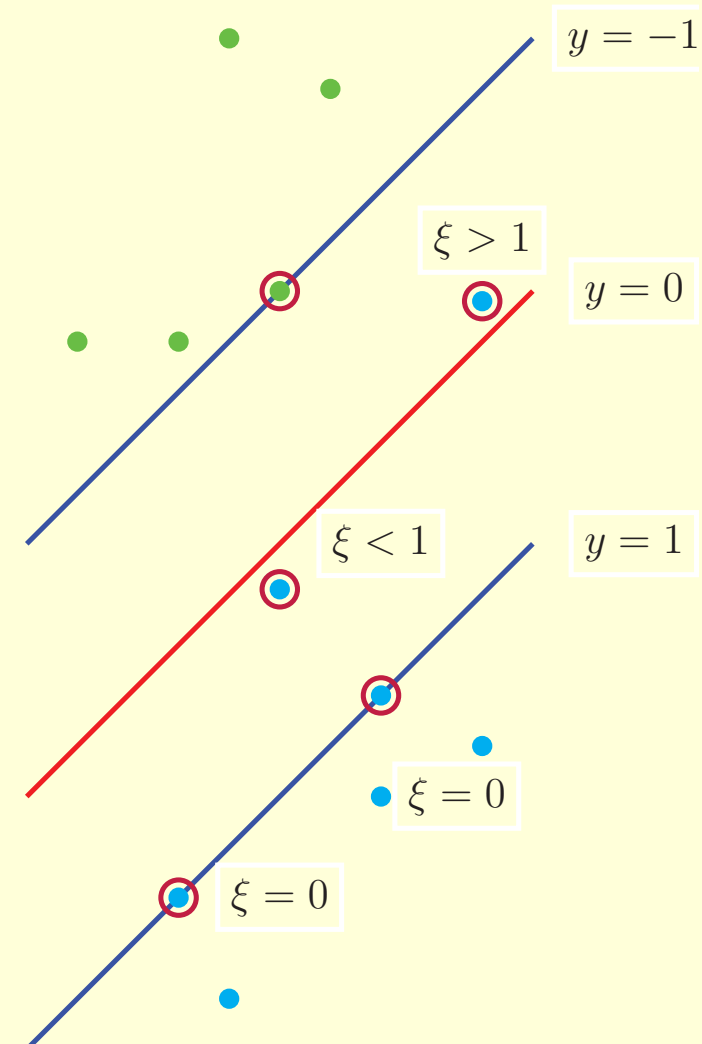
*and if  $\mathbf{w} \cdot \mathbf{x}^c + b < 0$  say its a negative case*

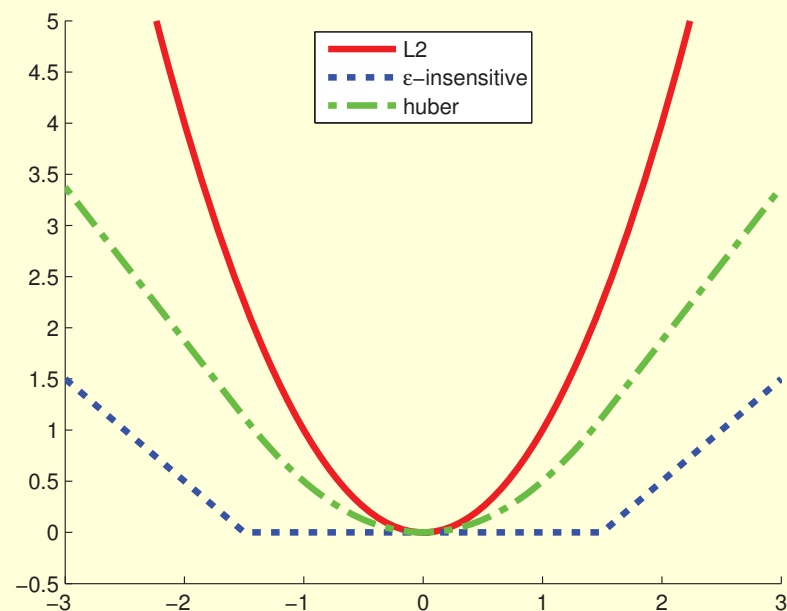


# What to do if there is no separating plane

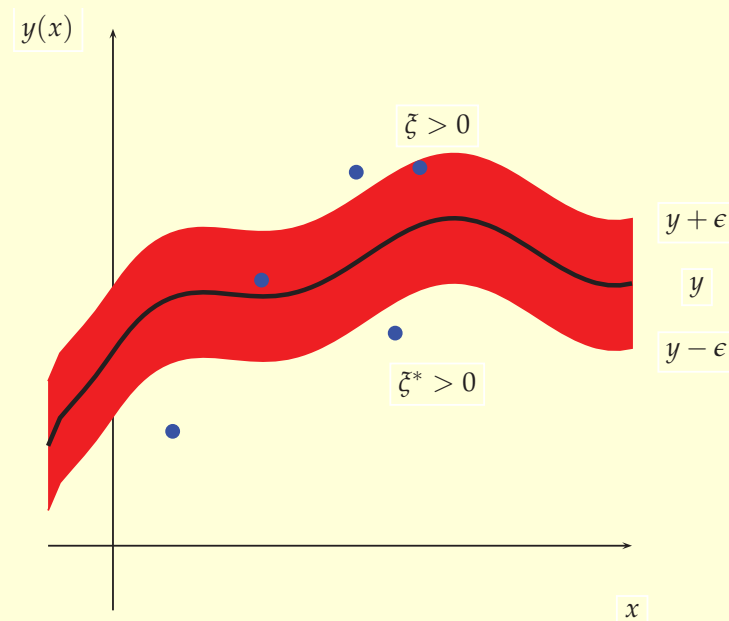
- Use a bigger set of features.
  - Makes the computation slow, but the “kernel” trick makes the computation fast even with many features.
- Extend definition of maximum margin to allow non-separating planes.
  - Can be done by using “slack” variables

$$\xi = |t_n - y(x_n)|$$





(a)



(b)

**Figure 14.10** (a) Illustration of  $\ell_2$ , Huber and  $\epsilon$ -insensitive loss functions, where  $\epsilon = 1.5$ . Figure generated by `huberLossDemo`. (b) Illustration of the  $\epsilon$ -tube used in SVM regression. Points above the tube have  $\xi_i > 0$  and  $\xi_i^* = 0$ . Points below the tube have  $\xi_i = 0$  and  $\xi_i^* > 0$ . Points inside the tube have  $\xi_i = \xi_i^* = 0$ . Based on Figure 7.7 of (Bishop 2006a).

## Introducing slack variables

- Slack variables are non-negative. When greater than zero they allow us to cheat by putting the plane closer to the datapoint than the margin. We need to minimize the amount of cheating. This means we have to pick a value for lambda (this sounds familiar!)

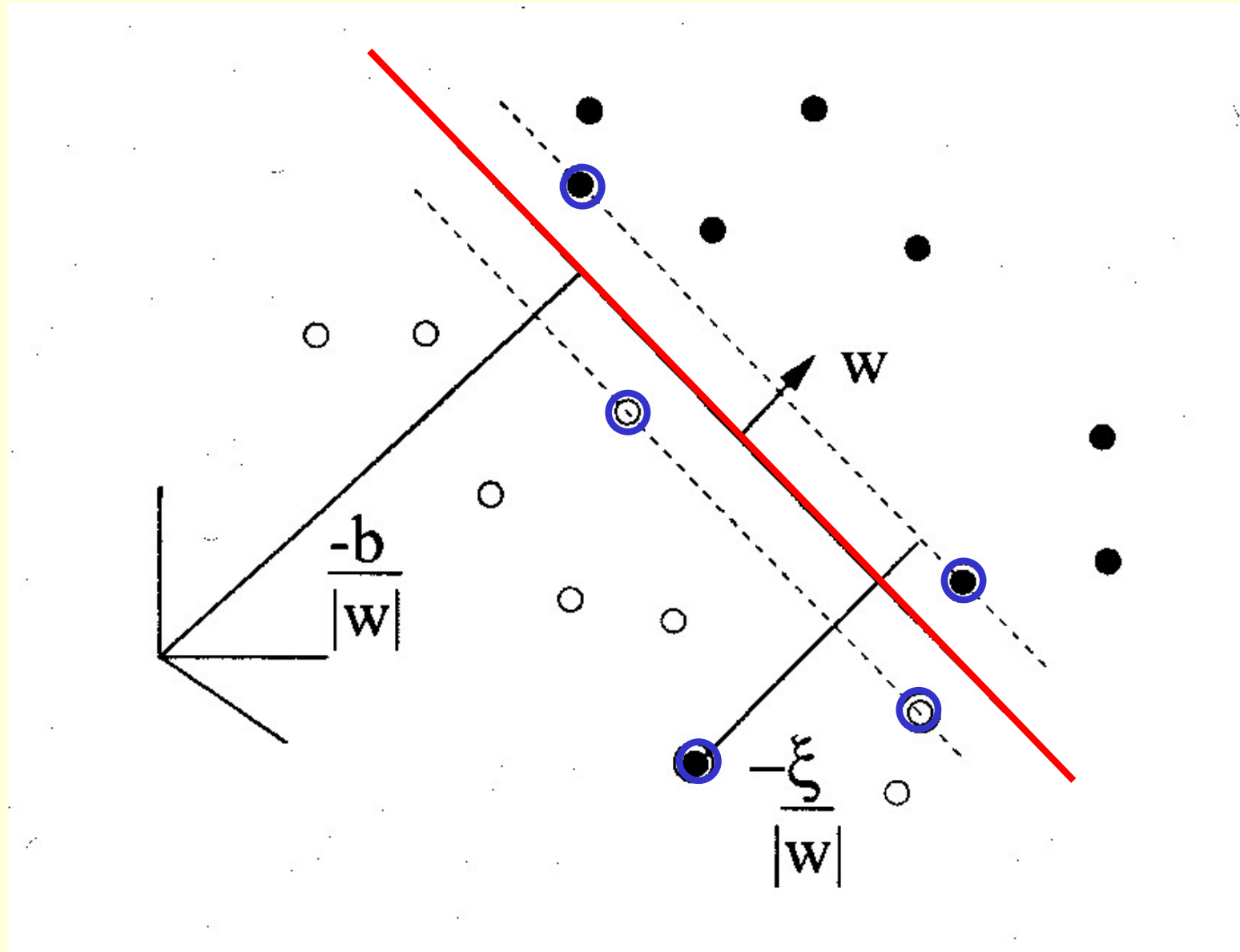
$$\mathbf{w} \cdot \mathbf{x}^c + b \geq +1 - \xi^c \quad \text{for positive cases}$$

$$\mathbf{w} \cdot \mathbf{x}^c + b \leq -1 + \xi^c \quad \text{for negative cases}$$

$$\text{with } \xi^c \geq 0 \quad \text{for all } c$$

$$\text{and } \frac{\|\mathbf{w}\|^2}{2} + \lambda \sum_c \xi^c \quad \text{as small as possible}$$

# A picture of the best plane with a slack variable



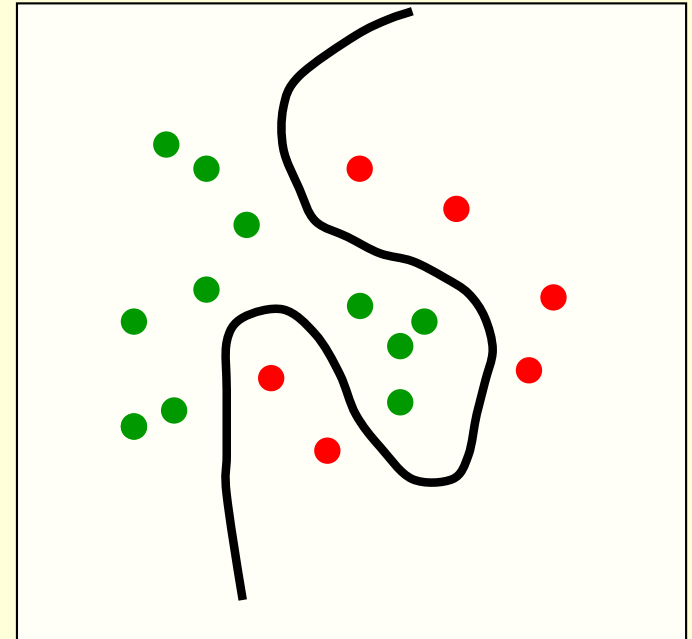


## The story so far

- Using a large set of non-adaptive features, we might make the two classes linearly separable.
  - But just fitting any separating plane, it will not generalize well to new cases.
- Fitting the separating plane maximizing the margin (minimum distance to any data points), gives better generalization.
  - Intuitively, maximizing the margin squeezes the surplus capacity that came from using a high-dimensional feature space.
- This is justified by a lot of clever mathematics which shows that
  - large margin separators have lower VC dimension.
  - models with lower VC dimension have a smaller gap between training and test error rates.

## How to make a plane curved

- Fitting hyperplanes as separators is mathematically easy.
  - The mathematics is **linear**.
- Replacing the raw input variables with a much larger set of features we get a nice property:
  - A planar separator in high-D feature space is a curved separator in the low-D input space.



A planar separator in a 20-D feature space projected back to the original 2-D space

## A potential problem and a magic solution

- Mapping input vectors into a **very** high-D feature space, surely finding the maximum-margin separator is computationally intractable?
  - The mathematics is all **linear**, but the vectors have a huge number of components.
  - Taking the scalar product of two vectors is expensive.
- The way to keep things tractable is **“the kernel trick”**
- The kernel trick makes your brain hurt when you first learn about it, but it is actually simple.

## What the kernel trick achieves

- All computations to find the maximum-margin separator is expressed as scalar products between pairs of datapoints (in high-D feature space).
- These scalar products are the only part of the computation that depends on the dimensionality of the high-D space.
  - We need a fast way to do the scalar products to solve the learning problem in the high-D space.
- The **kernel trick** is a magic way of doing scalar products a lot faster.
  - It relies on choosing a way of mapping to the high-D feature space that allows fast scalar products.

## Dealing with the test data

- Choosing a high-D mapping for which the kernel trick works, we do not use much CPU time for the high-D when finding the best hyper-plane.
  - We cannot express the hyperplane by using its normal vector in the high-D space because this vector is huge.
  - Luckily, we express it in terms of the support vectors.
- What about the test data. We cannot compute the scalar product  $\mathbf{w} \cdot \phi(\mathbf{x})$  because its in the high-D space.
- Deciding which side of the separating hyperplane a test point lies on, requires a scalar product  $\cdot \mathbf{w} \cdot \phi(\mathbf{x})$
- We express this scalar product as a weighted average of scalar products using stored support vectors
  - Could be slow many support vectors.

## The classification rule

- The classification rule is simple:

$$bias + \sum_{s \in SV} w_s K(x^{test}, x^s) > 0$$

↑ The set of  
support vectors

- The cleverness is in selecting the support vectors maximizing the margin and computing the weight for each support vector.
- Need choosing a good kernel function and maybe choosing a lambda for non-separable cases.

# Performance

- SVM work very well in practice.
  - The user must choose the kernel function and its parameters, but the rest is automatic.
  - The test performance is very good.
- They can be expensive in time and space for big datasets
  - The computation of the maximum-margin hyper-plane depends on the **square** of number of training cases.
  - Need storing all the support vectors.
- SVM's are good if you have no idea about what structure to impose.
- The kernel trick can also be used for PCA in a high-D space, thus giving a non-linear PCA in the original space.

# SVMs are Perceptrons!

- SVM's use each training case,  $x$ , to define a feature  $K(x, \cdot)$  where  $K$  is user chosen.
  - So the user designs the features.
- SVM do “feature selection” by picking support vectors, and learn feature weighting from a big optimization problem.
- So an SVM is a clever way to train a standard perceptron.
  - All that a perceptron cannot do, cannot be done by SVM's (but it's a long time since 1969 so people have forgotten this).
- SVM DOES:
  - Margin maximization
  - Kernel trick
  - Sparse



- **NOT USED**