IMAGE DATA ANALYSIS (6CFU)

MODULE OF

REMOTE SENSING

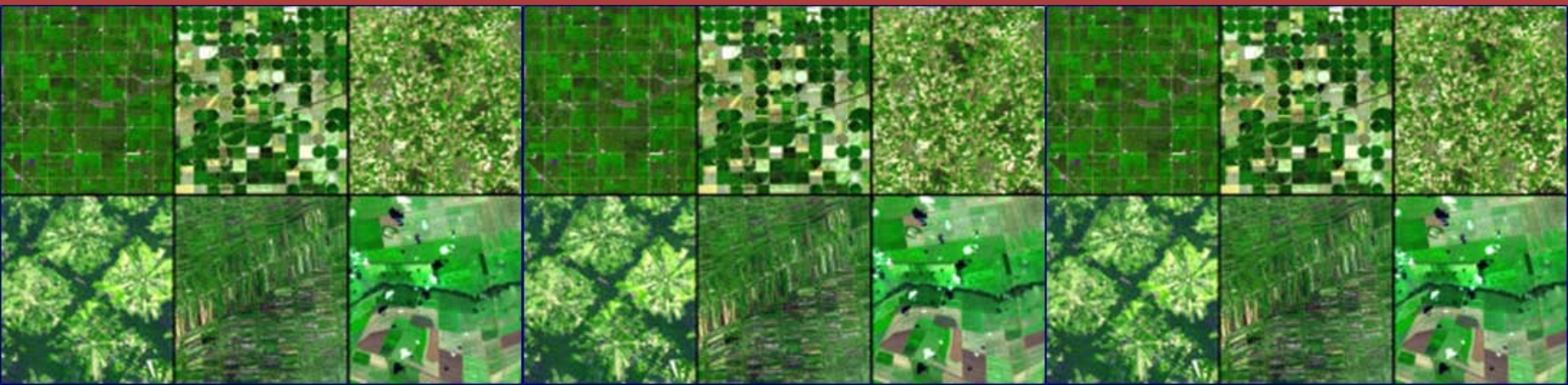
(9 CFU)

A.Y. 2022/23

MASTER OF SCIENCE IN COMMUNICATION TECHNOLOGIES AND MULTIMEDIA
MASTER OF SCIENCE IN COMPUTER SCIENCE, LM INGEGNERIA INFORMATICA

PROF. ALBERTO SIGNORONI

SUPERVISED NON-PARAMETRIC CLASSIFICATION: GEOMETRIC APPROACHES



Introduction

- Statistical classification algorithms are among the most commonly encountered labeling techniques used in remote sensing
  - One of the valuable aspects of a statistical approach is that a *set* of relative likelihoods is produced.
  - Even though, in the majority of cases, the *maximum* of the likelihoods is chosen to indicate the most probable label for a sample (e.g. pixel), *there remains nevertheless information in the remaining likelihoods that could be made use of in some circumstances*:
    - either to initiate a process such as relaxation labelling or MRF contextualization
    - or simply to provide the user with some feeling for the other likely classes.
  - Those situations are however not common and, in most applications, the maximum selection is made.

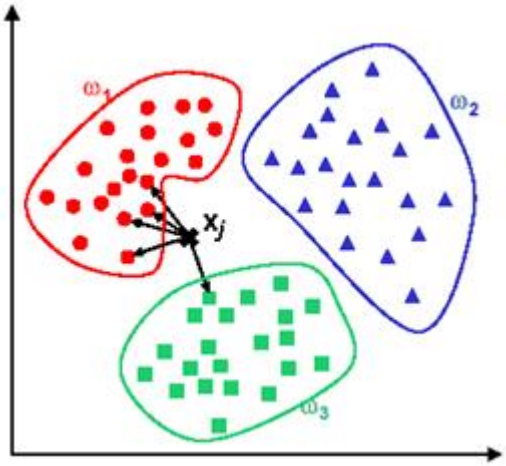
- That being so, the material presented about *decision surfaces* shows that the decision process has a geometric counterpart
  - in that a comparison of statistically derived discriminant functions leads equivalently to a *decision rule* that allows a pixel to be *classified on the basis of its* ***position*** in multispectral space compared with the location of a decision surface.
- This leads us to question whether a geometric interpretation can be adopted in general, without needing first to use statistical models.

THE *K*NN (NEAREST NEIGHBOUR) CLASSIFIER

The *kNN* (Nearest Neighbour) Classifier

- A first classifier that is particularly simple in concept, but can be time consuming to apply, is the *k*-Nearest Neighbour classifier.
  - It assumes that pixels close to each other in feature space are likely to belong to the same class.
  - In its simplest form, an unknown pixel is labelled by examining the available training pixels in multispectral space and choosing the class most represented among a pre-specified number *k* of nearest neighbours.
  - The comparison essentially requires to compute the distances from the unknown pixel to all training pixels to be computed (*computer point of view*).
  - In principle (*human point of view*), the kNN query starts at the test point **x** and grows a spherical region until it encloses *k* training samples and it labels the test point by a majority vote of these samples.

- **Example**: we have three classes and the goal is to find a class label for the unknown example $\mathbf{x}_j$. In this case we use the Euclidean distance and a value of *k*=5 neighbors. Of the 5 closest neighbors, 4 belong to ω_1 and 1 belongs to ω_3, so $\mathbf{x}_j$ is assigned to ω_1, the predominant class.



The *kNN* (Nearest Neighbour) Classifier

Suppose there are k_i neighbours labelled as class ω_i out of k nearest neighbours for a pixel vector $\boldsymbol{x}$, noting that $\sum_{i=1}^{M} k_i = k$ where M is the total number of classes.

In the basic *kNN* rule we define the discriminant function for the ith class as

$$g_i(\boldsymbol{x}) = k_i$$

and the decision rule is:

$$\boldsymbol{x} \in \omega_i, \qquad \text{if} \qquad g_i(\boldsymbol{x}) > g_j(\boldsymbol{x}) \qquad \text{for all} \qquad j \neq i$$

The basic rule does not take the distance of each neighbour to the current pixel vector into account and may lead to tied results. An improvement is to distance-weight the discriminant function:

$$g_i(\boldsymbol{x}) = \frac{\sum_{j=1}^{k_i} 1/\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i^j)}{\sum_{i=1}^{M} \sum_{j=1}^{k_i} 1/\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i^j)}$$

where $\mathrm{d}(\boldsymbol{x}, \boldsymbol{x}_i^j)$ is the spectral distance (commonly Euclidean) between the unknown pixel vector $\boldsymbol{x}$ and its neighbour $\boldsymbol{x}_i^j$, the jth of the k_i pixels in class ω_i.

The *kNN* (Nearest Neighbour) Classifier

If the training data for each class is not in proportion to its respective population, $p(\omega_i)$, in the image, a Bayesian Nearest-Neighbour rule can be used:

$$g_i(\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\omega_i)\,p(\omega_i)}{\sum_{j=1}^{M} p(\boldsymbol{x}|\omega_j)\,p(\omega_j)} = \frac{k_i\,p(\omega_i)}{\sum_{j=1}^{M} k_j\,p(\omega_j)}$$

In the *k*NN algorithm as many spectral distances as there are training pixels must be evaluated for each unknown pixel to be labelled. That requires an impractically high computational load, particularly when the number of spectral bands and/or the number of training samples is large. The method is not well-suited therefore to hyperspectral datasets, although it is possible to improve the efficiency of the distance search process.

Using an **appropriate nearest neighbor search** algorithm makes *k*-NN computationally tractable even for large data sets:

- partial distance (discard data which have an already high reduced dimensionality distance)
- use of precomputed searching trees → k-d trees (however not exact!)
- to learn more: see for example this video series

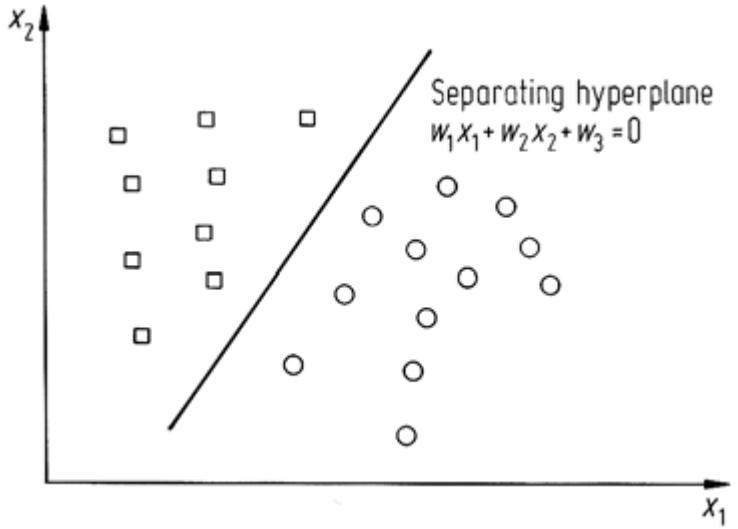
NON-PARAMETRIC METHODS FROM A GEOMETRIC BASIS: LINEAR DISCRIMINATION

Concept of a Weight Vector

- Consider the simple two class multispectral space shown in Figure, which has been constructed *intentionally* so that a simple straight line can be drawn between the pixels as shown.
- This straight line, which will be a multidimensional linear surface in general and which is called a ***hyperplane***, can function as a ***decision surface*** for classification.
- In the two dimensions shown, the equation of the line can be expressed

$$w_1 x_1 + w_2 x_2 + w_3 = 0$$

where the x_i are the *brightness value* co-ordinates of the multispectral space and the w_i are a set of coefficients, usually called ***weights***.



Concept of a Weight Vector

- There will be as many weights as the number of channels in the data, *plus one*.
- In general, if the number of channels or bands is *N*, the equation of a linear surface is

$$w_1x_1 + w_2x_2 + \ldots + w_Nx_N + w_{N+1} = 0$$

which can be written as

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + w_{N+1} = 0 \qquad (8.22)$$

where $\boldsymbol{x}$ is the *co-ordinate vector* and $\boldsymbol{w}$ is called the *weight vector*.

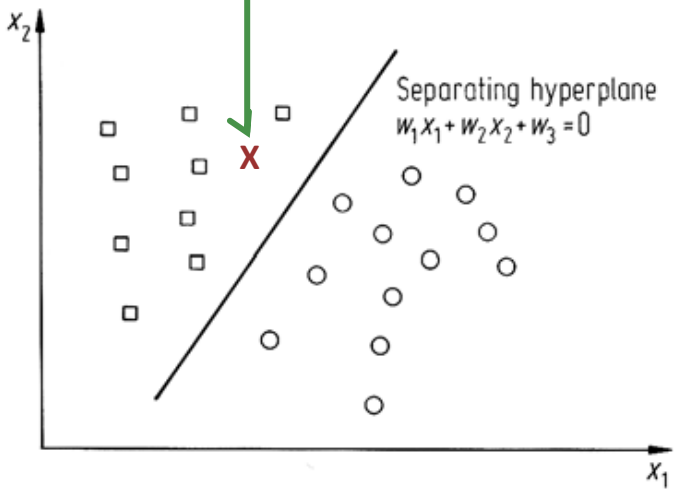
- In a real exercise the position of the separating surface would be unknown initially.
- Training a linear classifier amounts to determining an appropriate set of the weights that places the decision surface between the two sets of training samples.

- There is not necessarily a unique solution – any of an infinite number of (marginally different) decision hyperplanes will suffice to separate the two classes.
  - For a given data set, an explicit equation for the separating surface can be obtained using the minimum distance rule, as seen in minimum distance classification, which entails finding the mean vectors of the two class distributions.
  - An alternative method is outlined in the following, based on selecting an arbitrary surface and then iterating it into an acceptable position.
  - Even though not often used anymore, this method is useful to consider since it establishes some of the concepts used in *neural networks* and *support vector machines,* as we will see.

Testing Class Membership

- The calculation in (8.22) will be exactly zero only for values of $\boldsymbol{x}$ lying on the hyperplane (decision surface).
- If we substitute into that equation values of $\boldsymbol{x}$ corresponding to the pixel points indicated in the previous Figure, the left hand side will be non-zero.
- *For pixels in one class a positive result will be given, while pixels on the other side will give a negative result.*
- Thus, once the decision surface (i.e. the weights $\boldsymbol{w}$) has been identified (i.e. trained), then a **decision rule is**

$$\begin{aligned} \boldsymbol{x} &\in \text{ class 1 if } \boldsymbol{w}^t\boldsymbol{x} + w_{N+1} > 0 \\ \boldsymbol{x} &\in \text{ class 2 if } \boldsymbol{w}^t\boldsymbol{x} + w_{N+1} < 0 \end{aligned} \tag{8.23}$$



Training a linear classifier in the weight space

- A full discussion of linear classifier training is given in Nilsson (1965, 1990); only those aspects helpful to the neural network development following are treated here.

  It is expedient to define a new, augmented pixel vector according to

  $$\boldsymbol{y} = [\boldsymbol{x}^t, 1]^t$$

- If, in (8.22), we also take the term w_{N+1} into the definition of the weight vector, viz.

  $$\boldsymbol{w} = [\boldsymbol{w}^t, w_{N+1}]^t$$

  then the equation of the decision surface, can be expressed more compactly as

  $$\boldsymbol{w}^t \boldsymbol{y} = 0 \quad \text{(or equivalently } \boldsymbol{w} \cdot \boldsymbol{y} = 0\text{)}$$

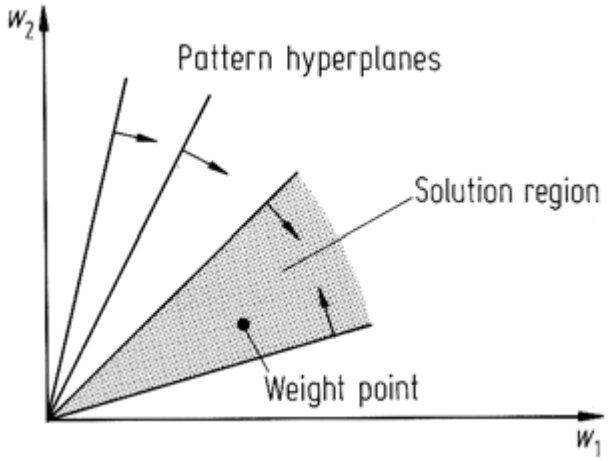
  so that the decision rule of (8.23) can be restated

  $$\begin{aligned} \boldsymbol{x} &\in \text{ class 1 if } \boldsymbol{w}^t \boldsymbol{y} > 0 \\ \boldsymbol{x} &\in \text{ class 2 if } \boldsymbol{w}^t \boldsymbol{y} < 0 \end{aligned} \tag{8.24}$$

- We usually think of $\boldsymbol{w}^t \boldsymbol{y} = 0$ as defining a linear surface in the $\boldsymbol{x}$ (or now $\boldsymbol{y}$) multispectral space, in which the coefficients of the variables (y_1, y_2, etc.) are the weights w_1, w_2, etc. However it is also possible to think of the equation as describing a linear surface in which the y's are the coefficients and the w's are the variables. This interpretation will see these surfaces plotted in a co-ordinate system which has axes w_1, w_2, etc.

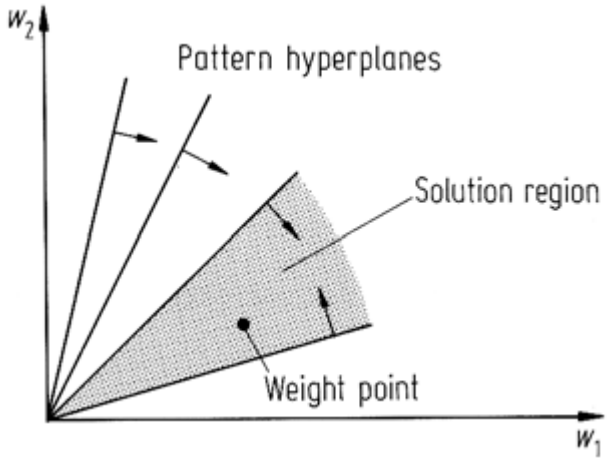
Training a linear classifier in the weight space

- A two-dimensional version of this **weight space**, as it is called, is shown in Figure, in which have been plotted a number of pattern hyperplanes; these are specific linear surfaces in the new co-ordinates that pass through the origin and have, as their coefficients, the components of the (augmented) pixel vectors. Thus, while the pixels plot as points in multispectral space, they plot as linear surfaces in weight space.
- Likewise, a set of weight coefficients will define a surface in multispectral space, but will plot as a point in weight space. Although this is an abstract concept it will serve to facilitate an understanding of how a linear classifier can be trained.



Training a linear classifier in the weight space

- In weight space the decision rule of (8.24) still applies – however *now it tests that the weight point is on the appropriate side of the pattern hyperplane*.
  - For example, Figure shows a single weight point which lies on the correct side of each pixel and thus defines a *suitable decision surface in multispectral space*. In the diagram, small arrows are attached to each pixel hyperplane to indicate the side on which the weight point must lie for that pixel in order to make easy to find the region where the test of (8.24) succeeds for all pixels.
- The purpose of training the linear classifier is to ensure that the weight point is located somewhere within the **solution region**. If, through some **initial guess**, the weight point is located somewhere else in weight space then it ***has to be moved*** to the solution region.



The aim in using this dual space is to directly see the **region** of possible compatible set of weights. This region is rapidly deducible from the training set.

Training a linear classifier in the weight space

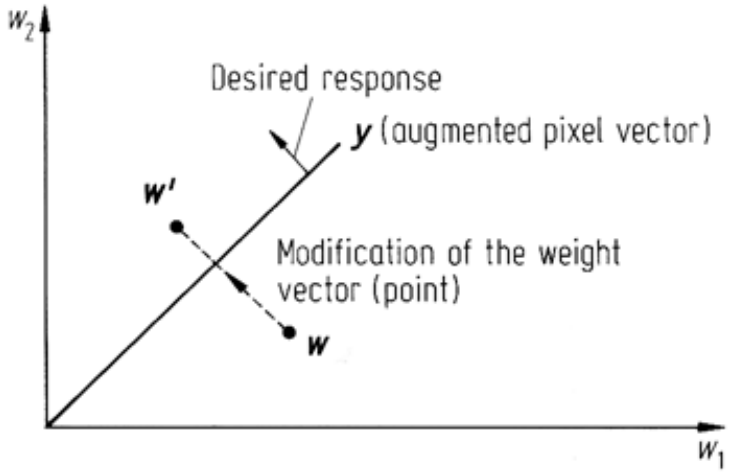
- Suppose an initial guess is made for the weight vector $\boldsymbol{w}$, but that this places the weight point on the wrong side of a particular pixel hyperplane as illustrated.
  - Clearly, the weight point *has to be shifted* to the other side to give a correct response in (8.24).
  - The most direct manner in which the weight point can be modified is to *move it straight across the pixel hyperplane*.
  - This can be achieved by adding a **scaled amount of the pixel vector** to the weight vector.

- The new position of the weight point is then

$$\boldsymbol{w}' = \boldsymbol{w} + c\boldsymbol{y} \qquad (8.25)$$

where c is called the **correction increment**, the size of which determines by how much the original weight point is moved in the direction $\boldsymbol{y}$ which is **orthogonal** to the pixel hyperplane. (*)

  - If it is large enough the weight point will be shifted right across the pixel plane, as required.



(*) Note that $\boldsymbol{w}^t \boldsymbol{y} = \boldsymbol{0}$ means that $\boldsymbol{w} \perp \boldsymbol{y}$

Training a linear classifier in the weight space

- Having so modified the weight vector, the product in (8.24) then becomes

$$\begin{aligned} \boldsymbol{w}'^t \boldsymbol{y} &= \boldsymbol{w}^t \boldsymbol{y} + c\,\boldsymbol{y}^t \boldsymbol{y} \\ &= \boldsymbol{w}^t \boldsymbol{y} + c|\boldsymbol{y}|^2 \end{aligned}$$

  Clearly, if the initial $\boldsymbol{w}^t \boldsymbol{y}$ was erroneously negative a suitable positive value of c will give a positive value of $\boldsymbol{w}'^t \boldsymbol{y}$; otherwise a negative value of c will correct an erroneous initial positive value of the product.

- Using the class membership test in (8.24) and the correction formula of (8.25) the following iterative nonparametric training procedure, referred to as *error correction feedback*, is adopted.
  - First, an initial position for the weight point is chosen arbitrarily. Then, pixel vectors from training sets are presented one at a time. If the current weight point position classifies a pixel correctly then no action need be taken; otherwise the weight vector is modified as in (8.25) with respect to that particular pixel vector. This procedure is repeated for each pixel in the training set, and the set is scanned as many times as necessary to move the weight point into the solution region. If the classes are linearly separable then such a solution will be found.

Setting the Correction Increment

- Several approaches can be adopted for choosing the value of the correction increment, c. The simplest is to set c equal to a positive or negative constant (according to the change required in the $\boldsymbol{w}^t \boldsymbol{y}$ product). A common choice is to make $c = \pm 1$ so that application of (8.25) amounts simply to adding the augmented pixel vector to or subtracting it from the weight vector, thereby obviating multiplications and giving fast training.
- Another rule is to choose the correction increment proportional to the difference between the desired and actual response of the classifier:

$$c = \eta(t - \boldsymbol{w}^t \boldsymbol{y})$$

so that (8.25) can be written

$$\boldsymbol{w}' = \boldsymbol{w} + \Delta \boldsymbol{w}$$

with

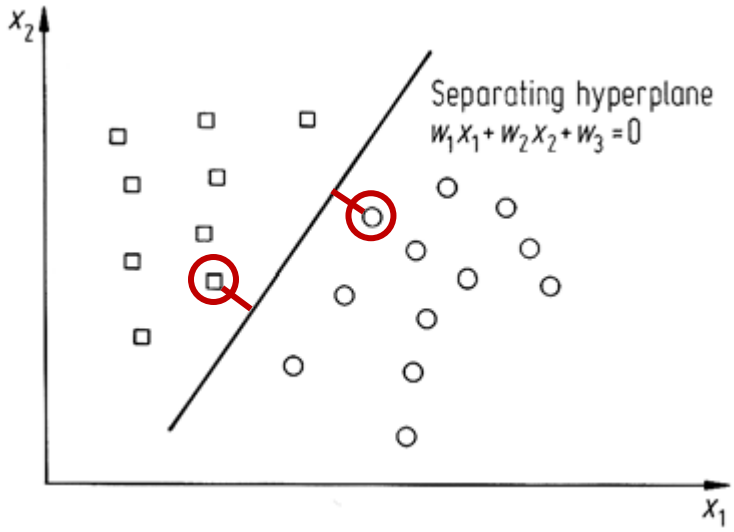
$$\Delta \boldsymbol{w} = \eta(t - \boldsymbol{w}^t \boldsymbol{y})\boldsymbol{y} \tag{8.26}$$

where t is the desired response to the training pattern $\boldsymbol{y}$ and $\boldsymbol{w}^t \boldsymbol{y}$ is the actual response; η is a factor which controls the degree of correction applied. Usually t would be chosen as $+1$ for one class and -1 for the other.

SUPPORT VECTOR CLASSIFIERS

Linearly Separable Classes

- **The training process outlined for *linear discrimination* can lead to many, non-unique, yet acceptable solutions for the weight vector.**
  - The actual *size of the solution region in the weight space* (as seen) is an indication of that.
  - Also, *every training pixel takes part in the training process*; yet examination of the vector space suggests that it is **only** those *pixels in the vicinity of the separating hyperplane* that define where the hyperplane needs to lie in order to give a reliable classification.
- The **support vector machine (SVM)** provides a *training approach that depends only on those pixels in the vicinity of the separating hyperplane* (called the ***support pixel vectors***).
  - It also leads to a hyperplane position that is in a sense **optimal** for the available training patterns, as will be seen shortly.
  - SVM was invented by Vladimir Vapnik (1992)
  - The support vector concept was introduced to remote sensing image classification by Gualtieri and Cromp (1998). Two recent reviews that contain more detail than is given in the following treatment are by Burges (1998) and Huang et al. (2002). A good treatment from a remote sensing perspective has been given by Melgani and Bruzzone (2004).



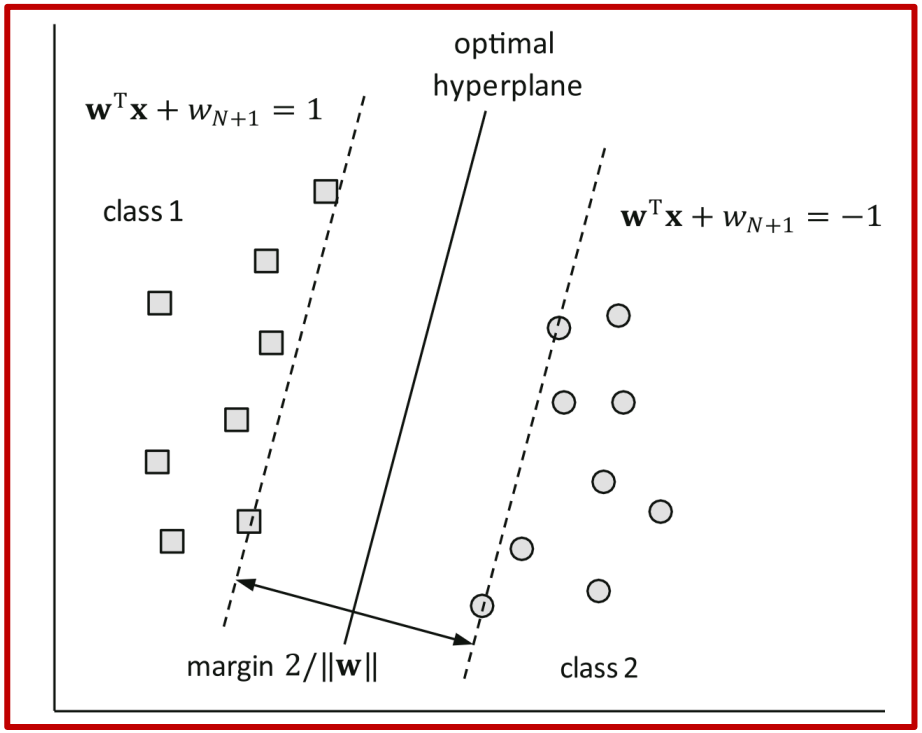
Linearly Separable Classes

- If we expand the region in the vicinity of the hyperplane (see previous Figure) we can see that the optimal orientation of the hyperplane is when there is a **maximum separation** between the patterns in the two classes (see Figure below).
  - We can then draw two further hyperplanes **parallel** to the separating hyperplane, as shown, bordering the nearest training pixels from the two classes.
  - The equations for the hyperplanes are shown in the figure. Note that the *choice of unity* on the right hand side of the equations for the two marginal hyperplanes is *arbitrary* (related to a *common scale factor*), but it helps in the analysis.
  - If it were otherwise it could be scaled to unity by appropriately *scaling* the weighting coefficients w_k.
- Note that for pixels that lie *beyond the marginal hyperplanes*, we have
  - for class 1 pixels

    $$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + w_{N+1} \geq 1 \qquad (8.28a)$$

  - for class 2 pixels

    $$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + w_{N+1} \leq -1 \qquad (8.28b)$$



Linearly Separable Classes

- It is useful now to describe the class label of the ith pixel by the variable y_i, which takes the value $+1$ for class 1 and -1 for class 2 pixels. Equations (8.28a) and (8.28b) can then be written as a single expression valid for pixels from both classes:

$$(\boldsymbol{w}.\boldsymbol{x} + w_{N+1})y_i \geq 1 \text{ for pixel } i \text{ in its correct class.}$$

- Alternatively

$$(\boldsymbol{w}.\boldsymbol{x} + w_{N+1})y_i - 1 \geq 0 \tag{8.29}$$

Equation (8.29) must hold for all pixels if the data is linearly separated by the two marginal hyperplanes of Figure. Those hyperplanes, defined by the equalities in (8.28), are described by

$$\boldsymbol{w}.\boldsymbol{x} + w_{N+1} - 1 = 0$$
$$\boldsymbol{w}.\boldsymbol{x} + w_{N+1} + 1 = 0$$

The perpendicular distances of these hyperplanes from the origin, respectively, are $-(w_{N+1} - 1)/||\boldsymbol{w}||$ and $-(w_{N+1} + 1)/||\boldsymbol{w}||$, where $\|\boldsymbol{w}\|$ is the Euclidean length of the weight vector. Therefore, the distance between the two hyperplanes, which is the *margin* in Figure, is $2/\|\boldsymbol{w}\|$.

Linearly Separable Classes

- The best position (orientation) for the separating hyperplane will be that for which $2/\|\boldsymbol{w}\|$ is a maximum, or equivalently when the magnitude of the weight vector, $\|\boldsymbol{w}\|$, is a minimum. However there is a constraint! As we seek to maximise the margin between the two marginal hyperplanes by minimising $\|\boldsymbol{w}\|$ we must not allow (8.29) to be invalidated. In other words, all the training pixels must be on their correct side of the marginal hyperplanes. We handle the process of minimising $\|\boldsymbol{w}\|$ subject to that constraint by the process known as Lagrange multipliers. This requires us to set up a function (called the Lagrangrian) which includes the expression to be minimised ($\|\boldsymbol{w}\|$) from which is subtracted a proportion (α_i) of each constraint (one for each training pixel) in the following manner:

$$L = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_i \alpha_i \{y_i(\boldsymbol{w}.\boldsymbol{x}_i + w_{N+1}) - 1\} \qquad (8.30)$$

- The α_i are called Lagrange multipliers and are positive by definition, i.e.

$$\alpha_i \geq 0 \text{ for all } i.$$

Linearly Separable Classes

- By minimising L we minimise $\|\boldsymbol{w}\|$ subject to the constraint (8.29).

  In (8.30) it is convenient to substitute

$$f(\boldsymbol{x}_i) = (\boldsymbol{w}.\boldsymbol{x}_i + w_{N+1})y_i - 1$$

  to give

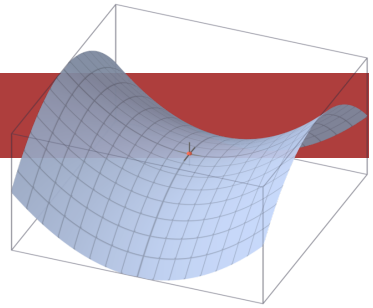
$$L = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_i \alpha_i f(x_i)$$

  noting that for pixels in their correct class $f(\boldsymbol{x}_i) \geq 0$.

- It is useful here to remember what our task is. We have to find the *most* separated marginal hyperplanes (see Fig.). In other words we need to find the $\boldsymbol{w}$ and w_{N+1} that minimises L and thus maximises the *margin* shown in the figure.

  But in seeking to minimise L (essentially during the training process) how do we treat the α_i? Suppose (8.29) is violated, as could happen for some pixels during training; then $f(\boldsymbol{x}_i)$ will be negative. Noting that α_i is positive that would cause L to increase. But we need to find values for $\boldsymbol{w}$ and w_{N+1} such that L is minimised.

Linearly Separable Classes



- The worst possible case to handle is when the α_i are such as to cause L to be a maximum, since that forces us to minimise L with respect to $\boldsymbol{w}$ and w_{N+1} while the α_i are trying to make it as large as possible. The most robust approach to finding $\boldsymbol{w}$ and w_{N+1} (and thus the hyperplanes) therefore is to find the values of $\boldsymbol{w}$ and w_{N+1} that minimise L while simultaneously finding the α_i that try to maximise it.
- Thus we require, first, that:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_i \alpha_i y_i \mathbf{x}_i = 0$$

so that $\boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x}_i$. (8.31a)

- Secondly we require:

$$\frac{\partial L}{\partial w_{N+1}} = -\sum_i \alpha_i y_i = 0$$

so that $\sum_i \alpha_i y_i = 0$. (8.31b)

Linearly Separable Classes

- Before proceeding, examine (8.30) again, this time for training pixels that satisfy the requirement of (8.29).
- What value(s) of α_i in (8.30) for those pixels maximise L?
- Since $y_i\,(\boldsymbol{w}.\boldsymbol{x}_i + w_{N+1}) - 1$ is now always positive then the only (non-negative) value of α_i that makes L as big as possible is $\alpha_i = 0$.
- Therefore, for any training pixels on the correct side of the marginal hyperplanes, $\alpha_i = 0$.
- This is an amazing, yet intuitive, result. It says we do not have to use any of the training pixel vectors, other than those that reside exactly on one of the marginal hyperplanes.
- The latter are called ***support vectors*** since they are the only ones that support the process of finding the marginal hyperplanes.
- Thus, in applying (8.31a) to find $\boldsymbol{w}$ **we only have to use those pixels on the marginal hyperplanes** (which can change as the plane moves).
- Therefore the training is not yet finished! We still have to find the relevant α_i (i.e. those that maximise L and are non-zero).

Linearly Separable Classes

- To proceed, note that we can put $\|\boldsymbol{w}\| = \boldsymbol{w}.\boldsymbol{w}$ in (8.30). Now (8.30), along with (8.31a), can be written most generally as:

$$
\begin{aligned}
L = \frac{1}{2}\left(\sum_i \alpha_i y_i \boldsymbol{x}_i\right) \cdot \left(\sum_j \alpha_j y_j \boldsymbol{x}_j\right) \\
- \sum_i \alpha_i \left[y_i \left(\left(\sum_j \alpha_j y_j \boldsymbol{x}_j \right) . \boldsymbol{x}_i + w_{N+1} \right) - 1 \right] \\
= \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i . \boldsymbol{x}_j - \sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i . \boldsymbol{x}_j - w_{N+1} \sum_i \alpha_i y_i + \sum_i \alpha_i
\end{aligned}
$$

- Using (8.31b) this simplifies to

$$
L = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i . \boldsymbol{x}_j \tag{8.32}
$$

which has to be maximised by the choice of α_i. This usually requires a numerical procedure to solve for any real problem. Once we have found the α_i – call them α_i^o – we can substitute them into (8.31a) to give the optimal training vector:

$$
\boldsymbol{w}^o = \sum_i \alpha_i^o y_i \boldsymbol{x}_i \tag{8.33a}
$$

Linearly Separable Classes

- But we still do not have a value for w_{N+1}. Recall that on a marginal hyperplane

$$(\boldsymbol{w}.\boldsymbol{x}_i + w_{N+1})y_i - 1 = 0 \ .$$

Choose two support (training) vectors $\boldsymbol{x}(1)$ and $\boldsymbol{x}(-1)$ on each of the two marginal hyperplanes respectively for which $y = 1$ and -1. For these vectors we have

$$\boldsymbol{w}.\boldsymbol{x}(1) + w_{N+1} - 1 = 0$$

and

$$-\boldsymbol{w}.\boldsymbol{x}(-1) - w_{N+1} - 1 = 0$$

so that

$$w_{N+1} = \frac{1}{2}(\boldsymbol{w}.\boldsymbol{x}(1) + \boldsymbol{w}.\boldsymbol{x}(-1)) \ . \qquad (8.33b)$$

Normally sets of $\boldsymbol{x}(1), \boldsymbol{x}(-1)$, would be used, with w_{N+1} found by averaging.

- With the values of α_i^o determined by numerical optimisation, (8.33a,b) now give the parameters of the separating hyperplane that provides the largest margin between the two sets of training data. In terms of the training data, the equation of the hyperplane is:

$$\boldsymbol{w}^\circ.\boldsymbol{x} + w_{N+1} = 0$$

so that the discriminant function, for an unknown pixel $\boldsymbol{x}$ is

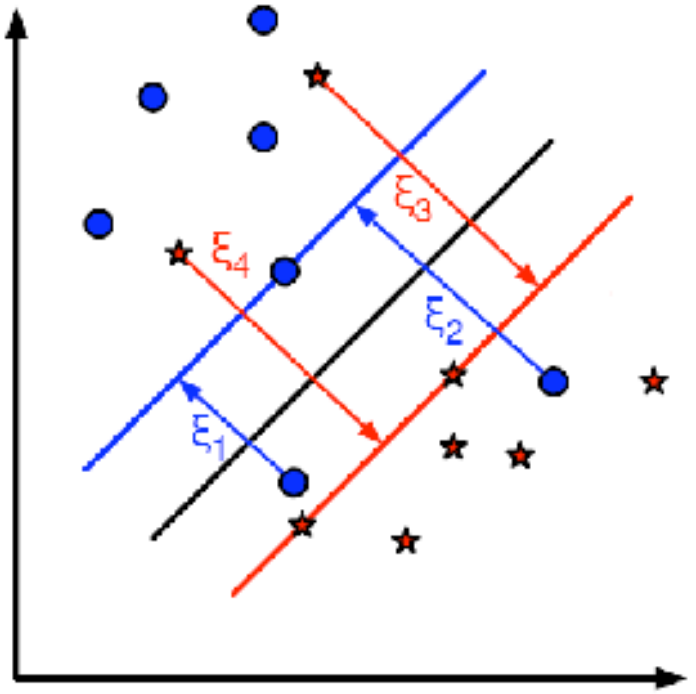
$$\boxed{g(\boldsymbol{x}) = \text{sgn}\ (\boldsymbol{w}^\circ.\boldsymbol{x} + w_{N+1})} \qquad (8.34)$$

Overlapping Classes – The Use of Slack Variables

- It is unrealistic to expect that the pixel vectors from two ground cover classes will be completely separated. Instead, there is likely to be class overlap (see Figure).
  - Any classifier algorithm, to be effective, must be able to cope with such a situation by generating the best possible discrimination between the classes in the circumstances.
  - As it has been developed previously, the support vector classifier will not find a solution for overlapping classes and requires modification.

- That is done by relaxing the requirement on finding a maximum margin solution
  - by agreeing that such a goal is not possible for all training pixels
  - by agreeing that we will have to accept that some will not be correctly separated during training.
- Such a situation is accommodated by introducing a degree of ‘‘slackness’’ in the training step, by the introduction of

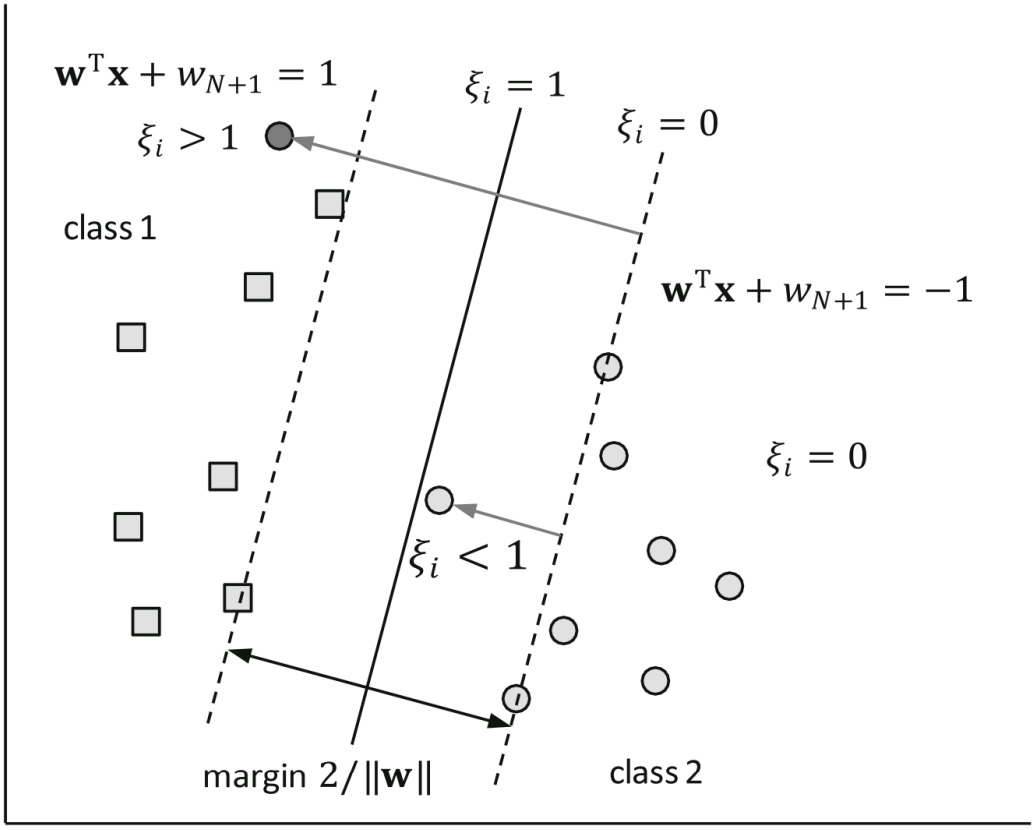
slack variables ξ_i



Overlapping Classes – The Use of Slack Variables

- We introduce a set of positive ‘‘slack variables’’ ξ_i, one for each of the training patterns, which are used to modify the constraint of (8.29) such that it now becomes

$$(\mathbf{w}^{\mathrm{T}}\mathbf{x}_i + w_{N+1})y_i \geq 1 - \xi_i \ \forall i \qquad (8.43a)$$



Overlapping Classes – The Use of Slack Variables

- The slack variables are defined such that:

$\xi_i = 0$	for training pixels that are on or on the correct side of the marginal hyperplane
$\xi_i = 1$	for a pixel on the separating hyperplane—the decision boundary—because $\mathbf{w}^{\mathrm{T}}\mathbf{x}_i + w_{N+1} = 0$ and $\lvert y_i \rvert = 1$
$\xi_i > 1$	for pixels that are on the wrong side of the separating hyperplane since $\mathbf{w}^{\mathrm{T}}\mathbf{x}_i + w_{N+1}$ has the opposite sign to y_i for misclassified pixels
$\xi_i = \lvert y_i - (\mathbf{w}^{\mathrm{T}}\mathbf{x}_i + w_{N+1}) \rvert$	for all other training pixels

- When training the support vector machine with overlapping data we minimise the number of pixels in error while maximising the margin by minimising $\|\mathbf{w}\|$.
- A measure of the number of pixels in error is the sum of the slack variables over all the training pixels; they are all positive and the sum increases with *misclassification error* because the corresponding slack variables are greater than one.

Overlapping Classes – The Use of Slack Variables

- Minimising misclassification error and maximising the margin together can be achieved by seeking to minimise

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i \qquad (8.43b)$$

in which the positive weight C, called the *regularisation parameter*, adjusts the relative importance of the margin versus misclassification error.

- As before the minimisation is subject to **constraints**: one is (8.43a); the other is that the slack variables are positive. We again accomplish the minimisation by introducing Lagrange multipliers. However, now there is a different multiplier associated with each constraint, so that the Lagrangian is

$$\mathcal{L} = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i - \sum_i \alpha_i\{y_i(\mathbf{w}^{\mathrm{T}}\mathbf{x}_i + w_{N+1}) - 1 + \xi_i\} - \sum_i \mu_i\xi_i \qquad (8.44)$$

in which the α_i and the μ_i are the Lagrange multipliers.

- We now equate to zero the first derivatives with respect to the weight vector and the slack variables in an attempt to find the values that minimise the Lagrangian.

Overlapping Classes – The Use of Slack Variables

- First

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_i \alpha_i y_i \mathbf{x}_i = 0$$

which gives

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \tag{8.45}$$

while

$$\frac{\partial \mathcal{L}}{\partial w_{N+1}} = -\sum_i \alpha_i y_i = 0 \tag{8.46}$$

Now

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \alpha_i - \mu_i = 0 \tag{8.47}$$

- Remarkably, (8.47) removes the slack variables when substituted into (8.44). Since (8.45) and (8.46) are the same as (8.31a) and (8.31b), (8.44) reduces to

$$\mathcal{L} = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_j^{\mathrm{T}} \mathbf{x}_i \tag{8.48}$$

which is identical to the dual formulation of (8.32) which has to be maximised with respect to the Langrange multipliers α_i .

Overlapping Classes – The Use of Slack Variables

- However, the constraints on α_i are now different. Since, by definition the Lagrange multipliers, both α_i and μ_i are non-negative we have

$$0 \leq \alpha_i \leq C \tag{8.49a}$$

and from (8.46)

$$\sum_i \alpha_i y_i = 0 \tag{8.49b}$$

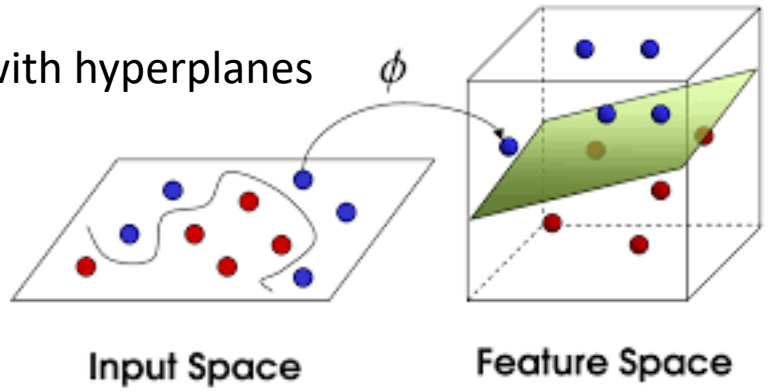
- Again, (8.48) needs to be solved numerically subject to the constraints of (8.49).
  - Once the α_i are found (8.34) is used to label unknown pixels.
  - As with the linearly separable case some of the α_i will be zero, in which case the corresponding training pixels do not feature in (8.33a) and thus in (8.34).
  - The training pixels for which $\alpha_i \neq 0$ are again the support vectors.

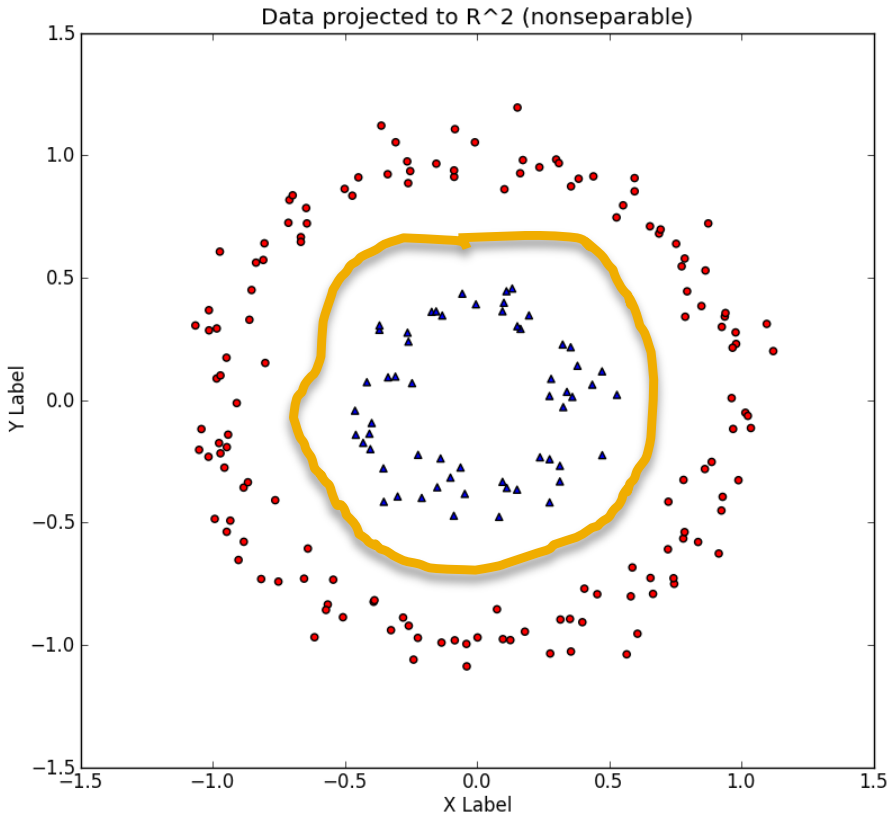
- Technicalities on how to compute w_{N+1} (with a specific role of the slack variables), and more details about the SVM framework, can be found in
  - C.M. Bishop, *Pattern Recognition and Machine Learning*, Springer Science + Business Media LLC, N.Y., 2006.
  - C.J.C. Burges, A tutorial on support vector machines for pattern recognition, *Data Mining and Knowledge Discovery*, vol. 2, 1998, pp. 121–166.

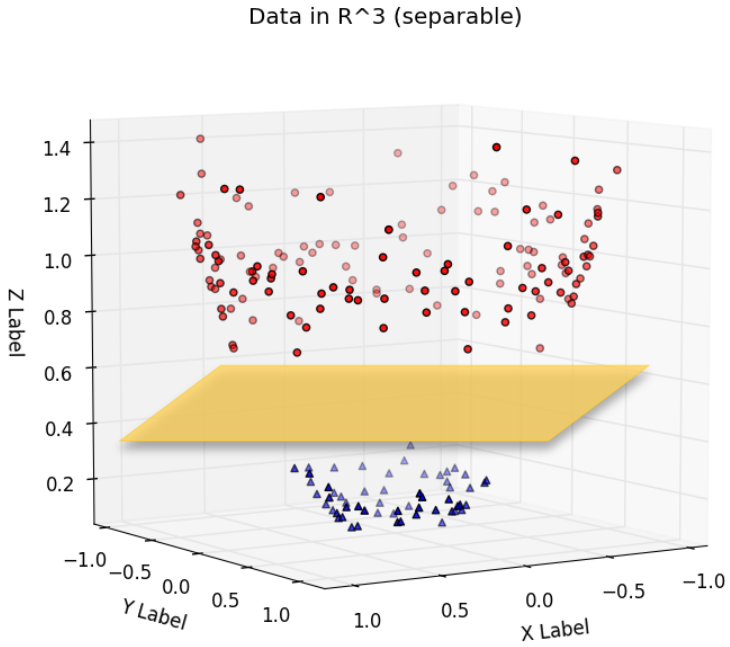
Linear Inseparability – The Use of Kernel Functions

To overcome the limitation of class separation with hyperplanes
Idea: use of feature space transform ϕ

When applied to SVM this leads to a powerful solution, the so called "**kernel trick**"







Linear Inseparability – The Use of Kernel Functions

- If the pixel space is not linearly separable then the development of the previous section will not work without modification. A transformation of the pixel vector $\boldsymbol{x}$ to a different (usually higher order) feature space can be applied that renders the data linearly separable allowing the earlier material to be applied.
- The two significant equations for the linear support vector approach of the preceding section are (8.32) (for finding α_i^o) and (8.34) (the resulting discriminant function). By using (8.33a), (8.34) can be rewritten

$$g(\boldsymbol{x}) = \text{sgn}\left\{\sum \alpha_i^\circ y_i \boldsymbol{x}_i.\boldsymbol{x} + w_{N+1}\right\} \tag{8.35}$$

- Now introduce the feature space transformation $\boldsymbol{x} \rightarrow \Phi(\boldsymbol{x})$ so that (8.32) and (8.35) become

$$L = \sum_i \alpha_i - 0.5 \sum_{i,j} \alpha_i \alpha_j y_i y_j \Phi(\boldsymbol{x}_i).\Phi\left(\boldsymbol{x}_j\right) \tag{8.36a}$$

$$g(\boldsymbol{x}) = \text{sgn}\left\{\sum \alpha_i^\circ y_i \Phi(\boldsymbol{x}_i).\Phi(\boldsymbol{x}) + w_{N+1}\right\} \tag{8.36b}$$

- In both (8.32) and (8.35) the pixel vectors occur only in the dot products. As a result the $\Phi(\boldsymbol{x})$ also appear only in dot products. So to use (8.36) it is strictly not necessary to know $\Phi(\boldsymbol{x})$ but only a scalar quantity equivalent to $\Phi(\boldsymbol{x}_i).\Phi(\boldsymbol{x}_j)$.

Linear Inseparability – The Use of Kernel Functions

- We call the product $\Phi(\boldsymbol{x}_i).\Phi(\boldsymbol{x}_j)$ a *kernel*, and represent it by $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ so that (8.36) becomes

$$L = \sum_i \alpha_i - 0.5 \sum_{i,j} \alpha_i \alpha_j y_i y_j k\left(\boldsymbol{x}_i . \boldsymbol{x}_j\right)$$

$$g(\boldsymbol{x}) = \text{sgn}\left\{\sum \alpha_i^\circ y_i k(\boldsymbol{x}_i, \boldsymbol{x}) + w_{N+1}\right\}$$

Provided we know the form of the kernel we never actually need to know the underlying transformation $\Phi(\boldsymbol{x})$! Thus, after choosing $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ we then find the α_i° that maximise L and use that value in $g(\boldsymbol{x})$ to perform a classification.

- When applying the support vector machine we have a number of parameters to find, along with the support vectors. The latter come out from the optimisation step but the kernel parameters, that we will see in the following examples, have to be estimated to give best classifier performance.
- The problem of choosing the values of these parameters which minimize the expectation of classification error is called the ***model selection***
  - Model selection is typically performed using a cross-validation approach.

Linear Inseparability – Example

- Let us consider this basic example: a kernel composed of the simple square of the scalar product

$$k(\mathbf{x}, \mathbf{y}) = \left[\mathbf{x}^{\mathrm{T}}\mathbf{y}\right]^2$$

  where we use $\mathbf{y}$ instead of $\mathbf{x}_i$ to avoid notation complications.

- We restrict the attention to bidimensional data, $\mathbf{x}=[x_1,x_2]$ and $\mathbf{y}=[y_1,y_2]$, and expanding the kernel operation we have

$$k(\mathbf{x}, \mathbf{y}) = \left[\mathbf{x}^{\mathrm{T}}\mathbf{y}\right]^2 = [x_1y_1 + x_2y_2]^2$$

$$= x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2$$

$$= [x_1^2, \sqrt{2}x_1x_2, x_2^2] \begin{bmatrix} y_1^2 \\ \sqrt{2}y_1y_2 \\ y_2^2 \end{bmatrix} = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} y_1^2 \\ \sqrt{2}y_1y_2 \\ y_2^2 \end{bmatrix}$$

- This show that the above quadratic kernel can be written in the scalar product form required for kernels $k(\mathbf{x}_i, \mathbf{x}) = \phi(\mathbf{x}_i)^{\mathrm{T}}\phi(\mathbf{x})$ and thus is valid.

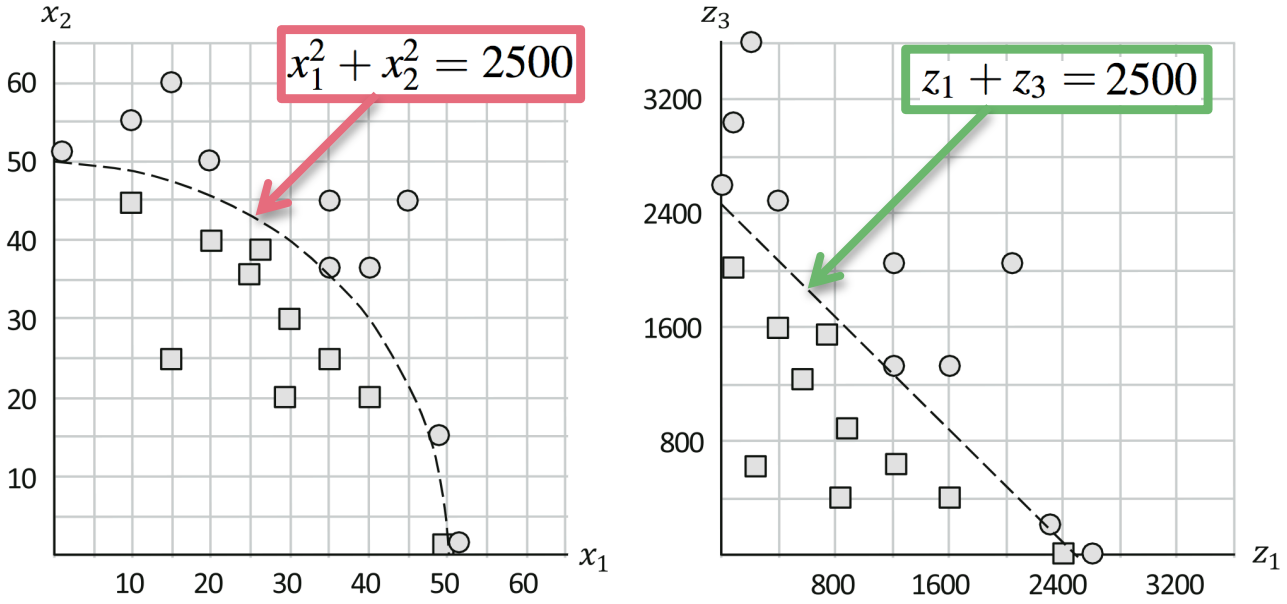
Linear Inseparability – Example

- Observe that the transformation can be now seen explicitly to be

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \tag{8.52}$$

  which transforms the original two dimensional space into three dimensions defined by the squares and products of the original variables.

- The following Figure shows how this transformation leads to linear separability of a two class data set that is not linearly separable in the original coordinates (the z_2 dimension is out of the page, but doesn't contribute to separation)



Linear Inseparability – Popular kernels

- The simple quadratic kernel is of restricted value, but the above example serves to demonstrate the importance of using kernels as substitutions for the scalar product in the key equations.
- A more general polynomial kernel that satisfies the $k(\mathbf{x}_i, \mathbf{x}) = \phi(\mathbf{x}_i)^{\mathrm{T}} \phi(\mathbf{x})$ condition is of the form

$$k(\mathbf{x}_i, \mathbf{x}) = [\mathbf{x}_i^{\mathrm{T}} \mathbf{x} + b]^m$$

  with $b>0$. Values for the parameters b and m have to be found to maximise classifier performance.
- A popular kernel in remote sensing applications is the Gaussian radial basis function kernel, which has one parameter $\gamma>0$ and is based on the distance between the two vector arguments:

$$k(\mathbf{x}_i, \mathbf{x}) = \exp\left\{-\gamma \|\mathbf{x} - \mathbf{x}_i\|^2\right\}$$

  The distance metric chosen is normally Euclidean, although others are also possible. It is interesting to note that the dimensionality of the transformed space with the radial basis function kernel is infinite, which explains its power and popularity.

Linear Inseparability – Popular kernels

- That can be seen by expanding

$$\|\mathbf{x}-\mathbf{x}_i\|^2 = (\mathbf{x}-\mathbf{x}_i)^{\mathrm{T}}(\mathbf{x}-\mathbf{x}_i) = \mathbf{x}^{\mathrm{T}}\mathbf{x} - 2\mathbf{x}^{\mathrm{T}}\mathbf{x}_i + \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_i$$

  so that

$$k(\mathbf{x}_i,\mathbf{x}) = \exp\{-\gamma\mathbf{x}^{\mathrm{T}}\mathbf{x}\}\exp\{2\gamma\mathbf{x}^{\mathrm{T}}\mathbf{x}_i\}\exp\{-\gamma\mathbf{x}_i^{\mathrm{T}}\mathbf{x}\}$$

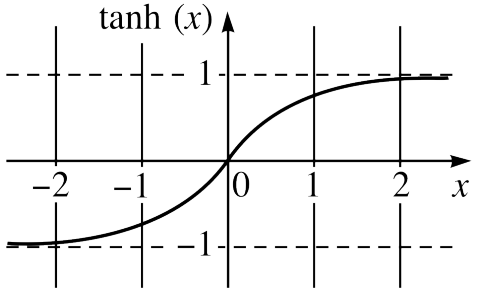
- Taking just the first exponential and replacing it by its power series we have

$$\exp\{-\gamma\mathbf{x}^{\mathrm{T}}\mathbf{x}\} = 1 - \gamma\mathbf{x}^{\mathrm{T}}\mathbf{x} + \frac{\gamma^2}{2}(\mathbf{x}^{\mathrm{T}}\mathbf{x})^2 - \frac{\gamma^3}{6}(\mathbf{x}^{\mathrm{T}}\mathbf{x})^3 + \ldots$$

  - Note the quadratic transformation of the scalar product in the former example (8.52) led to one more dimension than the power. It is straightforward to demonstrate for two dimensional data that the cubic term leads to a four dimensional transformed space. Thus the first exponential, consisting of an infinite set of terms, leads to an infinite transformed space, as do the other exponentials in $k(\mathbf{x}_i,\mathbf{x})$.

- A third option, that has an association with neural network classifiers treated later, is the sigmoidal kernel (which has two parameters to be determined):

$$k(\mathbf{x}_i,\mathbf{x}) = \tanh(\kappa\mathbf{x}_i^{\mathrm{T}}\mathbf{x} + b)$$

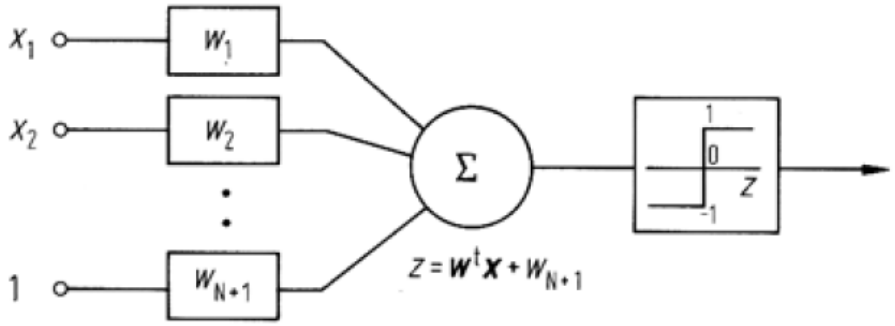


- There is a set of rules that allow new kernels to be constructed from valid kernels, such as those above.

BINARY VS MULTICATEGORY CLASSIFICATION

Binary Classification – The Threshold Logic Unit

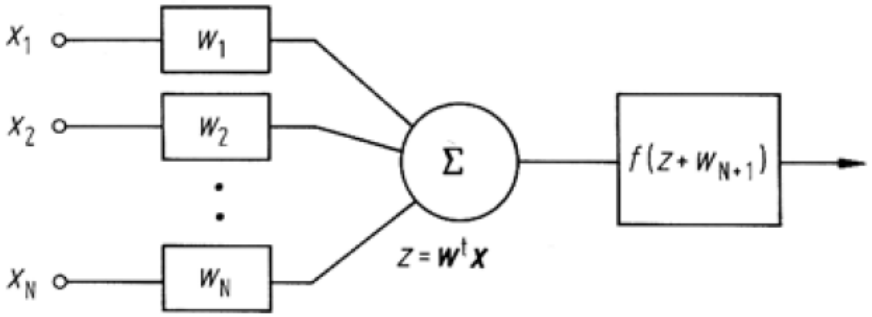
- After the linear (possibly exploiting the "kernel trick") two category classifier has been trained, so that the final version of the weight vector $\boldsymbol{w}$ is available, it is ready to be presented with pixels it has not seen before in order to attach ground cover class labels to those pixels.
- This is achieved through application of the decision rule (8.24), (8.34) or (8.36b).
- It is useful, in anticipation of neural networks, to *picture the classification rule* (8.24), (8.34) or (8.36b) *in diagrammatic form* as depicted in Figure **a**.
  - Simply, this consists of weighting elements, a summing device and an output element which, in this case, performs a thresholding operation.
  - Together these are referred to as a **threshold logic unit** (**TLU**). It bears substantial similarity to the concept of a *processing element used in neural networks* for which the output thresholding unit is *replaced by a more general function* and the *pathway for the unity input* in the augmented pattern vector is actually *incorporated* into the output function.



a

Binary Classification – The Threshold Logic Unit

- The latter can be done for a simple TLU as shown in Figure **b**, in which the simple thresholding element has been replaced by a functional block which performs the addition of the final weighting coefficient to the weighted sum of the input pixel components, and then performs a thresholding (or more general nonlinear) operation.
- Therefore, a more useful representation of a processing element in which the thresholding function is generalised is depicted in the following Figure **b**

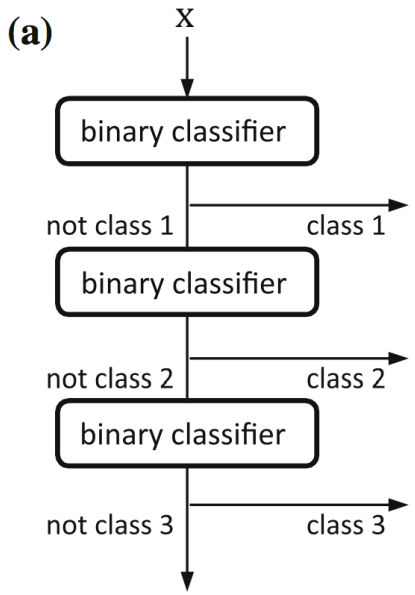


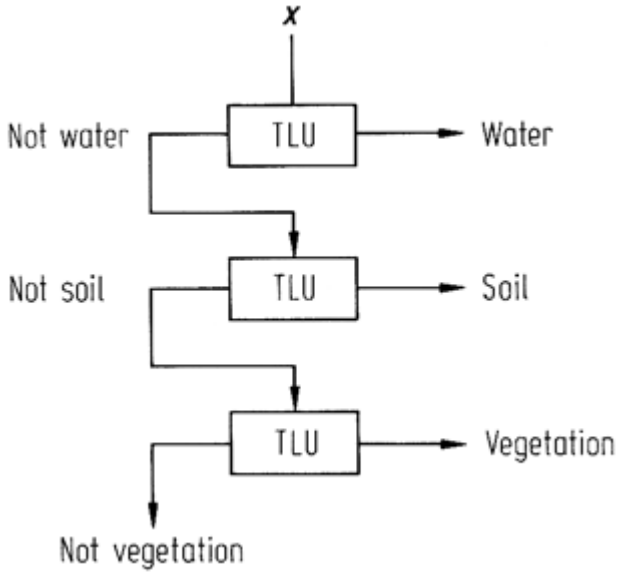
b

Multicategory Classification

- The foregoing work on linear and SVM classification has been based on an approach that can perform separation of pixel vectors into just two categories.
- Were it to be considered for remote sensing or for many other applications, it needs to be extended to be able to cope with a **multiclass problem**.
- Multicategory classification can be carried out in one of the two following ways:

1. First a **decision tree of linear classifiers** (TLUs) can be constructed (see Figure **a**)
   - at each decision node a binary decision is made ("water or not water" in the example)
   - at a subsequent node the (es. not water) category might be further differentiated (es. soil or not soil etc.)
   - *It should be noted that the decision process at each node has to be trained separately*.

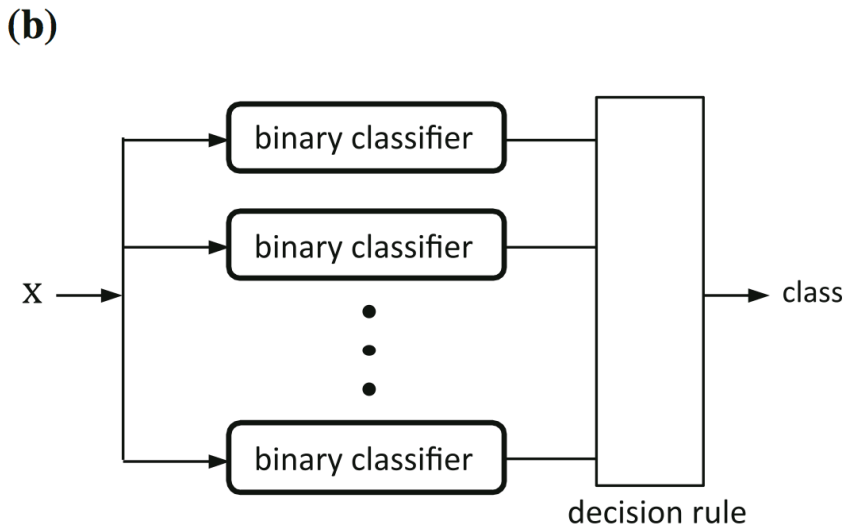




Multicategory Classification

- The disadvantage with this method is that the training classes are unbalanced, especially in the early decision nodes, and it is not known optimally which classes should be separated first.

2. Alternatively, **parallel networks** have been introduced to perform multiclass decisions from binary classifiers, as illustrated in Figure (**b**)



- parallel networks have been used principally in one of two following ways:
  - **One-against-all** (OAA)
  - **One-against-one** (OAO)

See F. Melgani and L. Bruzzone, Classification of hyperspectral remote sensing images with support vector machines, IEEE Transactions on Geoscience and Remote Sensing, vol. 42, no. 8, August 2004, pp. 1778–1790.

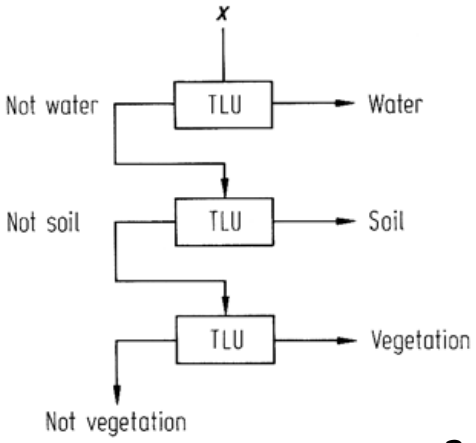
Multicategory Classification

- **One-against-all (OAA)**: each classifier is trained to separate one class from the rest, therefore there are as many classifiers as there are classes, that is *M*.
  - Again, this approach suffers from unbalanced training sets. If the classifiers are Support Vector Machines there is evidence to suggest that such imbalances can affect classifier performance.
  - Also, some form of logic needs to be applied in the decision rule to choose the most favoured class recommendation over the others; that could involve selecting the class which has the largest argument in (8.34). → $g(\boldsymbol{x}) = \text{sgn}\ (\boldsymbol{w}^{\circ}.\boldsymbol{x} + w_{N+1})$
  - Despite these considerations, the method is commonly adopted with SVM.

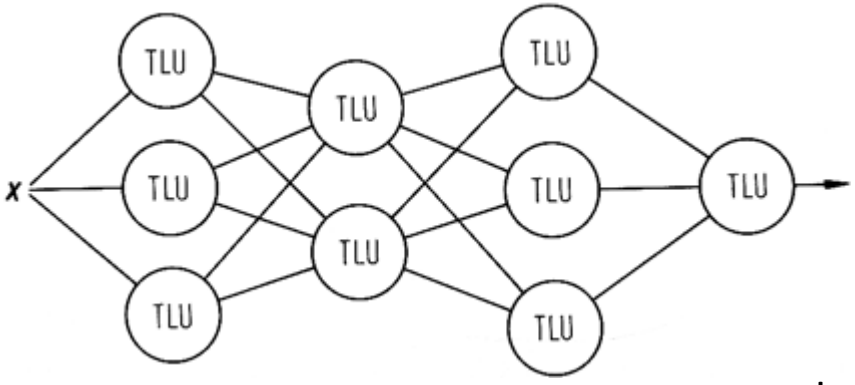
- A better approach is the **One-against-one (OAO)**: in this case $M(M-1)/2$ separate binary classifiersare trained, each of which is designed to separate a pair of the classes of interest. This number covers all possible class pairs.
  - Once trained, an unknown pixel is subjected to each classifier and is placed in the class with the highest number of classification recommendations in favour.
    - Of course, each individual classifier will be presented with pixels from classes for which it was not trained. However, by training for every possible class pair, choosing the most recommended class always works, in principle.
  - The disadvantage of this method is the very large number of binary classifiers needed.
    - Offsetting that problem, however, is the fact that the class-wise binary decisions are generally easier than those in the one against the rest strategy, usually involving simpler SVMs with fewer support vectors for each separate classifier.

Networks of Classifiers – Solutions of Nonlinear Problems

- The already seen decision tree structure for multiclass problems (Figure a) is a classifier network in that a collection of simple classifiers (in that case TLUs) is brought together to solve a complex problem.
- Nilsson (1965, 1990) has proposed a general **network structure** under the name of ***layered classifiers*** consisting entirely of interconnected TLUs, as shown in Figure b.
  - The benefit of forming a classifier network is that data sets that are inherently not separable with a simple linear decision surface should, in principle, be able to be handled since the layered classifier is known to be capable of implementing nonlinear surfaces.
  - The drawback however, is that training procedures for layered classifiers, consisting of *TLUs,* are difficult to determine.



a



b

SVM application: kernel selection and multicategory strategy

- In real SVM applications the ML designer needs to make two initial decisions:
  1. **which kernel to use** to improve separability
  2. **which multiclass strategy** to adopt
  - Most common kernel choices are polynomial and radial basis function
  - Most common multiclass strategy to extend the binary nature of SVM is one-against-one
- Here we focus on **radial basis function** (with parameter γ to be defined) and **OAO**, therefore:
  - there are **two parameters** to be found before the SVM can be applied: the regularisation parameter C which controls the misclassification error that can be tolerated, and the width parameter γ in the RBF kernel, if a radial basis function is used;
  - **however**, they are interdependent, so one cannot be found in isolation from the other;
  - they **possibly vary on a wide range** and a good choice of both **depend on the dataset**.
- A **grid searching process** is usually selected in which an **initial large range of values** for each parameter is chosen and the ranges discretised to give a matrix of C, γ pairs.
  - The SVM is trained on a representative set of data using each pair selected in turn, from which the **best pair is selected**.
  - The **grid spacing can then be narrowed** in the vicinity of that pair and the process repeated, allowing more effective values to be found.
  - This process should be used to find the best parameter pair for each binary classifier in the multiclass topology adopted but is sometimes done for the multiclass data set as a whole.
  - Once the user is satisfied with those values then the OAO multiclass network of SVMs can be fully trained (and best SVM parameters estimated for each classifier).
    - For SVM training several software can be used, e.g. LibSVM http://www.csie.ntu.edu.tw/cjlin/libsvm
- (*option*) each data channel can be shifted to zero mean and scaled to a common range

SVM application: example

- 500x600 pixel segment of a Quickbird 2 (four band multispectral) image taken on 22 April 2002 over the city of Boumerdès in Algeria, Mediterranean Sea, with spatial resolution 0.6m (obtained pansharpening single multispectral bands with resolution 2.4m).
  - Pansharpening is a process of merging high-resolution panchromatic (grayscale) and lower resolution multispectral imagery to increase spectral band resolution.

blue 450-520nm

green 520-600nm

red 630-690nm

NIR 760-890nm

natural color composite

SVM application: example (2)

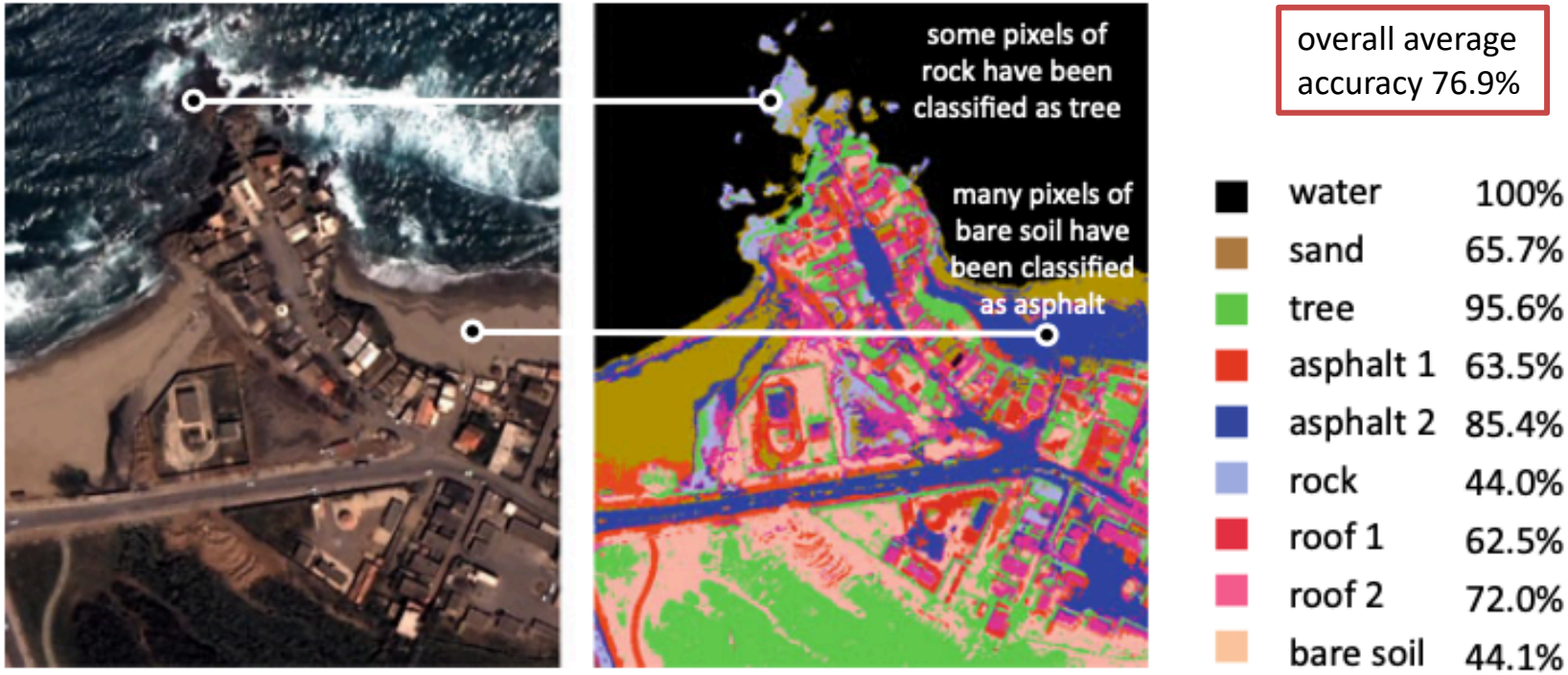
- Seven principal **information classes** are of interest in the scene:
  - two information classes were each separated into two **spectral classes**
    - streets (asphalt 1) and pavements (asphalt 2)
    - tiles (roof 1) and cement (roof 2)
  - train/test pixels were selected and counts as follows

Class	Training pixels	Testing pixels
Water	600	2400
Sand	600	2400
Tree	375	700
Asphalt 1	105	200
Asphalt 2	343	500
Rock	175	450
Roof 1	75	200
Roof 2	294	500
Bare soil	300	700
Total	2867	8050

- A radial basis function kernel was used as was the OAO multi-class strategy. Since there were 9 (spectral) classes, the total number of classifiers to be trained with the OAO strategy was 9 x 8/2 = 36.
- LibSVM was used to perform classification.

SVM application: example (3)

- Instead of grid searching the expertise of users led to a slightly simplified procedure:
  - First, the kernel parameter γ was initially set to 0.25.
  - Next, the regularisation parameter C was varied from 25 to 200 in steps of 25
  - Using the best value found for C, γ was then varied over 0.25 to 2 in steps of 0.25
  - The best parameter pair found was γ = 2 and C = 200.
  - These same parameter values were used for all 36 classifiers.
- Classification results with main errors indicated and some comments:

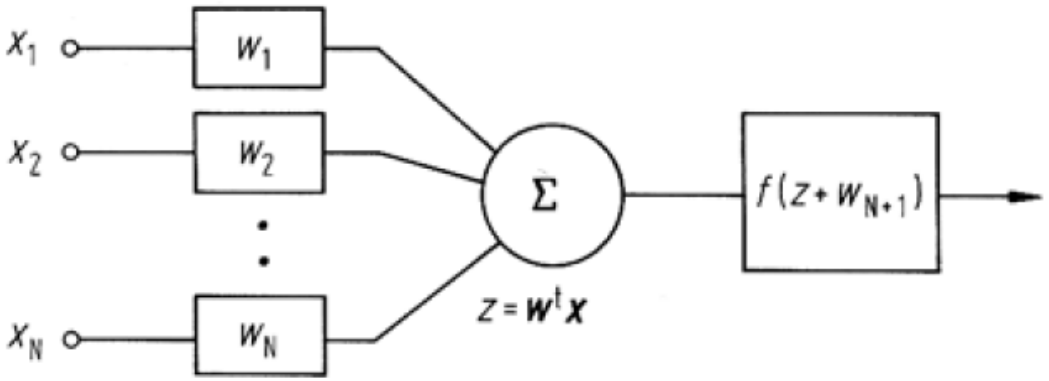


  - the results for rock and bare soil classes was very poor and would not usually be acceptable
  - For a real application the classification would be repeated, perhaps with a more careful choice of the values of γ and C, and with a re-examination of the training data to ensure it is not in error.

NETWORKS OF CLASSIFIERS: THE NEURAL NETWORK APPROACH

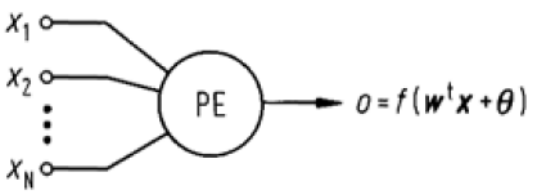
The Neural Network Approach

- For the purposes of this treatment a **neural network** is taken to be *of the nature of a layered classifier* such as depicted above, but with the *very important difference that* ***the nodes are not TLUs***, *although resembling them closely*.
  - **The node structure in Figure can be made much more powerful, and coincidentally lead to a training theorem for multicategory nonlinear classification, if the output processing element does not apply a thresholding operation to the weighted input but rather applies a softer, and mathematically differentiable, operation.**



The Processing Element

- The essential processing node in the neural network to be considered here (sometimes called a **neuron** by analogy to biological data processing from which the term neural network derives) is an element, as anticipated previously, with *many inputs and with a single output*, depicted simply in Figure.



- Its operation is described by

$$o = f(\mathbf{w}^t\mathbf{x} + \theta) \tag{8.37}$$

where θ is a threshold (sometimes set to zero), $\mathbf{w}$ is a vector of weighting coefficients and $\mathbf{x}$ is the vector of inputs.

  - For the special case when the inputs are the band values of a particular multispectral pixel vector it could be envisaged that the threshold θ takes the place of the weighting coefficient w_{N+1} in (8.22).
  - If the function f is a thresholding operation this processing element would behave as a TLU.
  - In general, the number of inputs to a node will be defined by network topology as well as data dimensionality, as will become evident.

- The major difference between the layered classifier of TLUs shown before and the neural network, known as the **multilayer perceptron**, is in the choice of the function f, called the **activation function**.
  - Its specification is simply that it should (not necessarily) emulate thresholding in a soft or asymptotic sense and that is must be differentiable.

The Processing Element

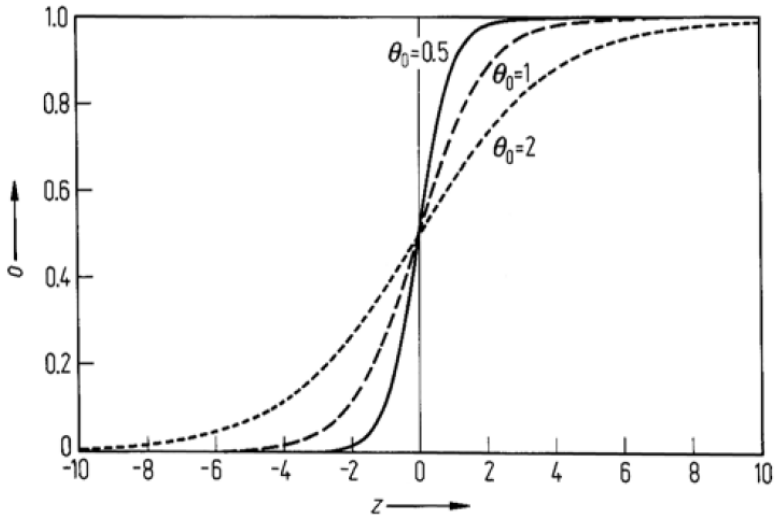
- The most commonly encountered expression is

$$f(z) = \frac{1}{1 + e^{-z/\theta_0}} \tag{8.38}$$

where the argument z is $\boldsymbol{w}^t\boldsymbol{x} + \theta$ as seen in (8.37) and θ_0 is a constant.

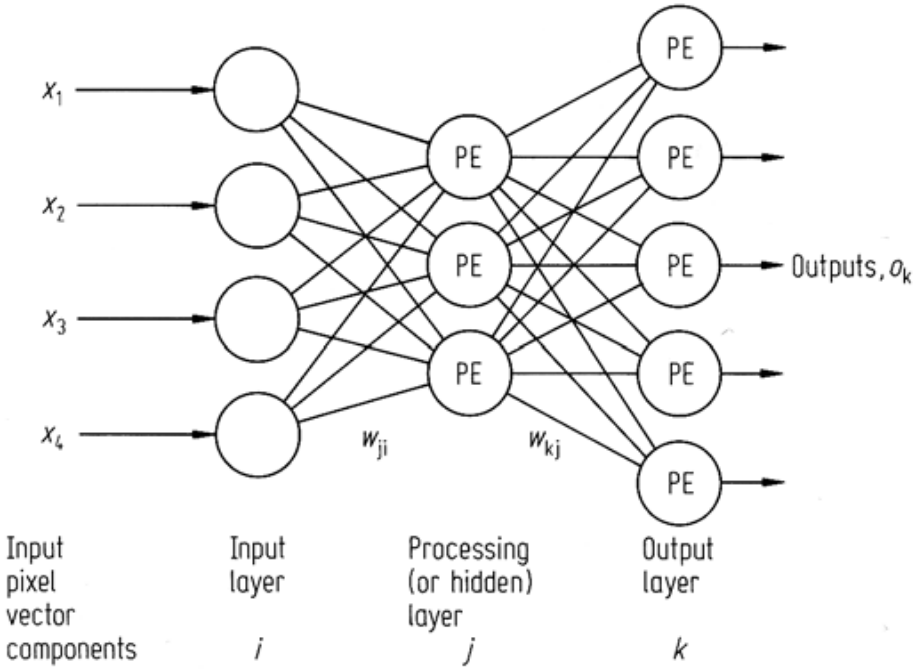
  - This approaches 1 for z large and positive and 0 for z large and negative and is thus *asymptotically thresholding*.
  - It is important to recognise that the outcome of the product $\boldsymbol{w}^t\boldsymbol{x}$ is a simple scalar

- When plotted with $\theta = 0$, (8.38) appears as shown in Figure.
  - For θ_0 very small the activation function approaches a thresholding operation.
  - Usually $\theta_0 = 1$.



The Processing Element

- A **neural network for use in sample (e.g. pixel)-based image analysis** will appear as shown in Figure, being a *layered classifier composed of processing elements of the type shown before*.
  - It is conventionally drawn with an *input layer of nodes* (which has the function of *distributing* the inputs to the processing elements of the next layer, and *scaling* them if necessary) and an *output layer* from which the class labelling information is provided.
  - In between there may be *one or more so-called hidden or other processing layers* of nodes. Before success of deep learning, one hidden layer was considered sufficient, and this is again true in several situations. The number of nodes to use in the hidden layer is often not readily determined. We will return to this below.



Training the Neural Network – Backpropagation

- Before it can perform a classification, the network must be trained.
  - This amounts to using labelled training data to help determine the weight vector $\boldsymbol{w}$ and the threshold θ in (8.37) for each processing element connected into the network.
  - Note that the *constant* θ_0 in (8.38), which governs the gradient of the activation function as seen in a previous Figure, as well as the very same selection of which activaton function to use (among the possible ones we will see), is generally pre-specified and *does not need to be estimated* from the training data. This could be the task of subsequent (*hyper-*)*parameter selection* phases.
- Part of the complexity in understanding the training process for a neural net is caused by the *need to keep careful track of the parameters and variables over all layers and processing elements*, how they vary with the presentation of training samples (or pixels) and (as it turns out) with iteration count.
  - This can be achieved by the use of a simple *generalised notation* (Pao, 1989) that avoids the need of more detailed subscript conventions.
- The derivation will be focused on the 3-layer neural network depicted before, since this architecture is sufficiently representative and has been found effective for many applications.
  - Although the results generalise *to more layers* and also in the modern context of *deep* neural networks, we will dedicate a separate treatment for a more general and powerful way to see the backpropagation method in the context of Deep Learning.

Training the Neural Network – Backpropagation

- The previous Figure incorporates the *nomenclature* used.
  - The three layers are lettered as i, j, k with k being the output.
  - The set of *weights* linking layer i PEs with those in layer j are represented generally by w_{ji}, while those linking layers j and k are represented by w_{kj}.
  - There will be a very large number of these weights, but in deriving the training algorithm it is *not necessary to refer to them all individually*.
  - Similarly, the general activation function *arguments* $z_{i,j,k}$ and *outputs* $o_{i,j,k}$, can be used to represent all the arguments and outputs respectively in the corresponding layer *i*, *j* or *k*.
- For j and k layer PEs (8.37) is

$$o_j = f(z_j) \quad \text{with} \quad z_j = \sum_i w_{ji} o_i + \theta_j \tag{8.39a}$$

$$o_k = f(z_k) \quad \text{with} \quad z_k = \sum_j w_{kj} o_j + \theta_k \tag{8.39b}$$

- The sums in (8.39) are shown with respect to the indices i and j.
  - This should be read as meaning the sums are taken over all inputs of particular layer j and layer k PEs respectively.
  - Note also that the sums are expressed in terms of the outputs of the previous layer since these outputs form the inputs to the PEs in question.

Training the Neural Network – Backpropagation

- An *untrained* or poorly trained network will give *erroneous* outputs.
  - Therefore, as a *measure of how well a network is functioning* during training, we can assess the outputs at the last layer (k).
  - A suitable measure along these lines is to use the sum of the *squared output error*.
  - The error made by the network when presented with *a single training sample* can thus be expressed

$$E = \frac{1}{2}\sum_{k}(t_k - o_k)^2 \qquad (8.40)$$

    where the t_k represent the desired or target outputs(*) and o_k represents the actual outputs from the output layer PEs in response to the training sample.
  - The factor of ½ is included for arithmetic convenience in the following.
  - The sum is over all output layer PEs.
- More in general the non ideal behavior of the network can be measured differently. We will see and refer to this as the **Loss function**.
- A useful training strategy is to adjust the weights in the processing elements until the error (or loss) has been minimised, at which stage the actual outputs are as close as possible to the desired outputs.

(*) *These will be specified from the training data labelling. The actual value taken by t_k however will depend on how the outputs themselves are used to represent classes. Each output could be a specific class indicator (e.g. 1 for class 1 and 0 class 2); alternatively, some more complex coding of the outputs could be adopted (this is considered later).*

Training the Neural Network – Backpropagation

- A common approach for adjusting weights to reduce (and thus minimise) the value of a function of which they are arguments, is to modify their values proportional to the negative of the partial derivative of the function. This is called a **gradient descent technique** (**). Thus for the weights linking the j and k layers let

$$w'_{kj} = w_{kj} + \Delta w_{kj}$$

with

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}}$$

where η is a positive constant that controls the amount of adjustment. This requires an expression for the partial derivative, which can be determined using the chain rule

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial z_k} \frac{\partial z_k}{\partial w_{kj}} \qquad (8.41)$$

each term of which must now be evaluated.

(**) Another optimisation procedure used successfully for neural network training in remote sensing is the *conjugate gradient method* (Benediktsson et al., 1993).

Training the Neural Network – Backpropagation

- From (8.39b) and (8.38) we see (for $\theta_0 = 1$)

$$\frac{\partial o_k}{\partial z_k} = f'(z_k) = (1 - o_k)o_k \tag{8.42a}$$

and

$$\frac{\partial z_k}{\partial w_{kj}} = o_j \tag{8.42b}$$

Now from (8.40)

$$\frac{\partial E}{\partial o_k} = -(t_k - o_k) \tag{8.42c}$$

Thus the correction to be applied to the weights is

$$\Delta w_{kj} = \eta(t_k - o_k)(1 - o_k)o_k o_j \tag{8.43}$$

For a given trial, all of the terms in this expression are known* so that a beneficial adjustments can be made to the weights which link the hidden layer to the output layer.

*They can be computed in a forward pass preceding the current backpropagation step

Training the Neural Network – Backpropagation

- Now consider the weights that link the i and j layers. The weight adjustments are

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}}$$

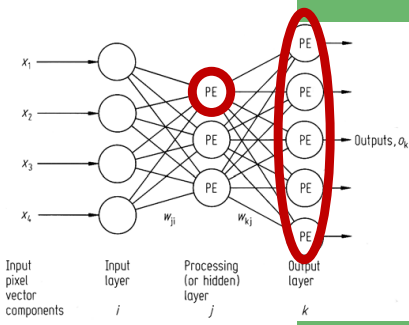
In a similar manner to the above development we have

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial o_j} (1 - o_j) o_j o_i$$

Unlike the case with the output layer, however, we cannot obtain an expression for the remaining partial derivative from the error formula, since the o_j are not the outputs at the final layer, but rather those from the hidden layer. Instead we express the derivative in terms of a chain rule involving the output PEs. Specifically

$$\frac{\partial E}{\partial o_j} = \sum_k \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial o_j}$$

$$= \sum_k \frac{\partial E}{\partial z_k} w_{kj}$$



The remaining partial derivative can be obtained from (8.42a) and (8.42c) as

$$\frac{\partial E}{\partial z_k} = -(t_k - o_k)(1 - o_k) o_k$$

so that

$$\Delta w_{ji} = \eta (1 - o_j) o_j o_i \sum_k (t_k - o_k)(1 - o_k) o_k w_{kj} \qquad (8.44)$$

Training the Neural Network – Backpropagation

- Having determined the w_{kj} from (8.43), it is now possible to find values for the w_{ji} since all other entries in (8.44) are known or can be calculated readily.

For convenience we now define

$$\delta_k = (t_k - o_k)(1 - o_k)o_k \tag{8.45a}$$

and

$$\delta_j = (1 - o_j)o_j \sum_k (t_k - o_k)(1 - o_k)o_k w_{kj}$$

$$= (1 - o_j)o_j \sum_k \delta_k w_{kj} \tag{8.45b}$$

so that we have

$$\Delta w_{kj} = \eta \delta_k o_j \tag{8.46a}$$

and

$$\Delta w_{ji} = \eta \delta_j o_i \tag{8.46b}$$

both of which should be compared with (8.26) to see the effect of a differentiable activation function.

The thresholds θ_j and θ_k in (8.39) are found in exactly the same manner as for the weights in that (8.46) is used, but with the corresponding inputs chosen to be unity.

Training the Neural Network – Backpropagation

- Now that we have the mathematics in place it is possible to describe how training is carried out.
  - The network is initialised with an arbitrary set of weights in order that it can function to provide an output.
  - The training samples are then presented one at a time to the network.
  - For a given sample the output of the network is computed using the network equations.
- Almost certainly the output will be incorrect to start with – i.e. the o_k will not match the desired class t_k for the sample, as specified by its labelling in the training data.
  - Correction to the output PE weights, described in (8.46a), is then carried out, using the definition of δ_k in (8.45a).
  - With these new values of δ_k and thus w_{kj} (8.45b) and (8.46b) can be applied to find the new weight values in the earlier layers.
  - In this way the effect of the output being in error is *propagated back through the network* in order to correct the weights.
- The technique is thus often referred to as **back propagation**.

Training the Neural Network – Backpropagation

- Pao (1989) recommends that the weights not be corrected on each presentation of a single training sample, but rather that the *corrections for all samples in the training set be aggregated into a single adjustment*.
- Thus for *p* training patterns the bulk adjustments are

$$\Delta w'_{kj} = \sum_p \Delta w^p_{kj} \qquad \text{and} \qquad \Delta w'_{ji} = \sum_p \Delta w^p_{ji}$$

This is tantamount to deriving the algorithm with the error being calculated over all pixels p in the training set, viz $E_p = \sum_p E$, where E is the error expressed for a single pixel in (8.40).

- Other sample *batching* strategies will be seen in the case of Deep Neural Nets
- After the weights have been so adjusted the *training samples are presented to the network again* and the outputs re-calculated to see if they correspond better to the desired classes.
- *Usually they will still be in error and the process of weight adjustment is repeated*.
  - Indeed, the process is iterated as many times as necessary in order that the network respond with the correct class for each of the training samples or until the number of errors in classifying the training pixels is reduced to an acceptable level.

Choosing the Network Parameters

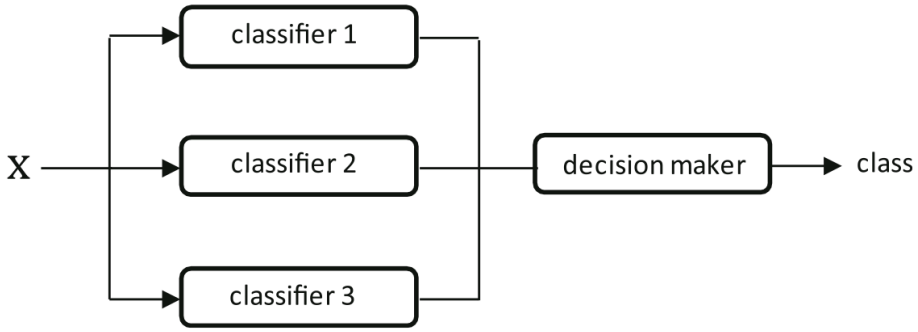
- When considering the use of the neural network approach to pixel classification it is necessary to make several key decisions beforehand.
  - First, the **number of layers** to use must be chosen.
    - For ***shallow* neural networks** (as opposed to the ***deep*** NN evolution), a *three-layer network is considered sufficient*, with the purpose of the first layer being simply to distribute (or fan out) the components of the input pixel vector to each of the processing elements in the second layer.
    - Thus, the first layer does no processing as such, apart perhaps from *scaling* the input data, if required.
  - The next choice relates to the **number of elements in each layer**.
    - The *input layer* will generally be given as many nodes as there are components (features) in the pixel vectors.
    - The number to use in the *output node* will depend on how the outputs are used to represent the classes.
      - The simplest method is to let each *separate output signify a different class*, in which case the number of output processing elements will be the same as the number of training classes.
      - Alternatively, a *single PE could be used to represent all classes*, in which case a different value or level of the output variable will be attributed to each class.
      - A further possibility is to *use the outputs as a binary code*, so that two output PEs can represent four classes, three can represent 8 classes and so on.
    - As a general guide the number of PEs to choose for the *hidden or processing layers* should be *the same as or larger than the number of nodes in the input layer* (Lippmann, 1987).

COMMITTEES OF CLASSIFIERS: ENSEMBLE CLASSIFICATION

- Bagging
- Boosting

Problem statement and decision logics

- Classically, a committee classifier consists of a number of algorithms that all operate on the same data set to produce individual, sometimes competing, recommendations about the class membership of a pixel, as shown in Figure.
  - Those recommendations are fed to a decision maker, or chairman, who resolves the final class label for the pixel.



- The decision maker resolves the conflicts by using one of several available logics.
  1. One is the **majority vote**, in which the chairman decides that the class most recommended by the committee members is the most appropriate one for the pixel.
  2. Another is **veto logic**, in which all classifiers have to agree about class membership before the decision maker will label the pixel.
  3. Yet another is **seniority logic**, in which the decision maker always consults one particular classifier first (the most ‘‘senior’’) to determine the label for a pixel.
     - If that classifier is unable to recommend a class label, then the decision maker consults the next most senior member of the committee, and so on until the pixel is labelled. Seniority logic has been used as a means for creating piecewise linear decision surfaces.

Bagging

- Apart from the simple approach of committee labelling based on logical decisions several other procedures that use a number of similar (so called ***weak***) classifiers (e.g. **TLU**s) have been devised in an endeavour to improve the classification performance:
  - **Bagging (b**ootstrap **agg**regat**ing)**
  - **Boosting (**mainly **ada**ptive **boost**ing ***AdaBoost*)**

- **Bagging**
  - Different data sets are chosen by the process called *bootstrapping*, in which the available K training pixels are used to generate L different data sets, each containing N training pixels, in the following manner.

1. N pixels are chosen randomly from the K available, with replacement (in other words the same pixel could appear more than once among the N chosen in the L data sets).
2. Each of the L data sets is used to train a classifier.
3. The results of the individual classifiers are then combined by majority voting.

Boosting: *AdaBoost*

- **Ada**ptive **boost**ing, called **AdaBoost**: is a committee of binary weak classifiers in which the members are trained sequentially, in the following manner.
  - The first weak classifier is trained.
  - Training pixels that are found to be in error are then emphasised in the training set and the next classifier trained on that enhanced set.
  - Training pixels unable to be separated correctly by the second classifier are given a further emphasis and the third classifier is trained, and so on.

  The final label allocated to a pixel is based on the outputs of all classifiers.

- Algorithmically, it proceeds in the following steps:

Suppose there are K training pixels; the correct label for the kth pixel is represented by the binary variable y_k which takes the values $\{+1,-1\}$ according to which class the kth training pixel belongs. Define $t_k \in \{+1, -1\}$ as the actual class that the pixel is placed into by the trained classifier. For correctly classified pixels $t_k = y_k$ while for incorrectly classified pixels $t_k \neq y_k$.

1. Initialise a set of weights for each of the training pixels in the first classifier step according to *

$$w_k^1 = 1/K$$

* All superscripts in this section are stage (iteration) indices and not powers.

Boosting: *AdaBoost*

2. For $l = 1 \ldots L$, where L is the number of classifiers in the committee, carry out the following steps in sequence
   a. Train a classifier using the available weighted training data. Initially each training pixel in the set is weighted equally, as in 1. above.
   b. Set up an error measure after the lth classification step according to

$$e^l = \frac{\sum_k w_k^l I(t_k, y_k)}{\sum_k w_k^l}$$

in which $I(t_k, y_k) = 1 \text{ for } t_k \neq y_k$
$= 0 \text{ otherwise}$

   c. Form the weights for the $(l+1)$th classifier according to

either $$w_k^{l+1} = w_k^l \exp\{\alpha^l I(t_k, y_k)\} \quad (8.56a)$$

or $$w_k^{l+1} = w_k^l \exp\{-\alpha^l t_k y_k\} \quad (8.56b)$$

with $$\alpha^l = \ln\{(1 - e^l)/e^l\} \quad (8.56c)$$

Boosting: *AdaBoost*

Both (8.56a) and (8.56b) enhance the weights of the incorrectly classified pixels; (8.56a) leaves the weights of the correct pixels unchanged, while (8.56b) de-emphasises them.

d. Test the class membership of the kth pixel according to

$$T_L(\mathbf{x}_k) = \text{sgn} \sum_l \alpha^l t_k^l(\mathbf{x}_k) \tag{8.56d}$$

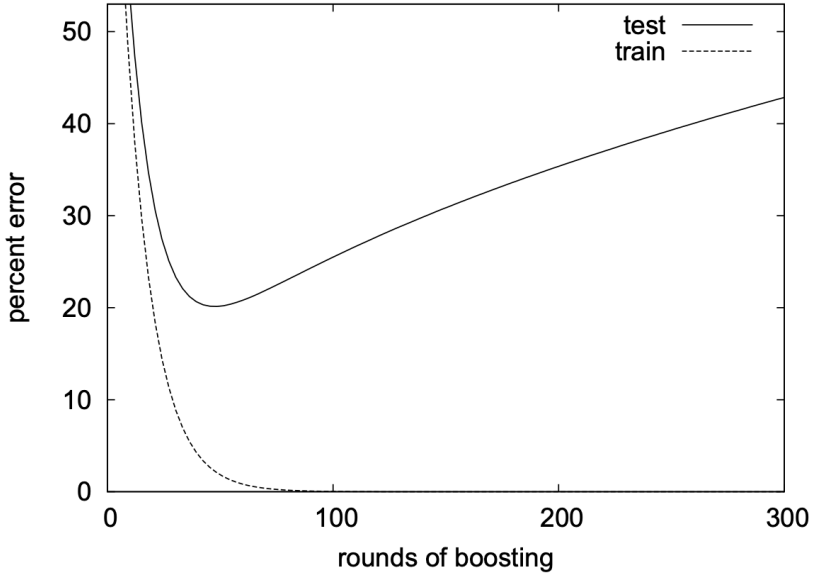
This weights the class memberships recommended by the individual classifiers to come up with the final label for the pixel. Note that $T_L \in \{1, -1\}$ for this binary classifier.

Bishop * gives a simple example of AdaBoost in which the improvement of accuracy is seen as more classifiers are added. Accuracy may not always improve initially and many, sometimes 100s, of stages are needed to achieve good results.

*C.M. Bishop, *Pattern Recognition and Machine Learning*, Springer Science + Business Media LLC, N.Y., 2006.

Boosting: *AdaBoost*

- Why AdaBoost training is effective
  - only those features known to improve the predictive power of the model are selected, reducing dimensionality and potentially improving execution time as irrelevant features need not be computed.
  - AdaBoost can works if the so called *weak learning* assumption is satisfied : every weak classifier has accuracy just a little bit better than random guessing
  - there is an overfitting risk as the number of committee classifiers L increases



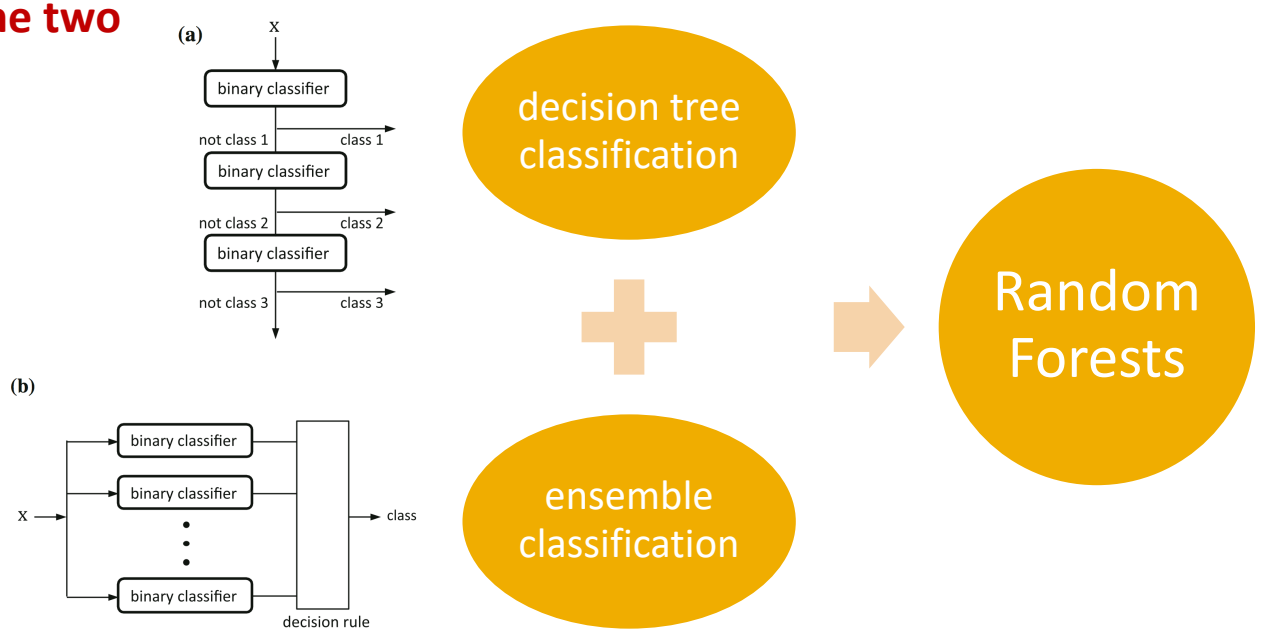
- Dataset preparation for AdaBoost
  - **Quality Data**: Because the ensemble method continues to attempt to correct misclassifications in the training data, training data must be of a high-quality (low noise).
  - **Outliers**: Outliers will force the ensemble classifier to work hard to correct for cases that are unrealistic. These should be removed from the training dataset.

COMMITTEES OF CLASSIFIERS: RANDOM FORESTS

- Decision Tree Classifiers
- CART (Classification and Regression Trees)
- Random Forests

The idea of ensembling multistage classifiers

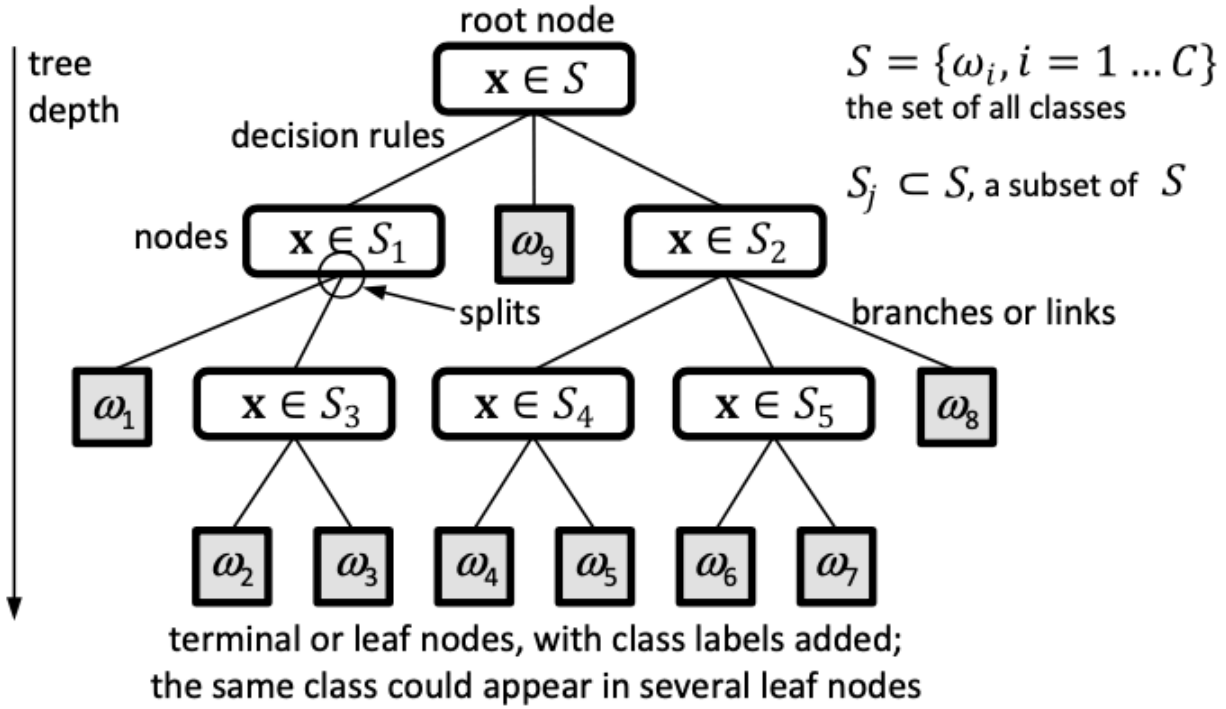
- The ***ensembling idea*** inbuild in committees of classifiers is powerful
- From the literature boosting is more accurate than bagging:
  - **bagging approaches reduce the classification variance** but they have little effect on the classification bias
  - **boosting approaches** have been shown to **reduce classification variance and bias**. However, they require large computational resources, overfit if there are insufficient training samples, and are sensitive to any outliers present in the training samples.
- Ensemble classifiers usually produce better results than other combinations of classifiers such as decisions tree classifiers (we already considered to obtain multiclass decisions)
- **Idea: combine the two**



Decision Tree Classifiers

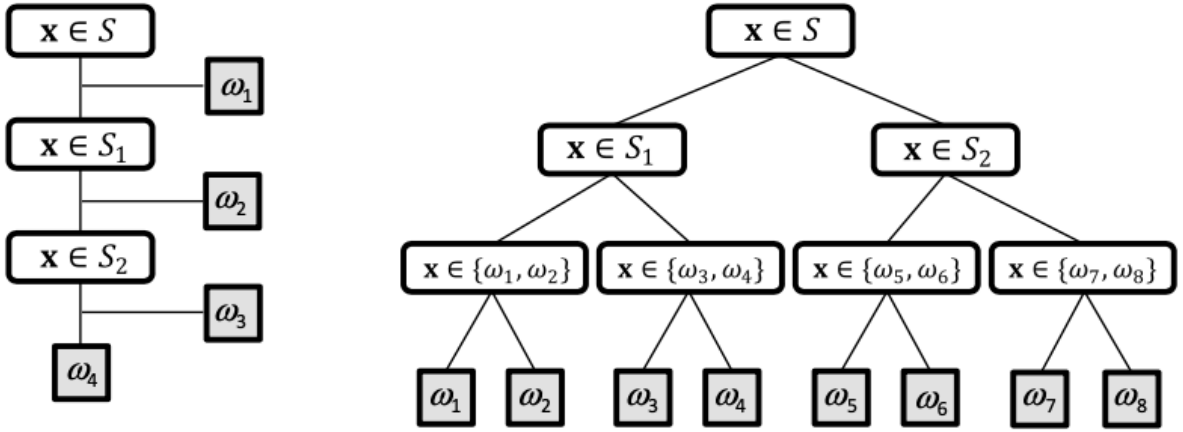
- Classifiers such as the maximum likelihood rule and the support vector machine are single stage processes.
- In multistage classification a series of (partial) decisions is taken to determine the most appropriate label for a pixel.
- The most commonly encountered multistage classifier is the **decision tree**:
  - a number of connected classifiers (called decision nodes) none of which is expected to perform the complete segmentation of the image data set (but only a part of the task).

Example of decision tree on 9 classes



Decision Tree Classifiers

- The binary decision tree we already saw for multicategory classification is the simplest form of decision tree.
- A binary (on 4-classes) and a more general version (on 8-classes) are depicted in figure:



- Taxonomy:

Root node	This is where the tree commences
Decision Node	Intermediate node (and the root node)
Terminal (leaf) node	Final node, which usually represents a single class
Link (or branch)	Connection between nodes
Tree depth	Number of layers from the root node to the most distant leaf
Antecedent	Node immediately above a node of interest (parent node)
Descendant	Node immediately following a node of interest (child node)
Split	The result of a decision to create new descendent nodes

Decision Tree Classifiers

- The **advantages** of the decision tree approach are that:
  - **different sets of features can be used at each decision node**; this allows
    - feature subsets to be chosen that ***optimise*** segmentations (i.e. pattern space region subdivisions)
    - reduced feature subsets at individual decisions (catch a dog if it were running around on the 2D plane is much easier than hunting birds in 3D, and so on for higher dimensionalities --> this is usually called the ***curse of dimensionality***)
  - **simpler segmentations** than those needed when a decision has to be made among all available labels for a pixel in a single decision
  - *different algorithms* (if suitable) can be used at each decision node
  - *different data types* (if suitable) can be used at each decision node
- On the other hand, decision tree design is usually ***not straightforward***.
  - Sometimes the analyst can (or have to) design a tree intuitively
    - For example, near infrared data might be used to segment between land and water bodies; subsequently, thermal infrared data might be used to map temperature contours within the water.
- **Decision tree design** involves
  1. finding the structure of the tree,
  2. choosing the subset of features to be used at each node,
  3. selecting the decision rule to use at each node.
  - If we restrict the range of possibilities for the last two requirements some automated procedures are possible, as developed in the following.

CART (Classification and Regression Trees)

- The CART tree growing methodology restricts, and thus simplifies, the possible options for how the decision nodes function:
  - **only one feature is involved in each decision step**
  - **a simple threshold rule is used in making that decision**
- CART is a supervised classification procedures:
  - it uses labelled training data to construct the tree
  - once the tree has been built it can then be used to label unseen data
- At each node in CART, including at the root node, a decision is made to split the training samples into two groups (***binary*** splits)
  - the aim is to **produce sub-groups that are purer class-wise** than in the immediately preceding node as follows

1. **all of the training data from all classes is fed to the root node**
2. **all of the possible binary partitions of the training pixels are evaluated (i.e. with respect to all possible single features and all possible thresholds)** and **the partition which minimises the class mixture in the two produced groups is chosen**
   - For example, if there were five separate classes in the training set then we would expect the sub-groups to have pixels from fewer than five classes and, in some cases, hope that one sub-group might have pixels from one class only.
3. **keep subdividing the groups as going down the tree** so that, ultimately, the process ends up with groups containing pixels from only one class, i.e. “pure” groups (leaf nodes)

CART (Classification and Regression Trees)

- To implement the above process we need a ***measure*** of **how the train classes are mixed vs pure** at every decision (split at the node N) within the tree.
- An impurity measure can be used to this aim. Common adopted metrics are:
  - **Gini impurity** or **Gini index**
  - **Entropy**
- **Gini index** at the N^{th} node for binary decisions is defined as

$$i(N) = \sum_j P_{\omega_j}(1 - P_{\omega_j})$$

where P_{ω_j} is the fraction of the training pixels at node *N* that are in class ω_j, while $1 - P_{\omega_j}$ is the fraction not in class ω_j.

  - If all the pixels at the node were from a single class, then $P_{\omega_j} = 1$ so that $i(N) = 0$, indicating no impurity.
  - If there were M equally distributed classes in the training set then $i(N)$ is a maximum and equal to $1 - 1/M$, which is larger for larger M, as would be expected.

- **Entropy** at the N^{th} node is defined as

$$i(N) = -\sum_j P_{\omega_j} \log_2 P_{\omega_j}$$

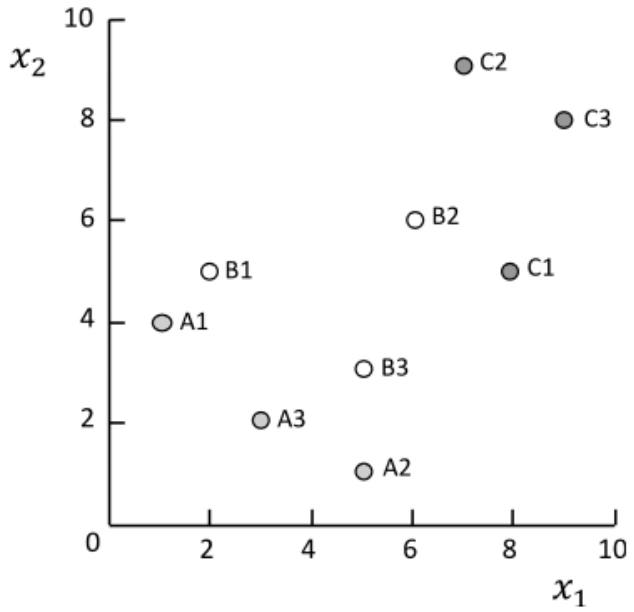
again, this is zero if all the training pixels are from the same class and is large when the group is mixed.

CART (Classification and Regression Trees)

- **In splitting the training pixels as we go down the tree, we are interested in that split which gives the greatest drop in impurity from the antecedent to the descendent nodes**.
- To find the split that generates the purest descendent groups, we can measure the reduction in impurity by subtracting the impurities of the descendent nodes from the impurity of their antecedent node, weighted by the relative proportions of the training pixels in each of the descendent nodes.
- Let N refer to a node and N_L and N_R refer to its left and right descendents; let P_L be the proportion of the training pixels from node N that end up in N_L.
- Then the **reduction in impurity** in splitting N into N_L and N_R is

$$\Delta i(N) = i(N) - P_L i(N_L) - (1 - P_L) i(N_R)$$

- To see how this is used in building a decision tree consider the training data shown in Figure This consists of *three classes*, each of which is described by *two features* (bands).



CART (Classification and Regression Trees)

- When the Gini impurity is used, the Table shows the original impurity for the complete set of data and the subsequent drops in impurity with various candidate splits until reaching leaves.

Not all possible splits are given because the number of combinations is excessive; only those that are clearly the most favoured are shown.

shaded boxes highlight the greatest reduction in impurity

Original unsplit training set					
A1 A2 A3 B1 B2 B3 C1 C2 C3			$i(N) = 0.667$		
First split candidates					
left descendent		*right descendent*	$i(N_L)$	$i(N_R)$	$\Delta i(N)$
A1 A2 A3 B1 B2 B3	(x_1)	C1 C2 C3 (leaf node)	0.500	0	0.334
A2 A3	(x_2)	A1 B1 B2 B3 C1 C2 C3	0	0.612	0.191
C2 C3	(x_2)	C1 A1 A2 A3 B1 B2 B3	0	0.612	0.191
A1	(x_1)	A2 A3 B1 B2 B3 C1 C2 C3	0	0.656	0.084
Second split candidates from A1 A2 A3 B1 B2 B3 | C1 C2 C3 first split					
B1 B2	(x_2)	A1 A2 A3 B3	0	0.375	0.250
A2 A3	(x_2)	A1 B1 B2 B3	0	0.375	0.250
A1	(x_1)	A2 A3 B1 B2 B3	0	0.480	0.100
Third split from B1 B2 | A1 A2 A3 B3 second split					
A1 A3	(x_1)	A2 B3	0	0.500	0.125
Fourth split from A1 A3 | A2 B3 third split					
A2 (leaf node)	(x_2)	B3	0	0	0.500
Third split from A2 A3 | A1 B1 B2 B3 second split					
A1 (leaf node)	(x_1)	B1 B2 B3 (leaf node)	0	0	0.375

two equally favourable splits

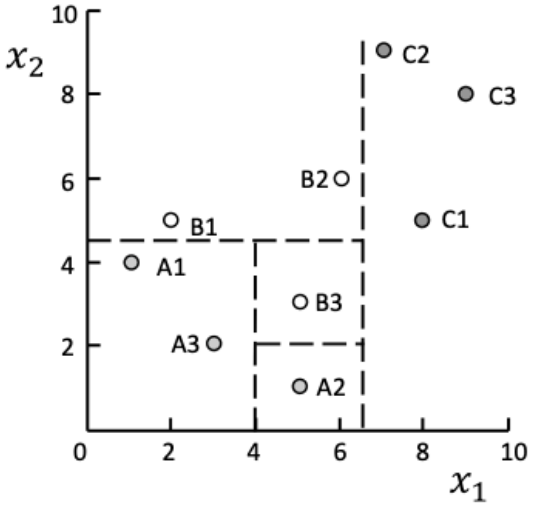
Tree 1

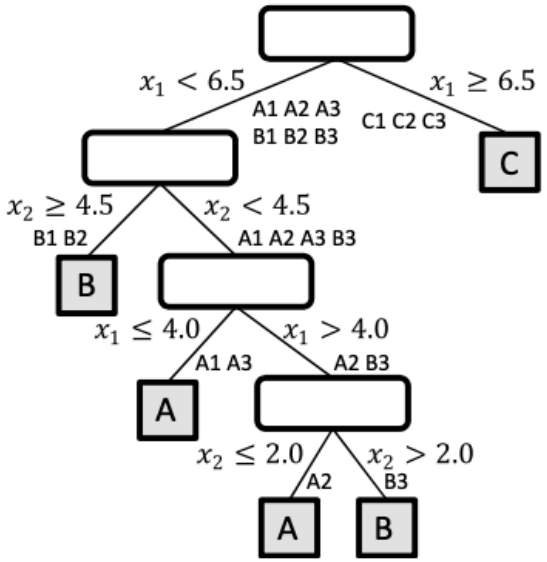
Tree 2

CART (Classification and Regression Trees)

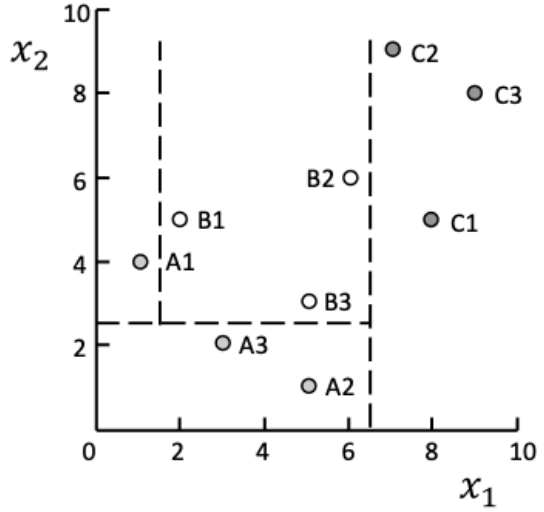
- There are several split options later in the tree;
  - only two (the best found) are given to demonstrate that trees are often not unique but will still segment the data as required.
  - One of the problems with splitting based on the **simple thresholding of individual features** is that quite complicated trees can be generated compared with what should be possible if more flexibility was introduced into the decision functions and thus the decision boundaries in the pattern space.
  - For example, the data could easily be split into the three classes by **two inclined linear surfaces**, one between class A and B pixels, and the other between class B and C pixels.

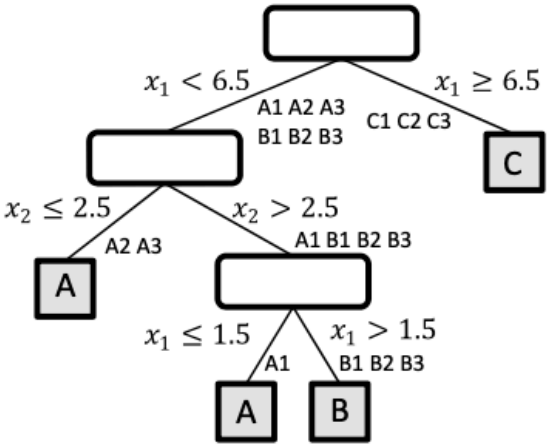
Tree 1





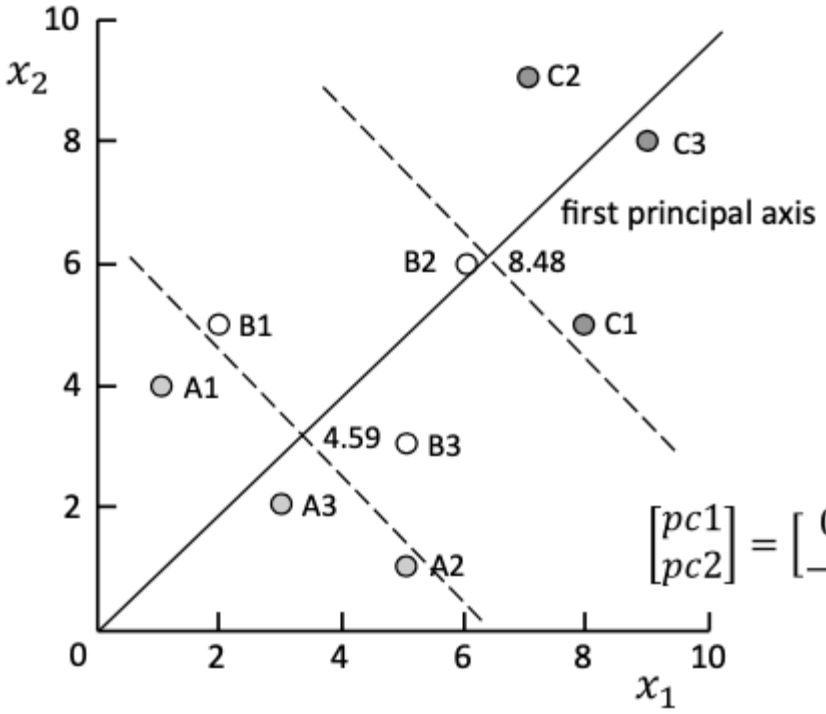
Tree 2

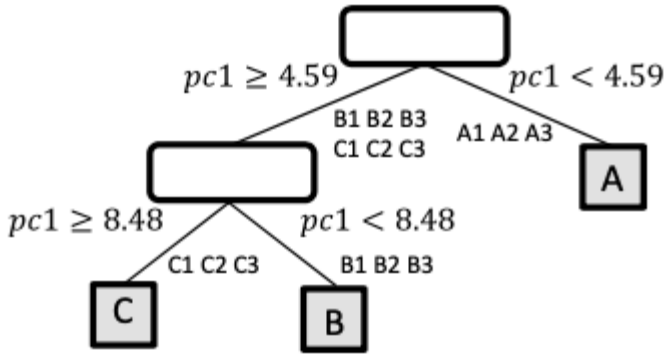




CART (Classification and Regression Trees)

- While it is feasible to develop a tree design methodology that implements linear decisions of that nature at each node, **it is sometimes simpler to transform the data prior to tree growth**.
- For example if our example dataset is used to generate its **principal components**, the principal axes will provide a simpler tree.
- The Figure shows the principal axes of the data from our dataset along with a decision tree generated with the CART methodology.

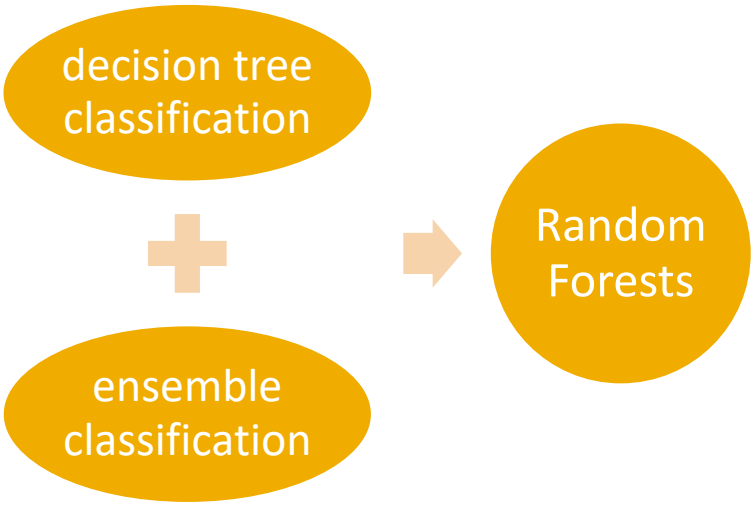


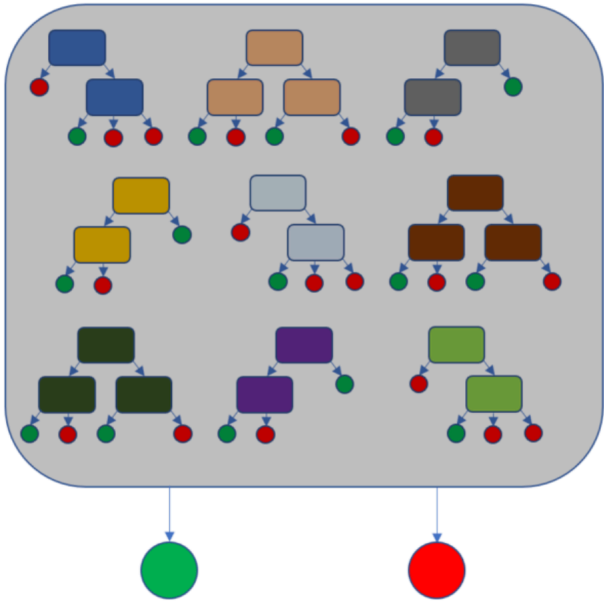


$$\begin{bmatrix} pc1 \\ pc2 \end{bmatrix} = \begin{bmatrix} 0.72 & 0.69 \\ -0.69 & 0.72 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Random Forests

- We already saw committee of classifiers as typical of *an ensemble approach that led to strong classification decisions* using component classifiers that in themselves may not perform well.
- We can also form **ensembles of decision trees** with the same goal in mind.
- One particularly successful decision tree committee is the **Random Forest**
  - See L. Breiman, Random forests, Machine Learning, vol 45, 2001, pp. 5–32
- As its name implies it is a ***collection of trees*** (a "forest") ***that are somehow random in their construction***.





Random Forests

- In common with other supervised classifiers, we assume that we have available a labelled set of ***training pixels***. However, those pixels ***are not used as a complete set*** as would be the case with single stage supervised classifiers.
- Instead, **bootstrapped samples** are used in the following manner.
  1. If there are K **pixels** in the training set, we **randomly select K with replacement** (in other words, the first training pixel is selected and then returned to the set; then a second pixel is selected, and so on). In this manner a certain number of training pixels will be be replicated in the bootstrapped sample chosen to train the first tree in the random forest.
  2. Using this training set a CART style decision tree is developed. Before that can happen though a decision has to be made as to the feature set that will be used to grow the tree. If there are N **features** in the pattern space (e.g. in the spectral domain) a small number $n \ll N$ is **selected randomly** for the first tree. That small set is used *at each node* in the CART process. Typically, the *Gini impurity* is employed to find the best split and the best one of the n features to use in that split.
  3. We then need to assess how well the (first) tree generalises. The classic approach is to have a separate testing set of pixels which would be run through the tree to see what errors are made. In the case of a random forest, however, those pixels in the original training set that are not picked up in the bootstrapped sample used to develop a particular tree can be used as testing pixels. It turns out that taking a sample K with replacement from the available pixels, leaves out about one third of the original set. They are the pixels that are used to check tree performance.

Random Forests

4. Clearly, the first tree, trained in this manner, would not be expected to perform well. As a consequence, a second tree is grown using another bootstrapped sample from the available training pixels along with a second random selection of features, but with the same dimension n as used in the growth of the first tree.
5. A third, a fourth and many other trees are developed in the same manner.
6. The newly grown trees are tested using the pixels from the original training set left over after the bootstrapped samples were chosen for training. We now have many trees capable of performing a classification on the same data set. In order to combine their results, we use a majority vote logic, in that the actual label allocated to an unknown pixel is given by the most favoured label among the trees. Sometimes this is called *modal logic*.

- The process just described is continued through the addition of as many randomly generated trees as necessary in order to reduce classification error and thus produce results with the desired accuracy.
  - It would not be uncommon for several hundreds to thousands of trees to be generated randomly in pursuit of such a goal.
- There are *two requirements* for the random forest methodology to work well.
  - First, the trees generated randomly have to be uncorrelated; the choice of the bootstrapped training sets (with replacement) provides that.
  - Secondly, the individual trees should be strong classifiers. Generally, classifier strength will increase with the number of features used at each decision node. However, that increases substantially the complexity of tree growth so that the number of features is nevertheless kept small and weak classifiers are generally used.

Random Forests