MODULE 6

Probabilistic Learning-Gaussian Mixture Models-Nearest Neighbor Models-Support Vector Machines-Optimal Separation-Kernels-The Support Vector Machine Algorithm-Extensions to the SVM

Guassian Mixture Model

- A **statistical law** explains a phenomenon in terms of the probability of occurrence of its underlying relationships
- multi-modal data:
- A ***k*-component mixture model** is a weighted sum of laws, the likelihood of a sample x being given by

with $$f(x) = \sum_{i=1}^{K} \pi_i f_i(x)$$

- If the laws f_i are probability distributions f is also a probability distribution (a **mixture of probability distributions**)

$$\sum_{i=1}^{K} \pi_i = 1$$

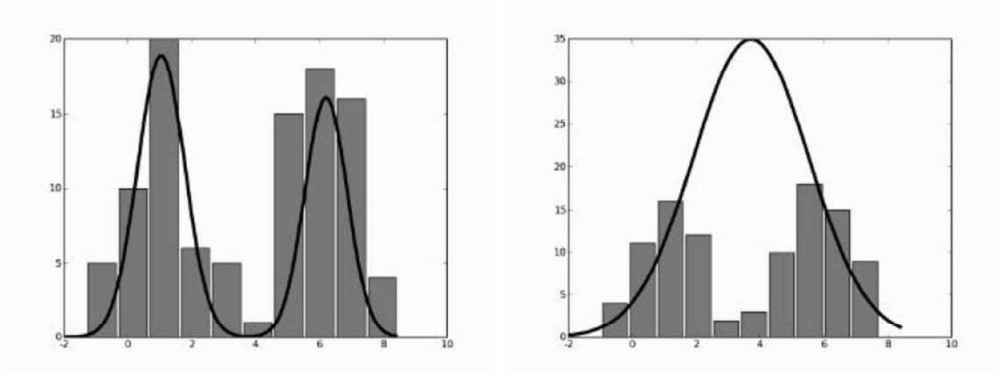
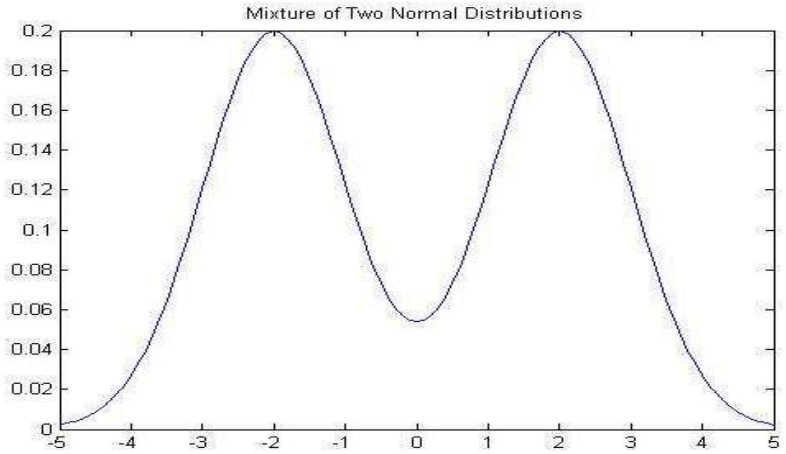


FIGURE 7.1 Histograms of training data from a mixture of two Gaussians and two fitted models, shown as the line plot. The model shown on the left fits well, but the one on the right produces two Gaussians right on top of each other that do not fit the data well.

Expectation Maximization

- The **Expectation-Maximization (EM) Algorithm** is a statistical method for finding maximum likelihood estimates of model parameters based on observations.
- In this case:
  - Assume we know that a joint distribution can be expressed as a weighted sum of two distributions (mixture distribution)
  - EM algorithm can estimate the parameters of these distributions



$$f(\pi_1, \pi_2, \mu_1, \mu_2, \sigma_1, \sigma_2) = \pi_1 N(\mu_1, \sigma_1) + \pi_2 N(\mu_2, \sigma_2)$$

Expectation –Maximization Algorithm

Idea of the EM algorithm

sometimes it is easier to add extra variables that are not actually known (called hidden or latent variables) and then

To maximize the function over those variables.

making a problem much more complicated to be, but it turns out for many problems that it makes finding the solution significantly easier.

Gaussian distribution

- Gaussian or normal distribution, 1D

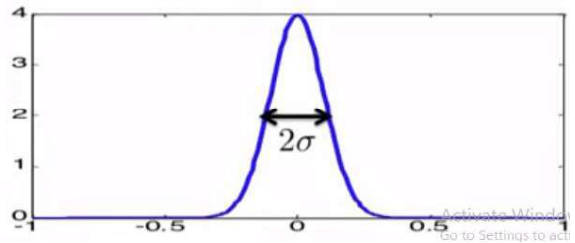
$$\mathcal{N}(x\ ;\ \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}(x-\mu)^2/\sigma^2\right]$$

  - Parameters: mean μ, variance σ^2 (standard deviation σ)

Maximum Likelihood estimates

$$\hat{\mu} = \frac{1}{N}\sum_i x^{(i)}$$

$$\hat{\sigma}^2 = \frac{1}{N}\sum_i (x^{(i)} - \hat{\mu})^2$$



Expectation –Maximization Algorithm

Working Principle

- Consider a combination of just two Gaussian mixtures.
- Data were created by randomly choosing one of two possible Gaussians, and then creating a sample from that Gaussian.
- If the probability of picking Gaussian one is *p*, then the entire model looks like this

$$G_1 = \mathcal{N}(\mu_1, \sigma_1^2)$$
$$G_2 = \mathcal{N}(\mu_2, \sigma_2^2)$$
$$y = pG_1 + (1-p)G_2.$$

(where $N(\mu,\sigma^2)$- Gaussian distribution with mean μ and standard deviation σ

If the probability distribution of p is written as π, then the probability density is:

$$P(\mathrm{y}) = \pi\phi(\mathrm{y};\mu_1,\sigma_1) + (1-\pi)\phi(\mathrm{y};\mu_2,\sigma_2).$$

Expectation –Maximization Algorithm

- Finding the maximum likelihood solution to this problem
  - computing the sum of the logarithm of Equation over all the training data, and differentiating it, (which would be rather difficult)
- The mean and standard deviation for each component could be computed from the datapoints that belong to that component,
- Variable f- To find which component each datapoint came from

$$\begin{aligned}\gamma_i(\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}) &= E(f|\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}, D) \\ &= P(f = 1|\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}, D)\end{aligned}$$

  - If *f* = 0 then the data came from Gaussian one, if *f* = 1 then it came from Gaussian two.
- compute its expectation (that is, the value that we 'expect' to see, which is the mean average) from the data

where *D*-> data. As *f* = 1 datapoint is choosen from Gaussian two

Expectation Maximization

Description

- Given: A vector of observed data, $x = (x_1, x_2, \ldots, x_n)$ and a number of Gaussian probability distributions.
- Goal: Discover the values of the parameters $\theta = (\mu, \sigma)$ for the Gaussian Probability distributions.

- Algorithm:
  - Pick $\theta_0 = (\mu_0, \sigma_0)$
  - For t = 0, 1, 2, 3…
    - Expectation step (E-step): $P(j \mid x_i, \theta_t)$
    - Maximization step (M-step): Calculate θ_{t+1}
    - Repeat until convergence or max number of iterations has been reached.

Expectation –Maximization Algorithm

- Pick a value for $\theta_0 = (\mu_0, \sigma_0)$ for each probability distribution (there are 2)
  - $\mu_1 = -2$, $\mu_2 = 3$, and $\sigma = 2$ for both of the probability distributions
  - Distribution 1: $\theta_{D1} = (\mu_1, \sigma) = (-2, 2)$
  - Distribution 2: $\theta_{D2} = (\mu_2, \sigma) = (3, 2)$
  - Probability for each distribution = 0.5
- $|X| = 20{,}000$; $X = \{0, \ldots\}$
- Expectation step
  - $P(Dj \mid x_i, \theta_t) = \frac{(0.5)*(P(x_i \mid \theta_{Dj}))}{(0.5)*P(x_i \mid \theta_{D_1})+(0.5)*P(x_i \mid \theta_{D_2})}$; $P(xi \mid \theta) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
  - $P(D1 \mid 0, \theta_t) = \frac{(0.5)*(P(0 \mid \theta_{D_1}))}{(0.5)*P(0_i \mid \theta_{D_1}) + (0.5)*P(0 \mid \theta_{D_2})} = \frac{0.12}{0.12+0.06} = 0.66$
  - $P(D2 \mid 0, \theta_t) = \frac{(0.5)*(P(0 \mid \theta_{D_2}))}{(0.5)*P(0 \mid \theta_{D_1}) + (0.5)*P(0 \mid \theta_{D_2})} = \frac{0.06}{0.12+0.06} = 0.33$

Expectation –Maximization Algorithm

Example part 2 – Maximization step

- $\mu_1 = \sum_{i=1}^{20,000} (x_i * \frac{P(D1 | x_i, \theta_t)}{\sum_{i=1}^{20,000} P(D1 | x_i, \theta t)})$
- $\mu_2 = \sum_{i=1}^{20,000} (x_i * \frac{P(D2 | x_i, \theta t)}{\sum_{i=1}^{20,000} P(D2 | x_i, \theta t)})$

Iteration	μ_1	μ_2
0	-2.00	3.00
1	-3.74	4.10
2	-3.94	4.07
3	-3.97	4.04
4	-3.98	4.03
5	-3.98	4.03

Expectation –Maximization Algorithm

The Gaussian Mixture Model EM Algorithm

- Initialisation
  - set $\hat{\mu}_1$ and $\hat{\mu}_2$ to be randomly chosen values from the dataset
  - set $\hat{\sigma}_1 = \hat{\sigma}_2 = \sum_{i=1}^{N}(y_i - \bar{y})^2/N$ (where $\bar{y}$ is the mean of the entire dataset)
  - set $\hat{\pi} = 0.5$
- Repeat until convergence:
  - (E-step) $\hat{\gamma}_i = \frac{\hat{\pi}\phi(y_i;\hat{\mu}_1,\hat{\sigma}_1)}{\hat{\pi}\phi(y_i;\hat{\mu}_1,\hat{\sigma}_1)+(1-\hat{\pi})\phi(y_i;\hat{\mu}_2,\hat{\sigma}_2)}$ for $i = 1 \ldots N$
  - (M-step 1) $\hat{\mu}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)y_i}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)}$
  - (M-step 2) $\hat{\mu}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i y_i}{\sum_{i=1}^{N}\hat{\gamma}_i}$

  - (M-step 3) $\hat{\sigma}_1 = \frac{\sum_{i=1}^{N}(1-\hat{\gamma}_i)(y_i-\hat{\mu}_1)^2}{\sum_{i=1}^{N}(1-\hat{\gamma}_i)}$
  - (M-step 4) $\hat{\sigma}_2 = \frac{\sum_{i=1}^{N}\hat{\gamma}_i(y_i-\hat{\mu}_2)^2}{\sum_{i=1}^{N}\hat{\gamma}_i}$
  - (M-step 5) $\hat{\pi} = \sum_{i=1}^{N}\frac{\hat{\gamma}_i}{N}$

Expectation –Maximization Algorithm

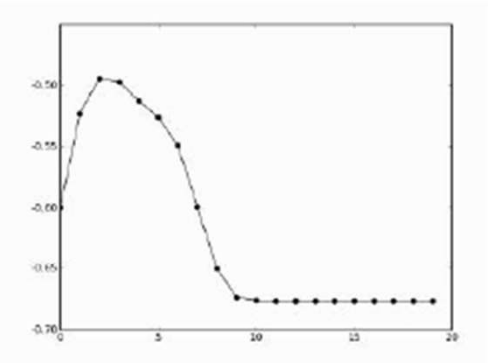
The General Expectation-Maximisation (EM) Algorithm

- Initialisation
  - guess parameters $\hat{\theta}^{(0)}$
- Repeat until convergence:
  - (E-step) compute the expectation $Q(\theta', \hat{\theta}^{(j)}) = E(f(\theta'; D')|D, \hat{\theta}^{(j)})$
  - (M-step) estimate the new parameters $\hat{\theta}^{(j+1)}$ as $\max_{\theta'} Q(\theta', \hat{\theta}^{(j)})$

the need to use model selection to identify the right time to stop learning.

the idea of using cross-validation if there was not enough data.

However, this replaces data with computation time, as many models are trained on different dataset



- An alternative idea is to identify some measure that tells us about how well we can expect this trained model to perform.
- In both cases, based on the way that they are written here, the model with the largest value is taken. Both of
- the measures will favour simple models, which is a form of Occam's razoused:

k is the number of parameters in the model, *N* is the number of training examples, and L is the best (largest) likelihood of the model

Aikake Information Criterium

$$\mathrm{AIC} = \ln(\mathcal{L}) - k$$

Bayesian Information Criterium

$$\mathrm{BIC} = 2\ln(\mathcal{L}) - k\ln N$$

Expectation –Maximization Algorithm

```
# E-step
A = np.dot(W,np.transpose(W)) + np.diag(Psi)
logA = np.log(np.abs(np.linalg.det(A)))
A = np.linalg.inv(A)

WA = np.dot(np.transpose(W),A)
WAC = np.dot(WA,C)
Exx = np.eye(nRedDim) - np.dot(WA,W) + np.dot(WAC,np.transpose(WA))

# M-step
W = np.dot(np.transpose(WAC),np.linalg.inv(Exx))
Psi = Cd - (np.dot(W,WAC)).diagonal()

tAC = (A*np.transpose(C)).sum()

L = -N/2*np.log(2.*np.pi) -0.5*logA - 0.5*tAC
if (L-oldL)<(1e-4):
    print "Stop",i
    break
```

Nearest Neighbour Methods

- Consider the datapoints positioned within input space, the objective is to work out which of the training data are close to it
- identify the *k* nearest neighbours to the test point, and then set the class of the test point to be the most common one out of those for the nearest neighbours
- Computing the distance to each datapoint in the training set is relatively expensive
- For normal Euclidean space, computation involves *d* subtractions and *d* squarings - ignore the square root (since

we only want to know which points are the closest, not the actual distance) –can be done $O(N^2)$ times.

- Choice of *k* is not trivial: Make it
  - too small and nearest neighbour methods are sensitive to noise
  - too large and the accuracy reduces as points that are too far away are considered.
- Some possible effects of changing the size of *k* on the decision boundary are shown in Figure 7.3.

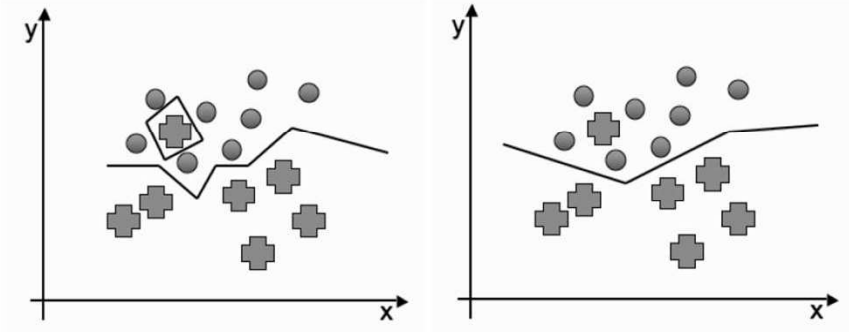


FIGURE 7.3 The nearest neighbours decision boundary with *left:* one neighbour and *right:* two neighbours.

Nearest Neighbour Methods

- This method suffers from the curse of dimensionality

➔the computational costs get higher as the number of dimensions grows.

➔KD-Trees compute this in O(*N* log*N*) time.

➔However, as the number of dimensions increases, so the distance to other datapoints tends to increase

- For the *k*-nearest neighbours algorithm the bias-variance decomposition can be computed as:

$$E((\mathrm{y} - \hat{f}(\mathrm{x}))^2) = \sigma^2 + \left[f(\mathrm{x}) - \frac{1}{k} \sum_{i=0}^{k} f(\mathrm{x}_i) \right]^2 + \frac{\sigma^2}{k}.$$

```
def knn(k,data,dataClass,inputs):

        nInputs = np.shape(inputs)[0]
        closest = np.zeros(nInputs)

        for n in range(nInputs):
                # Compute distances
                distances = np.sum((data-inputs[n,:])**2,axis=1)

                # Identify the nearest neighbours
                indices = np.argsort(distances,axis=0)

                classes = np.unique(dataClass[indices[:k]])
                if len(classes)==1:
                        closest[n] = np.unique(classes)
                else:
                        counts = np.zeros(max(classes)+1)
                        for i in range(k):
                                counts[dataClass[indices[i]]] += 1
                        closest[n] = np.max(counts)

        return closest
```

Nearest Neighbour Smoothing

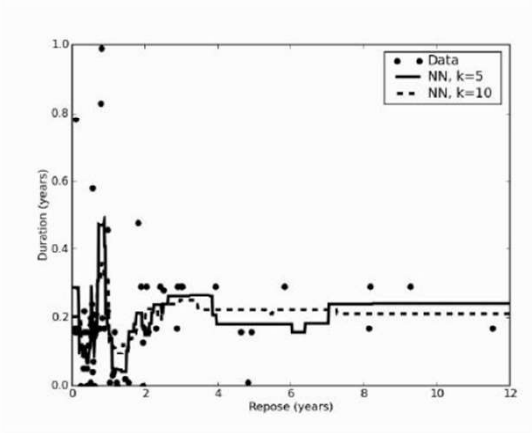
- Nearest neighbour methods can also be used for regression by returning the average value of the neighbours to a point, or a spline or similar fit as the new value
- **kernel smoothers –a common method to smooth**
- They use a kernel (a weighting function between pairs of points) that decides how much emphasis (weight) to put onto the contribution from each datapoint according to its distance from the input.
- Epanechnikov quadratic kernel:
- Using two kernels for smoothing.
- Both of these kernels are designed to give more weight to points that are closer to the current input, with the weights decreasing smoothly to zero as they pass out of the range of the current input, with the range specified by a parameter λ.

$$K_{E,\lambda}(x_0, x) = \begin{cases} 0.75\left(1 - (x_0 - x)^2/\lambda^2\right) & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}$$

and the tricube kernel:

$$K_{T,\lambda}(x_0, x) = \begin{cases} \left(1 - \left|\frac{x_0 - x}{\lambda}\right|^3\right)^3 & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}.$$

- Results of using these kernels are shown in Figure 7.4 on a dataset that consists of the time between eruptions (technically known as the repose) and the duration of the eruptions of Mount Ruapehu, the large volcano in the centre of New Zealand's north island.
- Values of λ of 2 and 4 were used here. Picking λ requires experimentation.
- Large values average over more datapoints, and therefore produce lower variance, but at the cost of higher bias



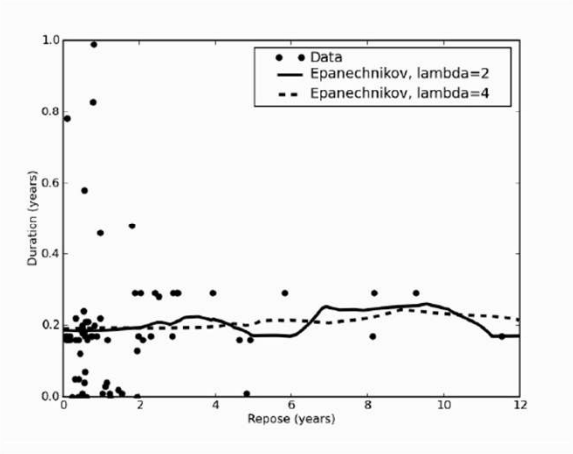


Fig:7.4. Output of the nearest neighbour method and two kernel smoothers on the data of duration and repose of eruptions of Mount Ruapehu 1860–2006

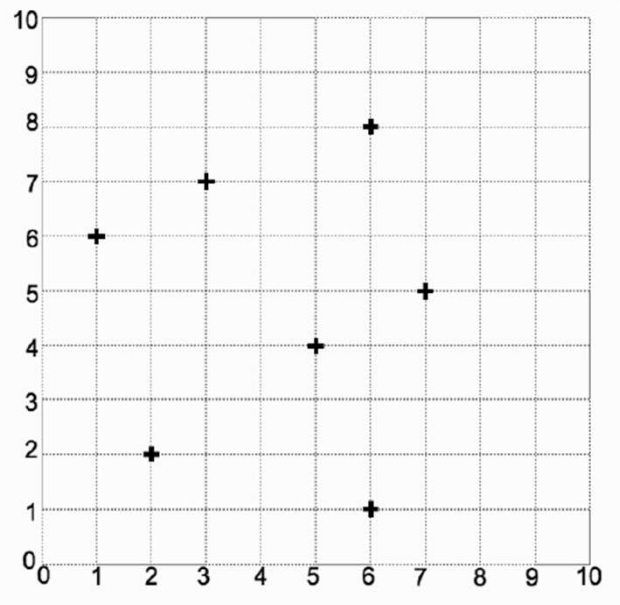
Efficient Distance Computations: the KD-Tree

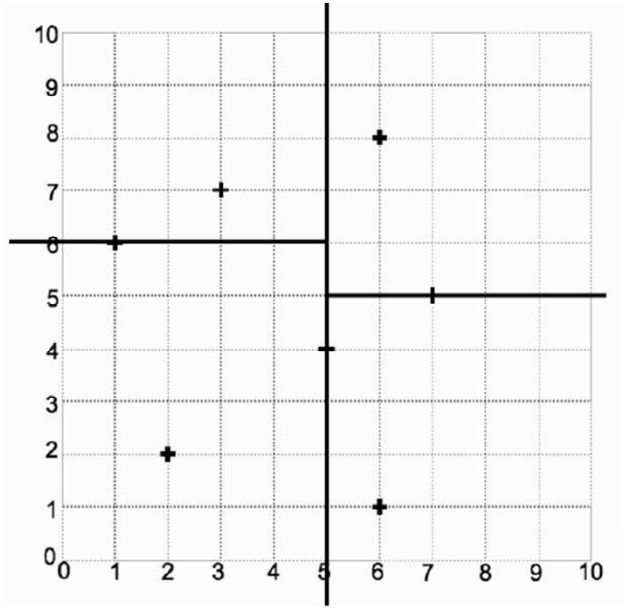
- Computing the distances between all pairs of points is very computationally expensive. Designing an efficient data structure can reduce the computational overhead a lot.
- **KD Tree :**
- A data structure chosen for finding the nearest neighbours.
- It was devised by Friedman and Bentley, and it reduces the cost of finding a nearest neighbour to $O(\log N)$ for $O(N)$ storage.
- The construction of the tree is $O(N \log^2 N)$, with much of the computational cost being in the computation of the median, which with a naïve algorithm requires a sort and is therefore $O(N \log N)$, or can be computed with a randomised algorithm in $O(N)$ time.
- Procedure :
  - create a binary tree by choosing one dimension at a time to split into two, and placing the line through the median of the point coordinates of that dimension.
- . The points themselves end up as leaves of the tree.
- Steps for making the tree are much the same steps as usual for constructing a binary tree:

Steps:

- identify a place to split into two choices -left and right, and then carry on down the tree
- The choice of what to split and where is what makes the KD-tree special. Just one dimension is split in each step, and the position of the split is found by computing the median of the points that are to be split in that one dimension and putting the line there.
- In general, the choice of which dimension to split alternates through the different choices or it can be made randomly.
- The algorithm below cycles through the possible dimensions based on the depth of the tree so far, so that in two dimensions it alternates horizontal and vertical splits.

- (5, 4), (1, 6), (6, 1), (7, 5), (2, 7), (2, 2), (5, 8)





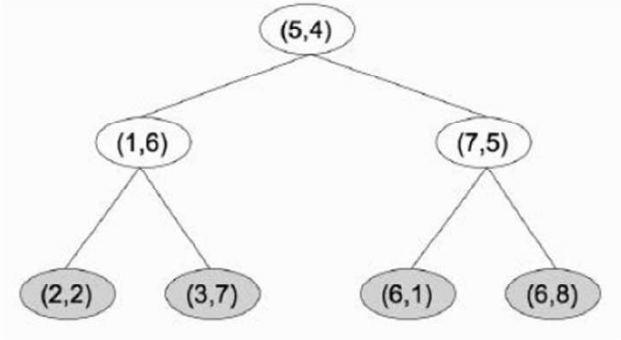
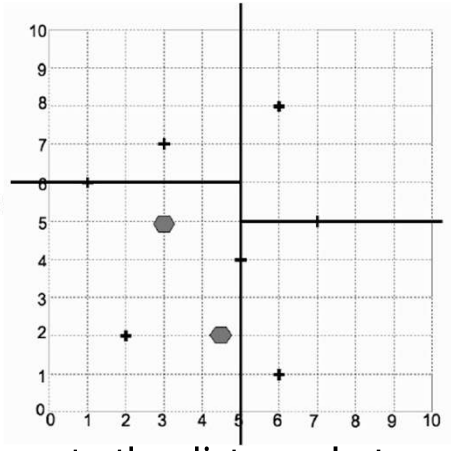


FIGURE 7.7 The KD-tree that made the splits.

KD Tree

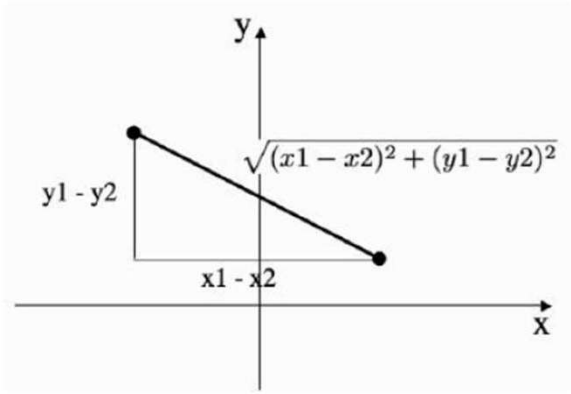
- **Construction of tree :**
  - pick the first coordinate to split on initially, and the median point here is 5, so the split is through $x = 5$.
  - Of those on the left of the line, the median y coordinate is 6, and for those on the right it is 5.
  - At this point we have separated all the points, and so the algorithm terminates with the split shown in Figure 7.6 and the tree shown in Figure 7.7.
- **Testing :**
  - Searching the tree is the same as any other binary tree; we are more interested in finding the neighbours of a test point. easy: starting at the root of the tree
  - you recurse down through the tree comparing just one dimension at a time until you find a le is in the region containing the test point.
  - For the test point (3, 5), (2, 2) is the leaf for the box that (3, 5) is in.
  - However, looking at Figure 7.8 we see that this is not the closest point at all, so we need to
  - do some more work.



- The first thing we label the leaf we have found as a potential nearest neighbour, and compute the distance between the test point and this point, since any other point has to be closer.
- Now we need to check any other boxes that could contain something closer.
- So checking the sibling is not enough —we also need to check the siblings of the parent node, together with its descendants (the cousins of the first point).
- For the test point (4.5,2), the sibling is too far away, but another point (6, 1) is closer

Distance Measures

Euclidean Metric $d_E = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

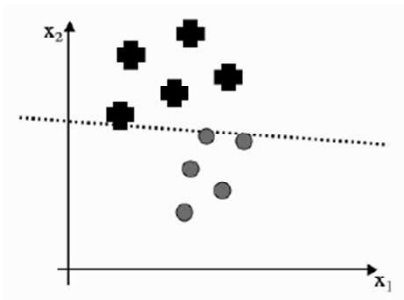


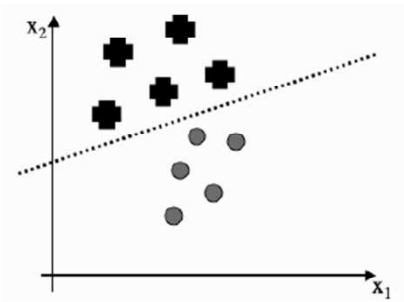
- To fix all of these things in preprocessing, use a different metric that is invariant to these changes, i.e., it does not vary as they do.
- The idea of invariant metrics is to find measures that ignore changes that you don't want.
- So if you want to be able to rotate shapes around and still recognize them, you need a metric that is invariant to rotation.
- A common invariant metric in use for images is the tangent distance, which is an approximation to the Taylor expansion in first derivatives, and works very well for small rotations and scalings;
- For example, it was used to halve the final error rate on nearest neighbour classification of a set of handwritten letters

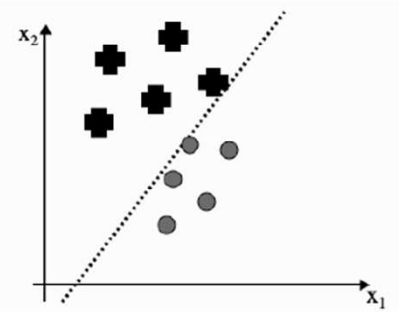
Support Vector Machine

OPTIMAL SEPARATION

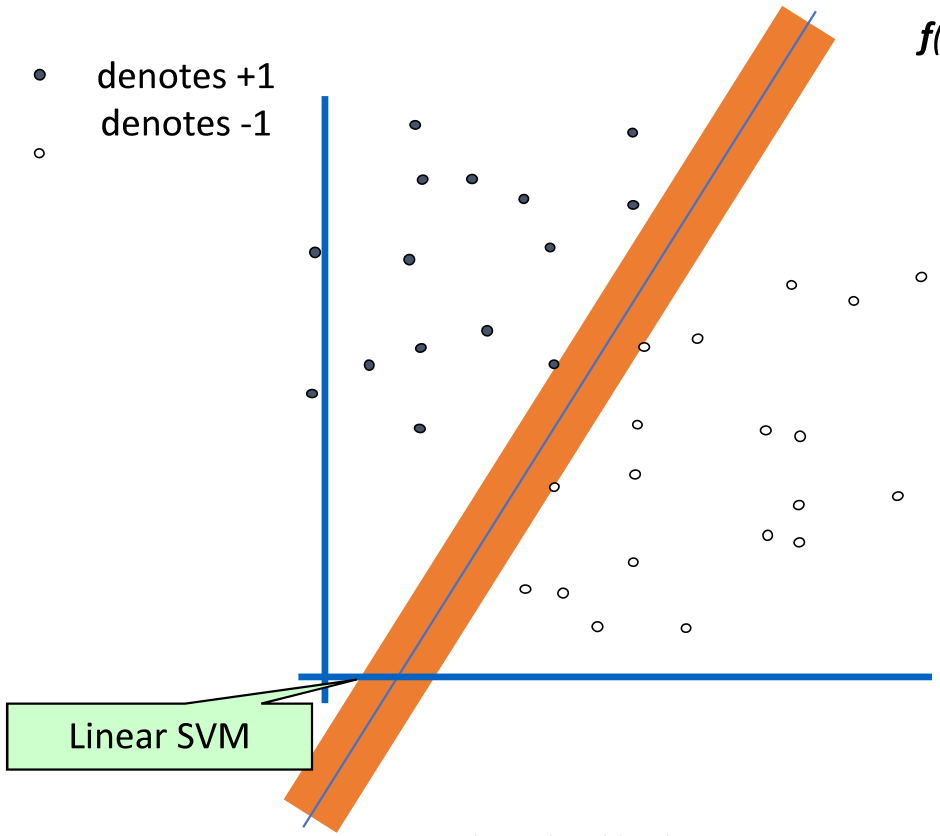
- line that runs through the middle of the separation between the datapoints from the two classes, staying approximately equidistant from the data in both classes
- any point that lies within that region is declared to be too close to the line to be accurately classified.
- This region is symmetric about the line, so that it forms a cylinder about the line in 3D and a hyper-cylinder in higher dimensions.
- The maximum margin (linear) classifier : The radius of this cylinder until we started to put points into a no-man's land, where we don't know which class they are from? This largest radius is known as the margin, labelled *M*.
- Support vectors:. The datapoints in each class that lie closest to the classification line have a name as well.
- Using the argument that the best classifier is the one that goes through the middle of no-man's land, we can now make two arguments:
  - The margin should be as large as possible,
  - The support vectors are the most useful datapoints because they are the ones that we might get wrong.







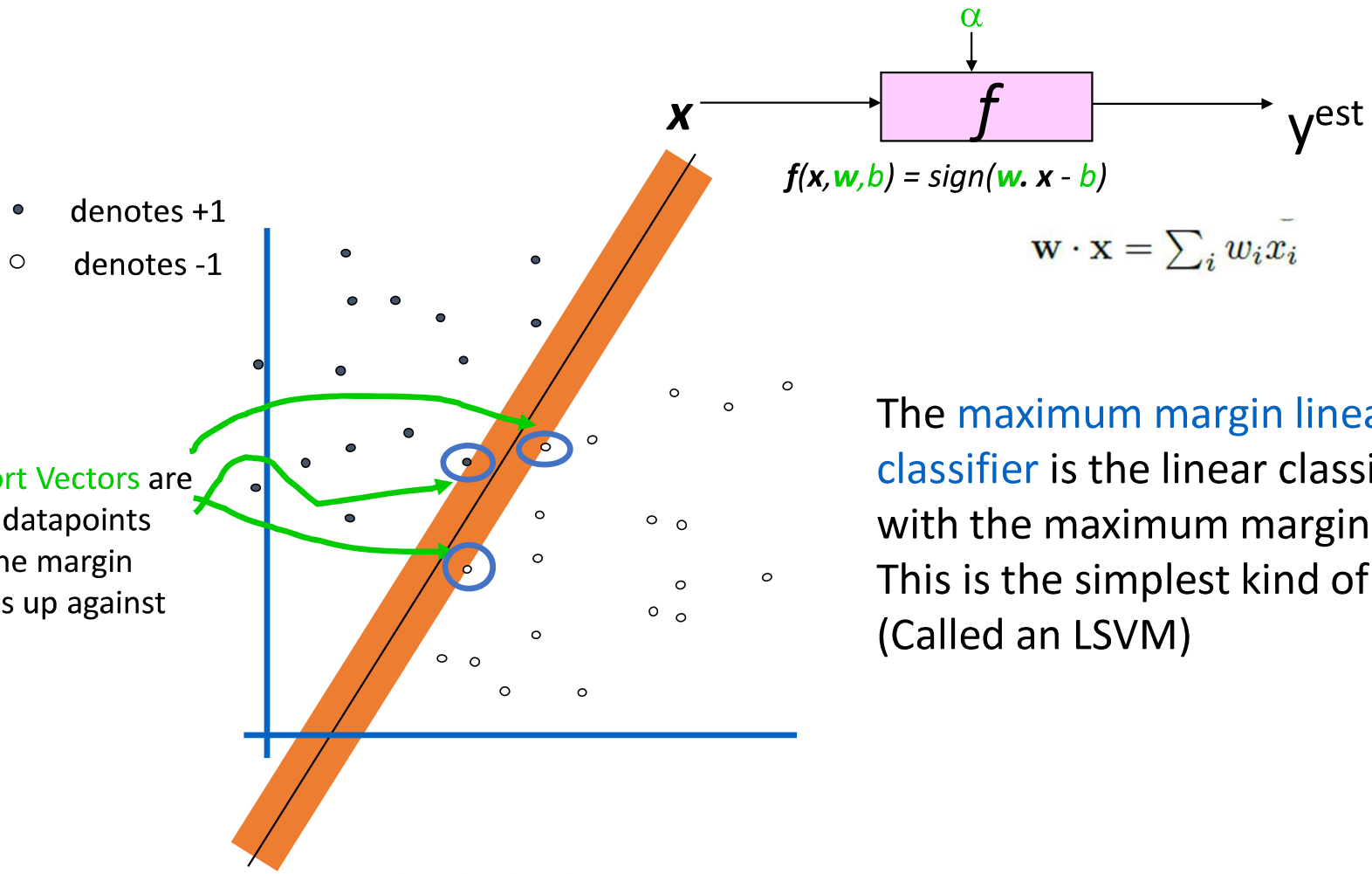
Maximum Margin



$f(x,w,b) = sign(w.\ x - b)$

The maximum margin linear classifier is the linear classifier with the maximum margin. This is the simplest kind of SVM (Called an LSVM)

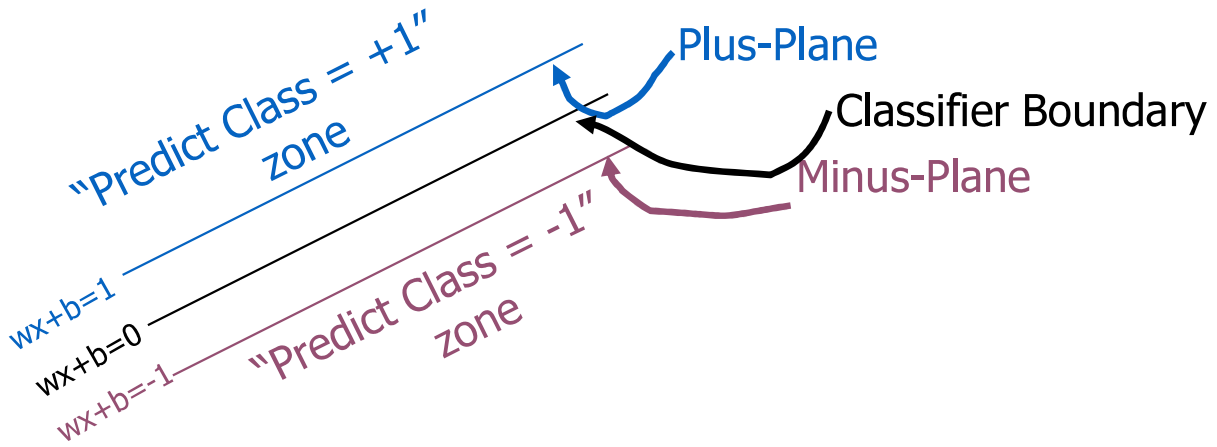
Why Maximum Margin?



$$\mathbf{w} \cdot \mathbf{x} = \sum_i w_i x_i$$

The maximum margin linear classifier is the linear classifier with the maximum margin. This is the simplest kind of SVM (Called an LSVM)

Specifying a line and margin



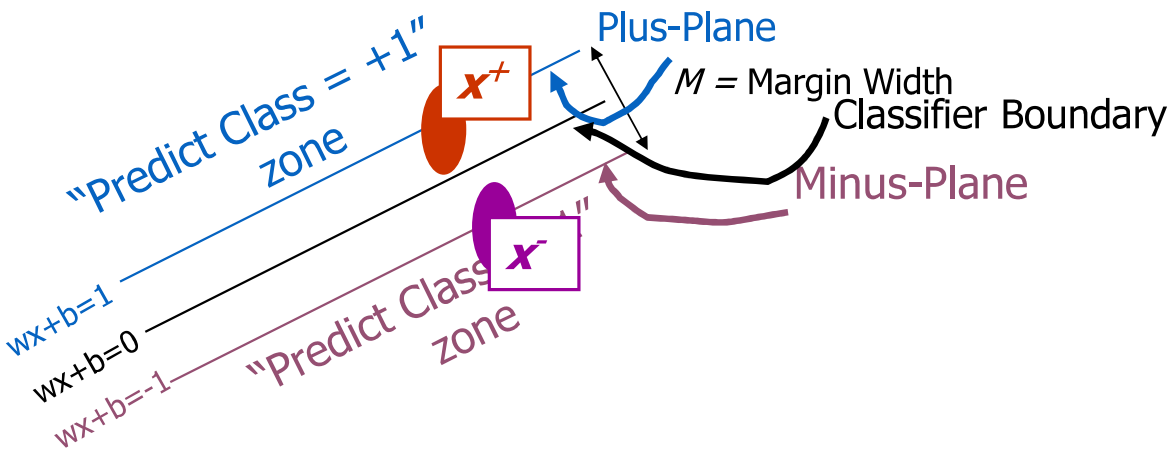
- Plus-plane = $\{ \boldsymbol{x} : \boldsymbol{w} . \boldsymbol{x} + b = +1 \}$
- Minus-plane = $\{ \boldsymbol{x} : \boldsymbol{w} . \boldsymbol{x} + b = -1 \}$

The vector **w** is perpendicular to the Plus Plane
Let $\boldsymbol{x}^-$ be any point on the minus plane
Let $\boldsymbol{x}^+$ be the closest plus-plane-point to $\boldsymbol{x}^-$.

Classify as..

+1	if	$\boldsymbol{w} . \boldsymbol{x} + b >= 1$
-1	if	$\boldsymbol{w} . \boldsymbol{x} + b <= -1$
Universe explodes	if	$-1 < \boldsymbol{w} . \boldsymbol{x} + b < 1$

Computing the margin width



- $\boldsymbol{w} \cdot \boldsymbol{x}^+ + b = +1$
- $\boldsymbol{w} \cdot \boldsymbol{x}^- + b = -1$
- $\boldsymbol{x}^+ = \boldsymbol{x}^- + \lambda \boldsymbol{w}$
- $|\boldsymbol{x}^+ - \boldsymbol{x}^-| = M = |\lambda \boldsymbol{w}|$

- Plus-plane = $\{ \boldsymbol{x} : \boldsymbol{w} \cdot \boldsymbol{x} + b = +1 \}$
- Minus-plane = $\{ \boldsymbol{x} : \boldsymbol{w} \cdot \boldsymbol{x} + b = -1 \}$
- The vector **w** is perpendicular to the Plus Plane
- Let $\boldsymbol{x}^-$ be any point on the minus plane
- Let $\boldsymbol{x}^+$ be the closest plus-plane-point to $\boldsymbol{x}^-$.

Any location in R^m: not necessarily a datapoint

$\boldsymbol{w} \cdot (\boldsymbol{x}^- + \lambda \boldsymbol{w}) + b = 1$

$\Rightarrow \boldsymbol{w} \cdot \boldsymbol{x}^- + b + \lambda \boldsymbol{w} \cdot \boldsymbol{w} = 1$

$\Rightarrow -1 + \lambda \boldsymbol{w} \cdot \boldsymbol{w} = 1$

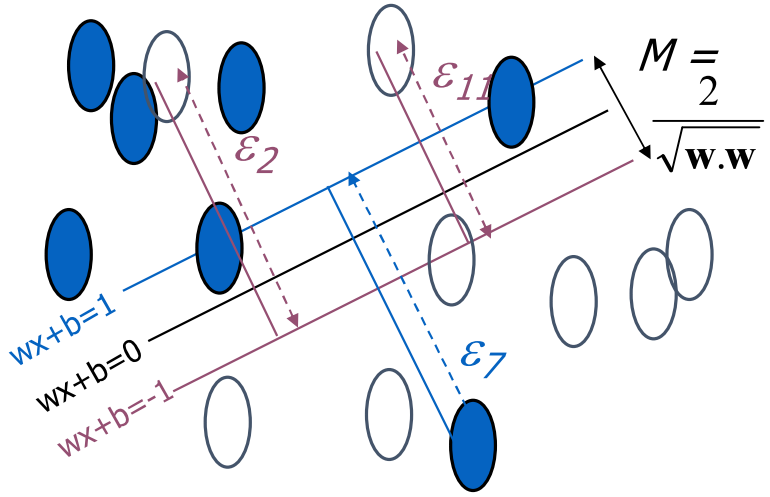
$\Rightarrow \lambda = \frac{2}{\mathbf{w} \cdot \mathbf{w}}$

$M = \lambda |\mathbf{w}| = \lambda \sqrt{\mathbf{w} \cdot \mathbf{w}}$

$M = \frac{2\sqrt{\mathbf{w} \cdot \mathbf{w}}}{\mathbf{w} \cdot \mathbf{w}} = \frac{2}{\sqrt{\mathbf{w} \cdot \mathbf{w}}}$

i.e The total distance between the support vectors, so that it would be twice the one **2/|w|** making *M* as large as possible is the same as making $\mathbf{w}^T\mathbf{w}$ as small as possible

Learning Maximum Margin with Noise



Given guess of $\mathbf{w}$, b we can

- Compute sum of distances of points to their correct zones
- Compute the margin width

Assume R datapoints, each $(\mathbf{x}_k, y_k)$ where $y_k = +/- 1$

For m input dimnesions , original (noiseless data) QP had $m+1$ variables: $w_1, w_2, \ldots w_m$ and b.

Our new (noisy data) QP has $m+1+R$ variables: $w_1, w_2, \ldots w_m, b, \varepsilon_k, \varepsilon_1, \ldots \varepsilon_R$

What should our quadratic optimization criterion be?

Minimize

$$\frac{1}{2}\mathbf{w}.\mathbf{w} + C\sum_{k=1}^{R}\varepsilon_k$$

How many constraints will we have? $2R$

What should they be?

$\mathbf{w} . \mathbf{x}_k + b >= 1-\varepsilon_k$ if $y_k = 1$

$\mathbf{w} . \mathbf{x}_k + b <= -1+\varepsilon_k$ if $y_k = -1$

$\varepsilon_k >= 0$ for all k

Learning Maximum Margin with Noise

- The same can be written as

Distance from support vector to $H(\mathbf{w}, b)$

$$\frac{\mathbf{w}^\top \mathbf{x}_i + b}{\|\mathbf{w}\|} = \frac{\pm 1}{\|\mathbf{w}\|}$$

$$\text{Margin} = \left| \frac{1}{\|\mathbf{w}\|} - \frac{-1}{\|\mathbf{w}\|} \right| = \frac{2}{\|\mathbf{w}\|}$$

$$\begin{aligned} &\text{minimize} && \frac{1}{2}\mathbf{w}^\top \mathbf{w} \\ &\text{subject to} && y_i(\mathbf{w}^\top \mathbf{x}_i + b) \geq 1 \qquad i = 1, 2, \ldots, n \end{aligned}$$

1. Convex quadratic program
2. Linear inequality constraints (many!)
3. $d + 1$ parameters, n constraints

A convex problem :

A convex problem is one where if we take any two points on the line and join them with a straight line, then every point on the line will be above the curve.

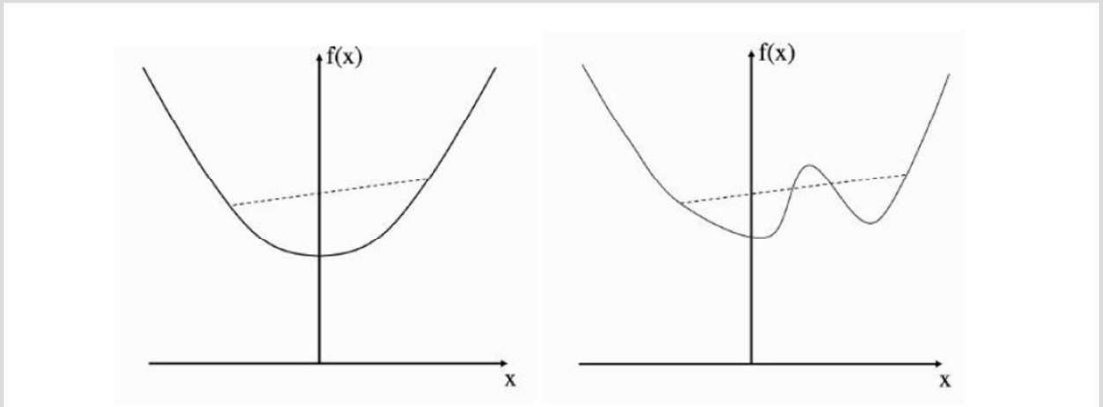


FIGURE 8.4 A function is convex if every straight line that links two points on the curve does not intersect the curve anywhere else. The function on the left is convex, but the one on the right is not, as the dashed line shows.

Figure 8.4 shows an example of a convex and a non-convex function. Convex functions have a unique minimum, which is easy to see in one dimension, and remains true in any number of dimensions

Optimization

- When we find that optimal solution, the Karush–Kuhn–Tucker (KKT) conditions will be satisfied.

  i. e For all values of *i* from 1 to *n*, and where the * denotes the optimal value of each parameter:

$$\begin{aligned} \lambda_i^*(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*)) &= 0 \\ 1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*) &\leq 0 \\ \lambda_i^* &\geq 0, \end{aligned}$$

- where the λ*i* are positive values known as Lagrange multipliers, which are a standard approach to solving equations with equality constraints.

$$\text{if } \lambda_i \neq 0 \text{ then } (1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*)) = 0$$

- This is only true for the support vectors (the SVMs provide a sparse representation of the data), andso we only have to consider them, and can ignore the rest. In the jargon, the support vectors are those vectors in the active set of constraints. For the support vectors the constraints are equalities instead of inequalities.

Linear SVMs Mathematically (cont.)

- Then we can formulate the *quadratic optimization problem:*

Find $\mathbf{w}$ and b such that $\rho = \frac{2}{\|\mathbf{w}\|}$ is maximized

and for all $(\mathbf{x}_i, y_i)$, $i=1..n$: $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$

Which can be reformulated as:

Find $\mathbf{w}$ and b such that

$\Phi(\mathbf{w}) = \|\mathbf{w}\|^2 = \mathbf{w}^T\mathbf{w}$ is minimized

and for all $(\mathbf{x}_i, y_i)$, $i=1..n$: $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$

Need to optimize a quadratic function subject to linear constraints.

Quadratic optimization problems are a well-known class of mathematical programming problems for which several (non-trivial) algorithms exist.

The solution involves constructing a dual problem where a Lagrange multiplier αi is associated with every inequality constraint in the primal (original) problem:

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{n} \lambda_i (1 - t_i(\mathbf{w}^T\mathbf{x}_i + b))$$

$$\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w} - \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i$$

differentiate this function with respect to the elements of **w** and *b*:

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \lambda_i t_i.$$

Dual Problem

- If we set the derivatives to be equal to zero, so that we find the saddle points (maxima) of the function

$$\mathbf{w}^* = \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i, \sum_{i=1}^{n} \lambda_i t_i = 0.$$

- We can substitute these expressions at the optimal values of **w** and *b* into Equation and, after a little bit of rearranging, we get (where is the vector of the *i*):

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{n} \lambda_i (1 - t_i(\mathbf{w}^T\mathbf{x}_i + b))$$

$$\mathcal{L}(\mathbf{w}^*, b^*, \boldsymbol{\lambda}) = \sum_{i=1}^{n} \lambda_i - \sum_{i=1}^{n} \lambda_i t_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \lambda_i \lambda_j t_i t_j \mathbf{x}_i^T \mathbf{x}_j$$

- using the derivative with respect to *b* we can treat the middle term as 0.
- This equation is known as the dual problem, and the aim is to maximise it with respect to the λ*i* variables. The constraints are that λ*i*= 0 for all *i*, and $\sum_{i=1}^{n} \lambda_i t_i = 0.$

Prediction

- Substitute w* for a support vector $ti(\mathbf{w}T\,\mathbf{x}i + b) = 1$,b* can be

$$b^* = \frac{1}{N_s} \sum_{\text{support vectors } j} \left(t_j - \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i^T \mathbf{x}_j \right)$$

- So, to make a prediction for a new datapoint z

$$\mathbf{w}^{*T}\mathbf{z} + b^* = \left(\sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i \right)^T \mathbf{z} + b^*$$

- This means that to classify a new point, we just need to compute the inner product between the new datapoint and the support vectors.

Slack Variables for non linearly Separable Problems

Objective: find a good separating hyper-plane for the non-separable case

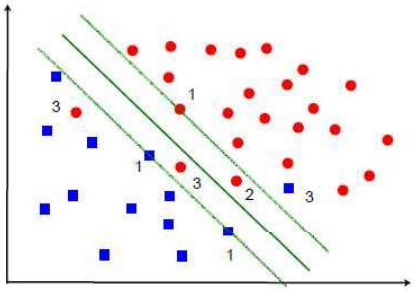
Problem: Cannot satisfy $y_i[\mathbf{w}^\top \mathbf{x}_i + b] \geq 1$ for all i

Solution: *Slack* variables

$$\begin{aligned} \mathbf{w}^\top \mathbf{x}_i + b &\geq +1 - \xi_i && \text{for } y_i = +1, \\ \mathbf{w}^\top \mathbf{x}_i + b &\leq -1 + \xi_i && \text{for } y_i = -1, \\ \xi_i &\geq 0 && k = 1, 2, \ldots, n. \end{aligned}$$

An error occurs if $\xi_i > 1$. Thus,

$$\sum_{i=1}^{n} I(\xi_i > 1) = \#\text{ errors}$$



- A non-linearly separable dataset cannot satisfy the constraints for all datapoints.
- Solution :Introduce some slack variables η*i* ≥ 0 so that the constraints become

  ti(**w**^{T} **x***i* +*b*) 1−η*i* , For inputs that are correct, we set η*i* = 0
- when comparing classifiers, we should consider the case where
  - one classifier makes a mistake by putting a point just on the wrong side of the line
  - Another Classifier puts the same point a long way onto the wrong side of the line.
- The first classifier is better than the second, because the mistake was not as serious, so we should include this information in our minimization criterion.
- We want to add a term into the minimization problem so that we will now minimize **w**^{T}**w** + *C*× (distance of misclassified points from the correct boundary line)

  where C is a tradeoff parameter that decides how much weight to put onto each of the two criteria:
  - small *C* means ➔ large margin over a few errors
  - large *C* ➔*Small margin*
- This transforms the problem into a soft-margin classifier, since we are allowing for a few mistakes.
- In a mathematical way, the function that we want to minimise is $L(\mathbf{w}, \epsilon) = \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^{n} \epsilon_i$.

Slack Variables

- When a problem can be converted to another problem whose solution is easier to compute and also provides the solution for the original problem, the two problems are said to exhibit **Duality** and the vice-versa holds true.
- The derivation of the dual problem that we worked out earlier still holds, except that $0 \leq \lambda i \leq C$, and the support vectors are now those vectors with $i > 0$. The KKT conditions are slightly different,

$$\begin{aligned} \lambda_i^*(1 - t_i(\mathbf{w}^{*T}\mathbf{x}_i + b^*) - \eta_i) &= 0 \\ (C - \lambda_i^*)\eta_i &= 0 \\ \sum_{i=1}^{n} \lambda_i^* t_i &= 0 \end{aligned}$$

- The second condition tells us that, if $i < C$, then $i = 0$, which means that these are the support vectors.
- If $\lambda i = C$, then the first condition tells us that if $\lambda i > 1$ then the classifier made a mistake.
- The problem with this is that it is not as clear how to choose a limited set of vectors, and so most of our training set will be support vectors

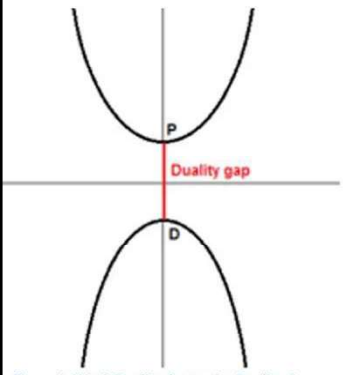


Figure 1: Weak Duality due to the Duality Gap.

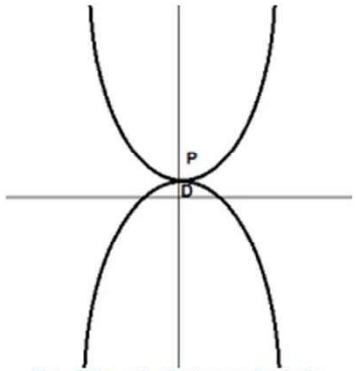
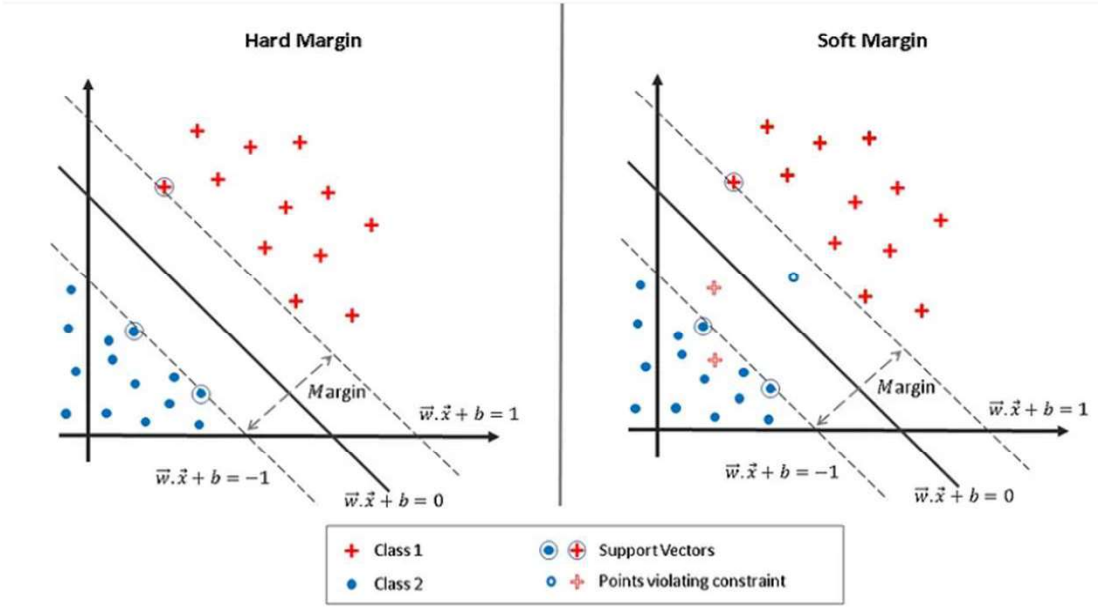
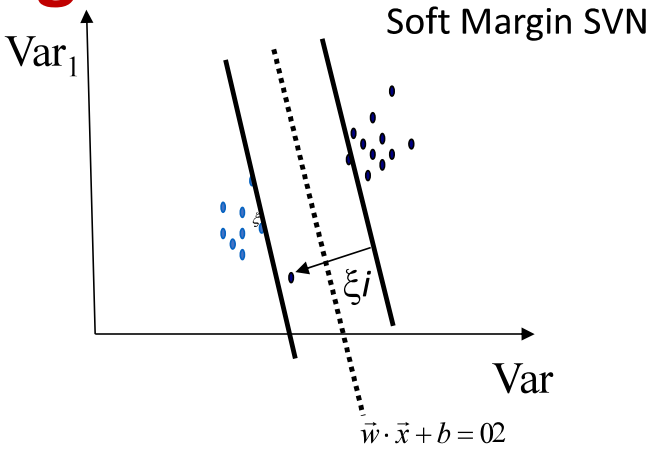


Figure 2: Strong Duality due to no Duality Gap (Exhibits Complementary Slackness)

Robustness of Soft vs Hard Margin SVMs





To allow the SVM to make some mistakes and yet keep the margin as wide as possible.

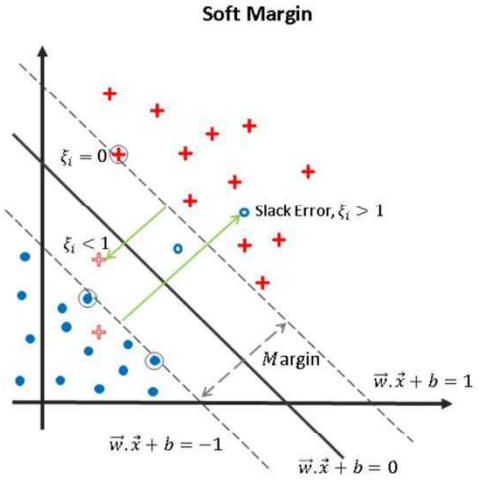
It means that there can be no points inside of the hyperplane as shown in the below figure a ***(Vanilla SVM).***

Soft-Margin always have a solution

Soft-Margin is more robust to outliers

Smoother surfaces (in the non-linear case)

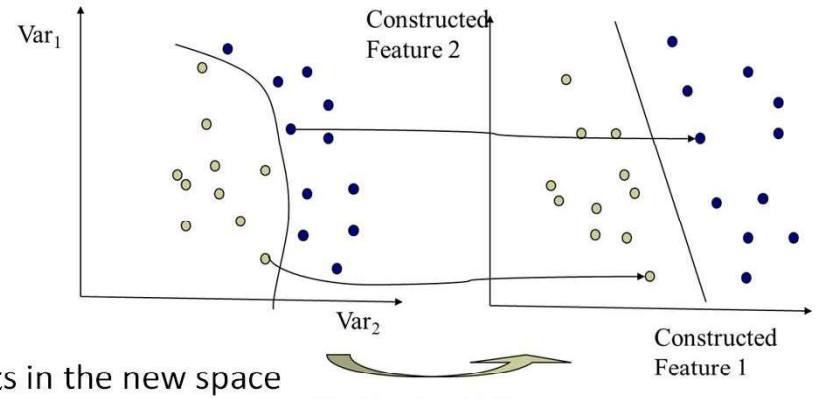
Hard-Margin does not require to guess the cost parameter(requires no parameters at all)



Kernels

- The nonlinear data can be separated linearly by modifying the features in some way
- Find function Φ(x) to map to a different space, then SVM formulation becomes:

$$\min \frac{1}{2}\|w\|^2 + C\sum_i \xi_i \qquad s.t.\ \ y_i(w\cdot\Phi(x)+b) \geq 1-\xi_i, \forall x_i \qquad \xi_i \geq 0$$



227

- Transform the Data by making Φ(xi) from input **x***i* , weights *w* are now weights in the new space

$$\mathbf{w}^T\mathbf{x}+b = \left(\sum_{i=1}^{n}\lambda_i t_i \phi(\mathbf{x}_i)\right)^T \phi(\mathbf{z}) + b.$$

- Explicit mapping expensive if Φ(x) is very high dimensional. Solving the problem without explicitly mapping the data is desirable
- A basis : Total Input vector Φ(x) having polynomial of degree 2 →
  - constant
  - scalar values of input elements *x*1, *x*2, . . . , *xd*, and then
  - the squares of each input element x_1^2, x_2^2 x_3^2. $\Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_3, x_1^2, x_2^2, x_3^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \sqrt{2}x_2x_3).$
  - the products of each pair of elements $x_1x_2, x_1x_3, \ldots, x_{d-1}x_d.$

Total no.of elements- $d^2/2$

If there was just one feature, *x*1, then we would have changed this from a one-dimensional problem into a three-dimensional one (1, x_1, x_1^2).

Quadratic Basis Functions

$$\Phi(\mathbf{x}) = \begin{pmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \vdots \\ \sqrt{2}x_m \\ x_1^2 \\ x_2^2 \\ \vdots \\ x_m^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \vdots \\ \sqrt{2}x_1x_m \\ \sqrt{2}x_2x_3 \\ \vdots \\ \sqrt{2}x_1x_m \\ \vdots \\ \sqrt{2}x_{m-1}x_m \end{pmatrix}$$

Constant Term

Linear Terms

Pure Quadratic Terms

Quadratic Cross-Terms

Number of terms (assuming m input dimensions) = (m+2)-choose-2 = (m+2)(m+1)/2= (as near as makes no difference) $m^2/2$

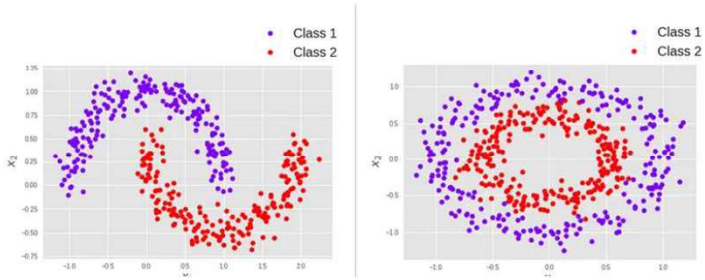
$$\Phi(\mathbf{a}) \bullet \Phi(\mathbf{b}) = \begin{pmatrix} 1 \\ \sqrt{2}a_1 \\ \sqrt{2}a_2 \\ \vdots \\ \sqrt{2}a_m \\ a_1^2 \\ a_2^2 \\ \vdots \\ a_m^2 \\ \sqrt{2}a_1a_2 \\ \sqrt{2}a_1a_3 \\ \vdots \\ \sqrt{2}a_1a_m \\ \sqrt{2}a_2a_3 \\ \vdots \\ \sqrt{2}a_1a_m \\ \vdots \\ \sqrt{2}a_{m-1}a_m \end{pmatrix} \bullet \begin{pmatrix} 1 \\ \sqrt{2}b_1 \\ \sqrt{2}b_2 \\ \vdots \\ \sqrt{2}b_m \\ b_1^2 \\ b_2^2 \\ \vdots \\ b_m^2 \\ \sqrt{2}b_1b_2 \\ \sqrt{2}b_1b_3 \\ \vdots \\ \sqrt{2}b_1b_m \\ \sqrt{2}b_2b_3 \\ \vdots \\ \sqrt{2}b_1b_m \\ \vdots \\ \sqrt{2}b_{m-1}b_m \end{pmatrix}$$

$$= 1 + \sum_{i=1}^{m} 2a_ib_i + \sum_{i=1}^{m} a_i^2b_i^2 + \sum_{i=1}^{m}\sum_{j=i+1}^{m} 2a_ia_jb_ib_j$$

The Kernel Trick

- $\Phi(x_i) \cdot \Phi(x_j)$: means, map data into new space, then take the inner product of the new vectors
- We can find a function such that: $K(x_i \cdot x_j) = \Phi(x_i) \cdot \Phi(x_j)$, i.e., the image of the inner product of the data is the inner product of the images of the data
- For the three dimensions data $\Phi(\mathrm{x})^T \Phi(\mathrm{y})$: $\Phi(\mathrm{x})^T \Phi(\mathrm{y}) = 1 + 2\sum_{i=1}^{d} x_i y_i + \sum_{i=1}^{d} x_i^2 y_i^2 + 2 \sum_{i,j=1;i<j}^{d} x_i x_j y_i y_j.$
- Factorising it produces ➔ $(1+\mathrm{x}^T \mathrm{y})^2$
- The dot product here is in the original space, so it only requires d multiplications, which is obviously much better—this part of the algorithm has now been reduced from O(d^2) to O(d).
- To remove the problem of computing the dot products of all the extended basis vectors, which is expensive
  - the computation of a kernel matrix (Gram matrix) **K** that is made from the dot product of the original vectors, which is only linear in cost
  - No need to explicitly map the data into the high-dimensional space to solve the optimization problem (for training)
- How do we classify without explicitly mapping the new instances?
  - No need to know have to know what (·) is, provided you know a kernel
- No need to do any computations in those higher-dimensional spaces

but only in the original (relatively cheap) low-dimensional space



Commonly used Kernels

Mercer's theorem relates kernel functions and inner product spaces

- Suppose that for all finite sets of points $\{\boldsymbol{x}_p\}_{p=1}^{N}$ and real numbers $\{\boldsymbol{a}\}_{p=1}^{\infty}$

$$\sum_{i,j} a_j a_i k(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq 0$$

- Then $\boldsymbol{K}$ is called a positive semidefinite kernel
- And can be written as

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

- For some vector-valued function $\boldsymbol{\phi}(\boldsymbol{x})$

Kernel: A symmetric function $K : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$

Inner product kernels: In addition

$$K(\mathrm{x}, \mathrm{z}) = \boldsymbol{\Phi}(\mathrm{x})^\top \boldsymbol{\Phi}(\mathrm{z})$$

Motivation: $\boldsymbol{\Phi} \in \mathbb{R}^D$, where D may be very large - inner products expensive

- polynomials up to some degree s in the elements x_k of the input vector $x_1 \times x_4$) with kernel:

$$\mathrm{K}(\mathrm{x}, \mathrm{y}) = (1 + \mathrm{x}^T\mathrm{y})^s$$

For $s = 1$ this gives a linear kernel

- sigmoid functions of the x_ks with parameters κ and δ, and kernel:

$$\mathrm{K}(\mathrm{x}, \mathrm{y}) = \tanh(\kappa \mathrm{x}^T\mathrm{y} - \delta)$$

- radial basis function expansions of the x_ks with parameter σ and kernel:

$$\mathrm{K}(\mathrm{x}, \mathrm{y}) = \exp\left(-(\mathrm{x} - \mathrm{y})^2/2\sigma^2\right)$$

Problems in SVM

- Testing
  - To use the kernel trick in order to reduce the computations for the testing set.

  The forward computation for the weights where

  2. To compute replace it $\mathbf{w}^T\Phi(\mathbf{x})$
- Overfitting $\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j),$ $\mathbf{w} = \sum_{i \text{ where } \lambda_i > 0} \lambda_i t_i \Phi(\mathbf{x}_i)$

- The overfitting problem will be reduced as optimizing $\mathbf{w}^T\mathbf{w}$ which tries to keep **w** small, which means that many of the parameters are kept close to 0.

Solving XOR using SVM

The SVM can solve XOR using a

Input vector $\mathbf{x}$	Desired response d
$(-1,-1)$	-1
$(-1,+1)$	$+1$
$(+1,-1)$	$+1$
$(+1,+1)$	-1

Basis of all terms upto quadratic of two features where $\sqrt{2}$ is to keep multiplication simple

$$1, \sqrt{2}x_1, \sqrt{2}x_2, x_1x_2, x_1^2, x_2^2,$$

And dual problem eqn becomes

$$\sum_{i=1}^{4}\lambda_i - \sum_{i,j}^{4}\lambda_i\lambda_j t_i t_j \Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j)$$

Subject to the constraints

$\lambda_1 - \lambda_2 + \lambda_3 - \lambda_4 = 0, \lambda_i \geq 0\ i = 1 \ldots 4.$

- Polynomial kernel in 2D, $c = 1, p = 2$

$$k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T\boldsymbol{x}' + 1)^2 = (x_1x_1' + x_2x_2' + 1)^2$$

$$= x_1^2x_1'^2 + x_2^2x_2'^2 + 2x_1x_1'x_2x_2' + 2x_1x_1' + 2x_2x_2' + 1$$

- If we define

$$\phi(\boldsymbol{x}) = \left[x_1^2, x_2^2, \sqrt{2}\mathrm{x}_1\mathrm{x}_2, \sqrt{2}\mathrm{x}_1, \sqrt{2}\mathrm{x}_2, 1\right]^T$$

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

In general, $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = (1 + \boldsymbol{x}_i^T\boldsymbol{x}_j)^2$

For example,

$$K_{11} = k\left(\begin{bmatrix}-1\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}\right) = (1+2)^2 = 9$$

$$K_{12} = k\left(\begin{bmatrix}-1\\-1\end{bmatrix}, \begin{bmatrix}-1\\+1\end{bmatrix}\right) = (1+0)^2 = 1$$

So

$$K = \begin{bmatrix}9 & 1 & 1 & 1\\ 1 & 9 & 1 & 1\\ 1 & 1 & 9 & 1\\ 1 & 1 & 1 & 9\end{bmatrix}$$

Solving XOR using SVM

XOR: first compute the kernel matrix

- Or compute $\phi(\boldsymbol{x}_i)$ and their inner products, e.g.,
  - Remember, $\phi(\boldsymbol{x}) = [x_1^2, x_2^2, \sqrt{2}x_1x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1]^T$ For b
  - Since $\phi(\boldsymbol{x})$ includes 1, no need for separate b later

$$\phi(\boldsymbol{x}_1) = \phi\left(\begin{bmatrix}-1\\-1\end{bmatrix}\right) = [1,1,\sqrt{2},-\sqrt{2},-\sqrt{2},1]^T$$

$$\phi(\boldsymbol{x}_2) = \phi\left(\begin{bmatrix}-1\\+1\end{bmatrix}\right) = [1,1,-\sqrt{2},-\sqrt{2},\sqrt{2},1]^T$$

- Then

$$K_{11} = \phi^T(\boldsymbol{x}_1)\phi(\boldsymbol{x}_1) = 1+1+2+2+2+1 = 9$$
$$K_{12} = \phi^T(\boldsymbol{x}_1)\phi(\boldsymbol{x}_2) = 1+1-2+2-2+1 = 1$$

- Results in same K matrix, but more computation

XOR: Combine class labels into K

- Define matrix $\tilde{K}$ such that $\tilde{K}_{ij} = K_{ij}d_id_j$
- Recall $\boldsymbol{d} = [-1,+1,+1,-1]^T$

$$\tilde{K} = \begin{bmatrix}+9 & -1 & -1 & +1\\ -1 & +9 & +1 & -1\\ -1 & +1 & +9 & -1\\ +1 & -1 & -1 & +9\end{bmatrix}$$

Solving XOR using SVM

XOR: Solve dual Lagrangian for $\boldsymbol{a}$

- Find fixed points of

$$\tilde{L}(\boldsymbol{a}) = \mathbf{1}^T\boldsymbol{a} - \frac{1}{2}\boldsymbol{a}^T\tilde{K}\boldsymbol{a}$$

- Set matrix gradient to 0

$$\nabla\tilde{L} = \mathbf{1} - \tilde{K}\boldsymbol{a} = \mathbf{0}$$

$$\Rightarrow \boldsymbol{a} = \tilde{K}^{-1}\mathbf{1} = \left[\frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\right]^T$$

- Satisfies all conditions: $a_p \geq 0\ \forall p \quad \sum_p a_p d_p = 0$
  - So this is the solution
- All points are support vectors

XOR: Compute $\boldsymbol{w}$ (including b) from $\boldsymbol{a}$

$$\boldsymbol{w} = \sum_p a_p d_p \boldsymbol{x}_p$$

$$= -\frac{1}{8}\phi(\boldsymbol{x}_1) + \frac{1}{8}\phi(\boldsymbol{x}_2) + \frac{1}{8}\phi(\boldsymbol{x}_3) - \frac{1}{8}\phi(\boldsymbol{x}_4)$$

$$= \frac{1}{8}\left(-\begin{bmatrix}1\\1\\\sqrt{2}\\-\sqrt{2}\\-\sqrt{2}\\1\end{bmatrix} + \begin{bmatrix}1\\1\\-\sqrt{2}\\-\sqrt{2}\\\sqrt{2}\\1\end{bmatrix} + \begin{bmatrix}1\\1\\-\sqrt{2}\\\sqrt{2}\\-\sqrt{2}\\1\end{bmatrix} - \begin{bmatrix}1\\1\\\sqrt{2}\\\sqrt{2}\\\sqrt{2}\\1\end{bmatrix}\right) = \begin{bmatrix}0\\0\\-\frac{1}{\sqrt{2}}\\0\\0\\0\end{bmatrix} \leftarrow b$$

Solving XOR using SVM

XOR: Examine prediction function

- Prediction function

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x})$$

$$= \left[0,0,-\frac{1}{\sqrt{2}},0,0,0\right]^T \left[x_1^2, x_2^2, \sqrt{2}x_1x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1\right]$$

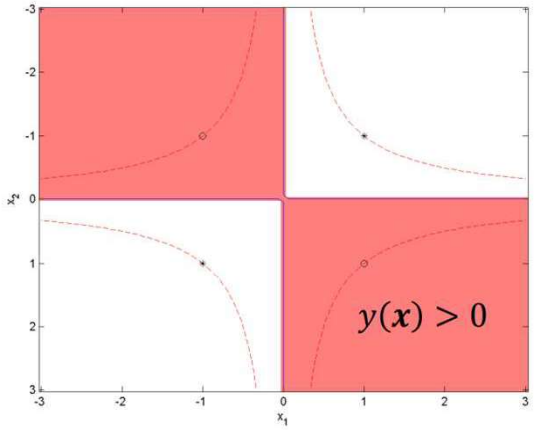
$$= -x_1x_2$$

- Predictions are based on product of the dimensions

$$y(\boldsymbol{x}_1) = -(-1)(-1) = -1$$
$$y(\boldsymbol{x}_2) = -(-1)(+1) = +1$$
$$y(\boldsymbol{x}_3) = -(+1)(-1) = +1$$
$$y(\boldsymbol{x}_4) = -(+1)(+1) = -1$$

XOR: Decision boundaries

- Decision boundary at $y(\boldsymbol{x}) = -x_1x_2 = 0$
- Support vectors at $y(\boldsymbol{x}) = -x_1x_2 = 1$



XOR – Objective Function of dual Form Derivation

$$K(\mathbf{x}, \mathbf{x}_i) = (1 + \mathbf{x}^T\mathbf{x}_i)^2$$

With $\mathbf{x} = [x_1, x_2]^T$ and $\mathbf{x}_i = [x_{i1}, x_{i2}]^T$, we may thus express the inner-product kernel $K(\mathbf{x}, \mathbf{x}_i)$ in terms of *monomials* of various orders as follows:

$$K(\mathbf{x}, \mathbf{x}_i) = 1 + x_1^2 x_{i1}^2 + 2x_1x_2x_{i1}x_{i2} + x_2^2 x_{i2}^2 + 2x_1x_{i1} + 2x_2x_{i2}$$

The image of the input vector $\mathbf{x}$ induced in the feature space is therefore deduced to be

$$\varphi(\mathbf{x}) = \left[1, x_1^2, \sqrt{2}x_1x_2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2\right]^T$$

Similarly,

$$\varphi(\mathbf{x}_i) = \left[1, x_{i1}^2, \sqrt{2}x_{i1}x_{i2}, x_{i2}^2, \sqrt{2}x_{i1}, \sqrt{2}x_{i2}\right]^T, \qquad i = 1, 2, 3, 4$$

$$\mathbb{K} = \begin{bmatrix} 9 & 1 & 1 & 1 \\ 1 & 9 & 1 & 1 \\ 1 & 1 & 9 & 1 \\ 1 & 1 & 1 & 9 \end{bmatrix}$$

- The objective functon of the dual form is

$$Q(\alpha) = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 - \frac{1}{2}(9\alpha_1^2 - 2\alpha_1\alpha_2 - 2\alpha_1\alpha_3 + 2\alpha_1\alpha_4$$

- $$+ 9\alpha_2^2 + 2\alpha_2\alpha_3 - 2\alpha_2\alpha_4 + 9\alpha_3^2 - 2\alpha_3\alpha_4 + 9\alpha_4^2)$$

$$9\alpha_1 - \alpha_2 - \alpha_3 + \alpha_4 = 1 \quad \Rightarrow \quad \alpha_{o,1} = \alpha_{o,2} = \alpha_{o,3} = \alpha_{o,4} = \frac{1}{8}$$

$$-\alpha_1 + 9\alpha_2 + \alpha_3 - \alpha_4 = 1$$

$$-\alpha_1 + \alpha_2 + 9\alpha_3 - \alpha_4 = 1$$

$$\alpha_1 - \alpha_2 - \alpha_3 + 9\alpha_4 = 1$$

The result indicates all xi are support vectors

Optimal value of Q: $Q_o(\alpha) = \frac{1}{4}$

$$\frac{1}{2}\|\mathbf{w}_o\|^2 = \frac{1}{4} \Rightarrow \quad \|\mathbf{w}_o\| = \frac{1}{\sqrt{2}}$$

Solving XOR using SVM

$$\mathbf{w}_o = \frac{1}{8}[-\varphi(\mathbf{x}_1) + \varphi(\mathbf{x}_2) + \varphi(\mathbf{x}_3) - \varphi(\mathbf{x}_4)]$$

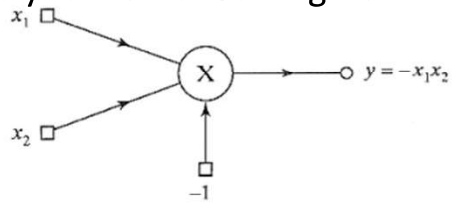
$$= \frac{1}{8}\left[-\begin{bmatrix}1\\1\\\sqrt{2}\\1\\-\sqrt{2}\\-\sqrt{2}\end{bmatrix} + \begin{bmatrix}1\\1\\-\sqrt{2}\\1\\-\sqrt{2}\\\sqrt{2}\end{bmatrix} + \begin{bmatrix}1\\1\\-\sqrt{2}\\1\\\sqrt{2}\\-\sqrt{2}\end{bmatrix} - \begin{bmatrix}1\\1\\\sqrt{2}\\1\\\sqrt{2}\\\sqrt{2}\end{bmatrix}\right]$$

$$= \begin{bmatrix}0\\0\\-1/\sqrt{2}\\0\\0\\0\end{bmatrix}$$

The first element of $\mathbf{w}_o$ indicates that the bias b is zero.

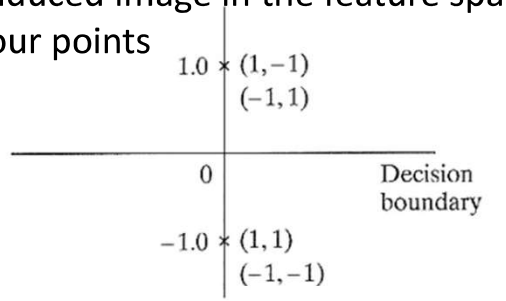
The optimal hyperplane is defined by $\mathbf{w}_o^T\boldsymbol{\varphi}(\mathbf{x}) = 0$

Polynomial for solving XOR



(a)

Induced image in the feature space due to four points



(b)

$$\left[0, 0, \frac{-1}{\sqrt{2}}, 0, 0, 0\right]\begin{bmatrix}1\\x_1^2\\\sqrt{2}x_1x_2\\x_2^2\\\sqrt{2}x_1\\\sqrt{2}x_2\end{bmatrix} = 0$$

which reduces to

$$-x_1x_2 = 0$$

The polynomial form of support vector machine for the XOR problem is as shown in Fig. 6.6a. For both $x_1 = x_2 = -1$ and $x_1 = x_2 = +1$, the output $y = -1$; and for both $x_1 = -1, x_2 = +1$ and $x_1 = +1$ and $x_2 = -1$, we have $y = +1$. Thus the XOR problem is solved.

SVM Algorithm➔

SVM Approach:

- we choose a kernel and then for given data,
- assemble the relevant quadratic problem and its constraints as matrices,
- pass them to the solver, which finds the decision boundary and necessary support vectors for us.
- These are then used to build a classifier for that training data

SVM Implementation

cvxopt, is a convex optimisation package that includes a wrapper for Python
Cvxopt has a nice and clean interface for the computational heavy lifting for an implementation of the SVM

- **Initialisation**
  - for the specified kernel, and kernel parameters, compute the kernel of distances between the datapoints
    - the main work here is the computation $\mathbf{K} = \mathbf{X}\mathbf{X}^T$
    - for the linear kernel, return $\mathbf{K}$, for the polynomial of degree d return $\frac{1}{\sigma}\mathbf{K}^d$
    - for the RBF kernel, compute $\mathbf{K} = \exp(-(\mathbf{x} - \mathbf{x}')^2/2\sigma^2)$
- **Training**
  - assemble the constraint set as matrices to solve:

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T t_i t_j \mathbf{K}\mathbf{x} + \mathbf{q}^T\mathbf{x} \text{ subject to } \mathbf{G}\mathbf{x} \leq \mathbf{h}, \mathbf{A}\mathbf{x} = b$$

  - pass these matrices to the solver
  - identify the support vectors as those that are within some specified distance of the closest point and dispose of the rest of the training data
  - compute b^* using equation (8.10)
- **Classification**
  - for the given test data $\mathbf{z}$, use the support vectors to classify the data for the relevant kernel using:
    - compute the inner product of the test data and the support vectors
    - perform the classification as $\sum_{i=1}^{n} \lambda_i t_i \mathbf{K}(\mathbf{x}_i, \mathbf{z}) + b^*$, returning either the label (hard classification) or the value (soft classification)

Implementation

- The first bit of computational work
- *m* is the number of datapoints and
- the solver, which has to factorise a
- iteration. Factorisation costs O(*m*3)
- to use for large datasets

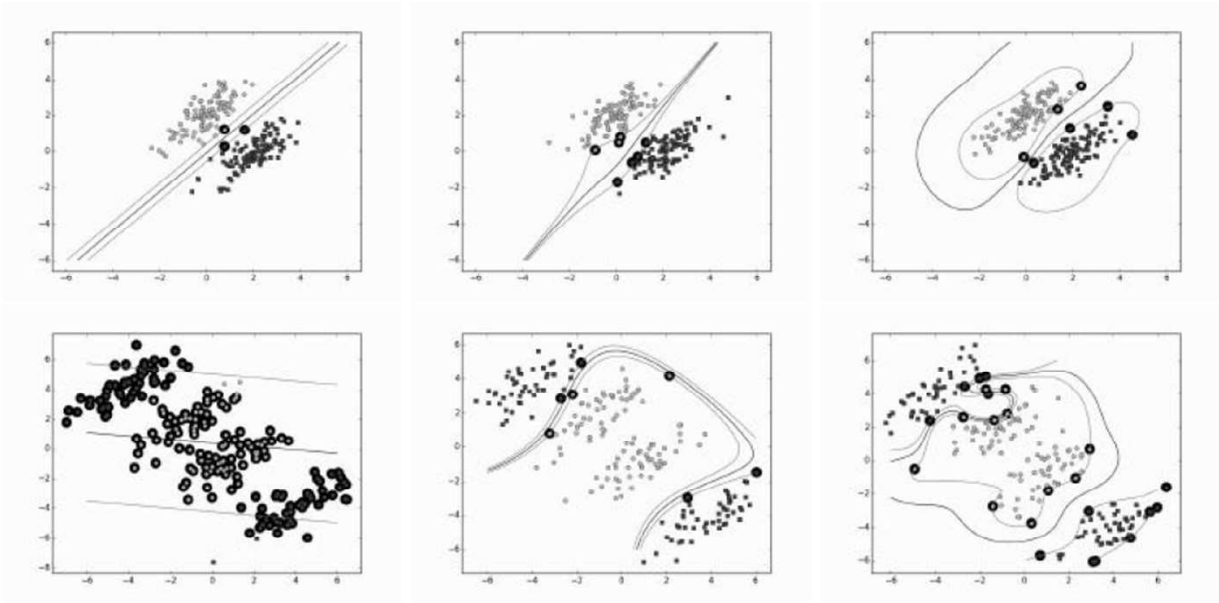


FIGURE 8.7 The SVM learning about a linearly separable dataset (*top row*) and a dataset that needs two straight lines to separate in 2D (*bottom row*) with *left* the linear kernel, *middle* the polynomial kernel of degree 3, and *right* the RBF kernel. $C = 0.1$ in all cases.

```
output = sv.classifier(Y,soft=False)
```

Python Implementation

```
# Assemble the matrices for the constraints
P = targets*targets.transpose()*self.K
q = -np.ones((self.N,1))
if self.C is None:
    G = -np.eye(self.N)
    h = np.zeros((self.N,1))
else:
    G = np.concatenate((np.eye(self.N),-np.eye(self.N)))
    h = np.concatenate((self.C*np.ones((self.N,1)),np.zeros((self.N,1))))
A = targets.reshape(1,self.N)
b = 0.0

# Call the quadratic solver
sol = cvxopt.solvers.qp(cvxopt.matrix(P),cvxopt.matrix(q),cvxopt.matrix(G),
cvxopt.matrix(h), cvxopt.matrix(A), cvxopt.matrix(b))

if self.kernel == 'poly':
    def classifier(Y,soft=False):
        K = (1. + 1./self.sigma*np.dot(Y,self.X.T))**self.degree

        self.y = np.zeros((np.shape(Y)[0],1))
        for j in range(np.shape(Y)[0]):
            for i in range(self.nsupport):
                self.y[j] += self.lambdas[i]*self.targets[i]*K[j,i]
            self.y[j] += self.b

        if soft:
            return self.y
        else:
            return np.sign(self.y)
```

Kernel Computaion

```
self.xsquared = (np.diag(self.K)*np.ones((1,self.N))).T
b = np.ones((self.N,1))
self.K -= 0.5*(np.dot(self.xsquared,b.T) + np.dot(b,self.xsquared.T))
self.K = np.exp(self.K/(2.*self.sigma**2))
```

```
elif self.kernel == 'rbf':
    def classifier(Y,soft=False):
        K = np.dot(Y,self.X.T)
        c = (1./self.sigma * np.sum(Y**2,axis=1)*np.ones((1,np.shape(Y)[0])))
        .T
        c = np.dot(c,np.ones((1,np.shape(K)[1])))
        aa = np.dot(self.xsquared[self.sv],np.ones((1,np.shape(K)[0]))).T
        K = K - 0.5*c - 0.5*aa
        K = np.exp(K/(2.*self.sigma**2))

        self.y = np.zeros((np.shape(Y)[0],1))
        for j in range(np.shape(Y)[0]):
            for i in range(self.nsupport):
                self.y[j] += self.lambdas[i]*self.targets[i]*K[j,i]

            self.y[j] += self.b

        if soft:
            return self.y
        else:
            return np.sign(self.y)
```

Identifying the Difference between the Kernels

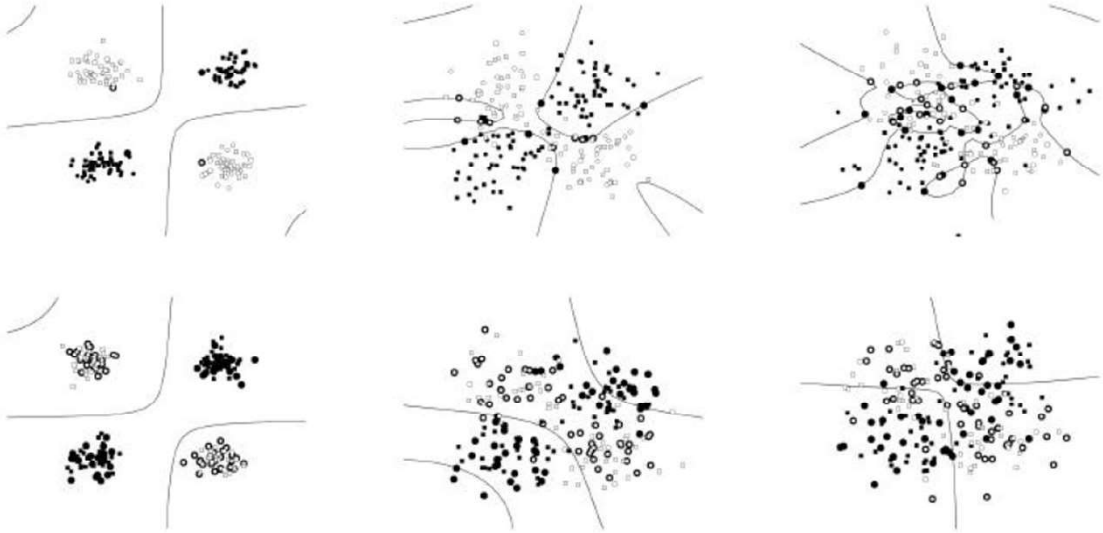


FIGURE 8.8 The effects of different kernels when learning a version of XOR with progressively more overlap (*left* to *right*) between the classes. *Top row:* polynomial kernel of degree 3 with no slack variables, *second row:* polynomial of degree 3 with $C = 0.1$, *third row:* RBF kernel, no slack variables, *bottom row:* RBF kernel with $C = 0.1$. The support vectors are highlighted, and the decision boundary is drawn for each case.

- It can be seen that where the classes start to overlap, the inclusion of slack variables leads toto far simpler decision boundaries and a better model of the underlying data. Both the polynomial and RBF kernels perform well on this problem.

EXTENSIONS TO THE SVM

MULTI-CLASS CLASSIFICATION

- The SVM only works for two classes.
- For the problem of ***N*-class classification**, train an SVM that learns to classify class one from all other classes, then another that classifies class two from all the others. So for *N*-classes, we have *N* SVMs.

➔**Problem**: how do we decide which of these SVMs is the one that recognises the particular input?

- **Solution:**
  - Choose the one that makes the strongest prediction, that is, the one where the basis vector input point is the furthest into the positive class region.
  - It might not be clear how to work out which is the strongest prediction.
  - The classifier examples in the code snippets return either the class label (as the sign of *y*) or the value of *y*, and this value of *y* is telling us how far away from the decision boundary it is, and clearly it will be negative if it is a misclassification.
  - Therefore use the **maximum value of this soft boundary** as the best classifier.

```
output = np.zeros((np.shape(test)[0],3))
output[:,0] = svm0.classifier(test[:,:2],soft=True).T
output[:,1] = svm1.classifier(test[:,:2],soft=True).T
output[:,2] = svm2.classifier(test[:,:2],soft=True).T

# Make a decision about which class
# Pick the one with the largest margin
bestclass = np.argmax(output,axis=1)
err = np.where(bestclass!=target)[0]
print len(err)/ np.shape(target)[0]
```

Multi Class SVM

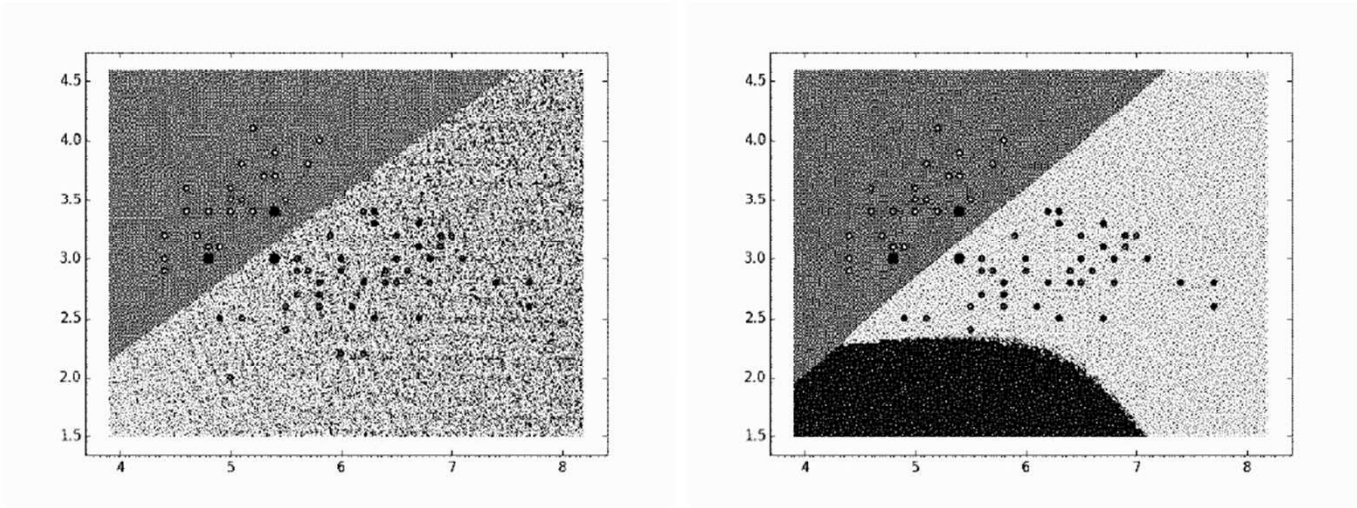


FIGURE 8.9 A linear (*left*) and polynomial, degree 3 (*right*) kernel learning the first two dimensions of the iris dataset, which separates one class very well from the other two, but cannot distinguish between the other two (for good reason). The support vectors are highlighted.

Support Vector Machine Regression

- Support Vector Regression is a supervised learning algorithm that is used to predict discrete values.
- Support Vector Regression uses the same principle as the SVMs.
- The basic idea behind SVR is to find the best fit line.
- In SVR, the best fit line is the hyperplane that has the maximum number of points.
- Unlike other Regression models that try to minimize the error between the real and predicted value, the SVR tries to fit the best line within a threshold value.
- The threshold value is the distance between the hyperplane and boundary line.
- The fit time complexity of SVR is more than quadratic with the number of samples which makes it hard to scale to datasets with more than a couple of 10000 samples.

SVM REGRESSION

- The key is to take the usual least-squares error function (with the regulariser that keeps the norm of the weights small):

$$\frac{1}{2}\sum_{i=1}^{N}(t_i - y_i)^2 + \frac{1}{2}\lambda\|\mathbf{w}\|^2$$

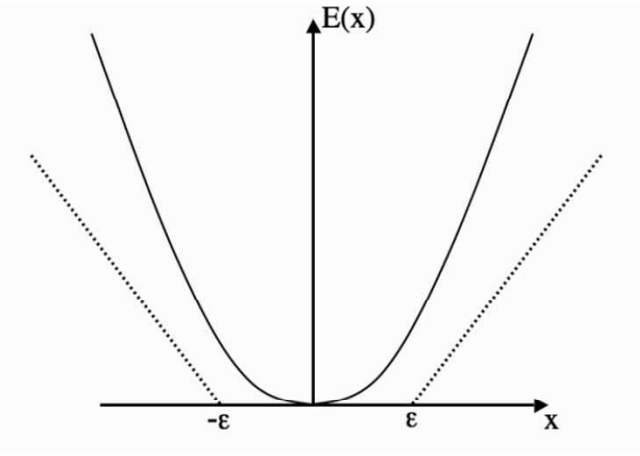
- and transform it using what is known as an -insensitive error function (*E*) that gives 0 if the difference between the target and output is less thanε (and subtracts in any other case for consistency).
- Reason ➔a small number of support vectors. so we are only interested in the points that are not well predicted

$$\sum_{i=1}^{N} E_\epsilon(t_i - y_i) + \lambda\frac{1}{2}\|\mathbf{w}\|^2$$

- the predictions to be inside the tube of radius that surrounds the correct line.
- To allow for errors, slack variables are introduced for each datapoint (*i* for datapoint *i*) with their constraints and follow the same procedure of introducing Lagrange multipliers, transferring to the dual problem, using a kernel function and solving the problem with a quadratic solver.
- The upshot of all this is that the prediction we make for test point **z** is

$$f(\mathbf{z}) = \sum_{i=1}^{n}(\mu_i - \lambda_i K(\mathbf{x}_i, \mathbf{z}) + b)$$

where μ_i and λ_i are two sets of constraint variables.



The ϵ-insensitive error function is zero for any error below ϵ.