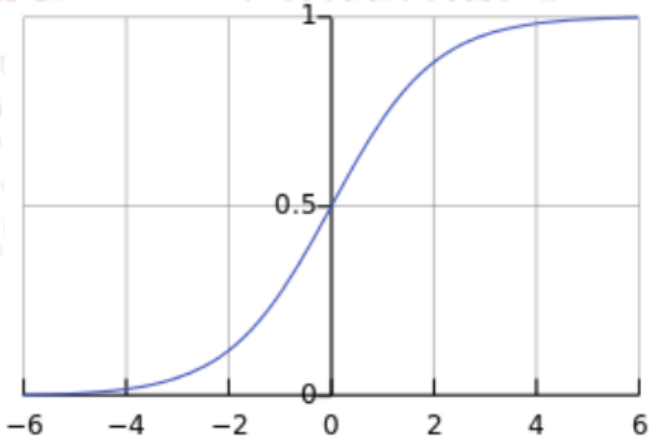
Section 2.3

Logistic Regression

Logistic Regression

- An extension of Linear Regression for environments where the dependent variable is categorical (1 or 0), for classification purporses.
- Predicts the probability of the outcome variable of being TRUE.
- Use of the logistic function, which produces a number between 0 and 1 (thus, interpretable as a probability).

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$$



Logistic Regression (2)

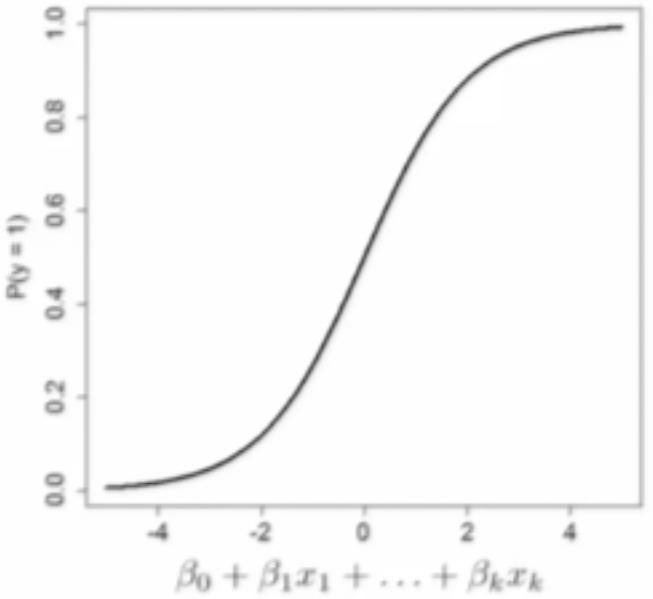
- Prediction of the probability of variable Y being TRUE:
  - **P (y = 1) and P (y = 0) = 1 – P(y = 1)**
- Let us see t as a linear combination of independent variables x_k.
- Then, the logistic function would be:

$$P(y=1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k)}}$$

Logistic Regression (3)

$$P(y=1) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_k x_k)}}$$

- Positive values are predictive for class 1
- Negative values are predictive of class 0



Logistic Regression (4)

- Another representation: Odds ratio

$$\text{Odds} = \frac{P(y=1)}{P(y=0)}$$

- Replacing probabilities by logistic functions and taking log:

$$\text{Odds} = e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_k x_k}$$

$$log(\text{Odds}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k$$

- Log(Odds) = Logit. The bigger Logit is, the bigger P(y = 1)

Logistic Regression (5)

- The outcome of a logistic regression model is a probability.
- In order to make a binay classification, we set a threshold value t.
- We convert probabilities to predictions:
  - If P(Y=1) >= t, then predict 1.
  - If P(Y=1) < t, then predict 0.
  - T chosen depending on the type of error we want.

Logistic Regression (6)

- Confusion matrix to see the predictive ability of the model.

	Predicted = 0	Predicted = 1
Actual = 0	True Negative (TN)	False Positive (FP)
Actual = 1	False Negative (FN)	True Positive (TP)

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN}$$

$$\text{Precision} = \text{Positive predictive value} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \text{Sensitivity} = \frac{TP}{TP + FN}$$

$$\text{Specificity} = \frac{TN}{TN + FP}$$

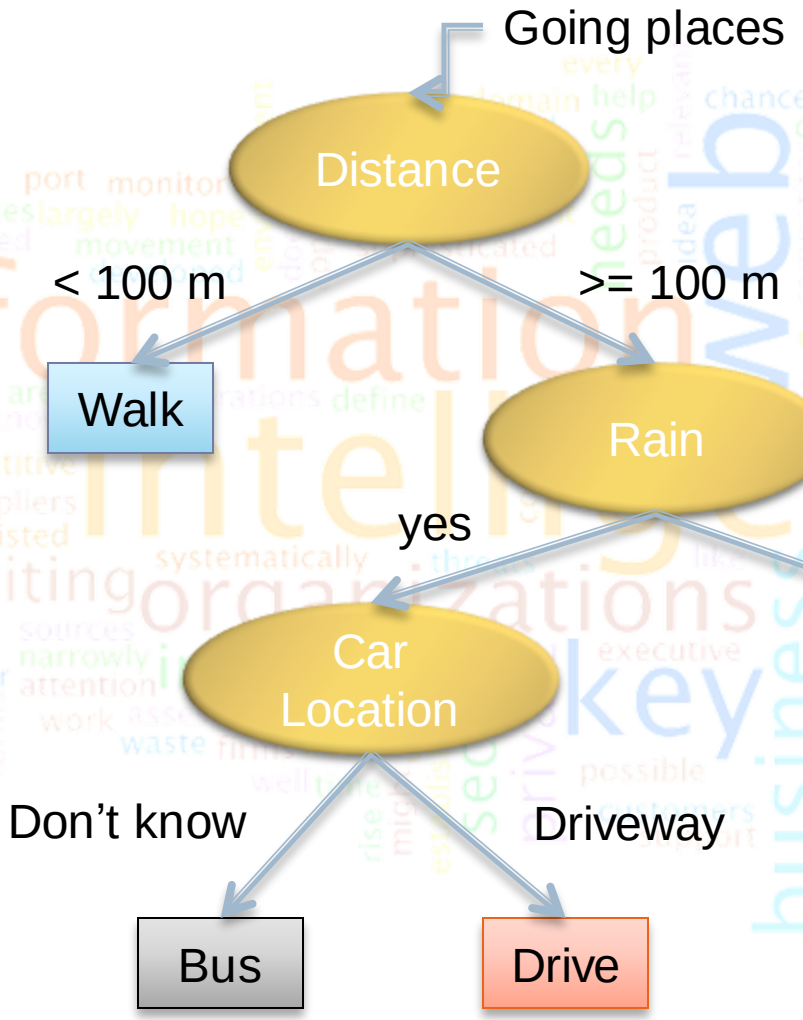
Section 2.4

Decision Trees

Decision Trees: How do they look like?

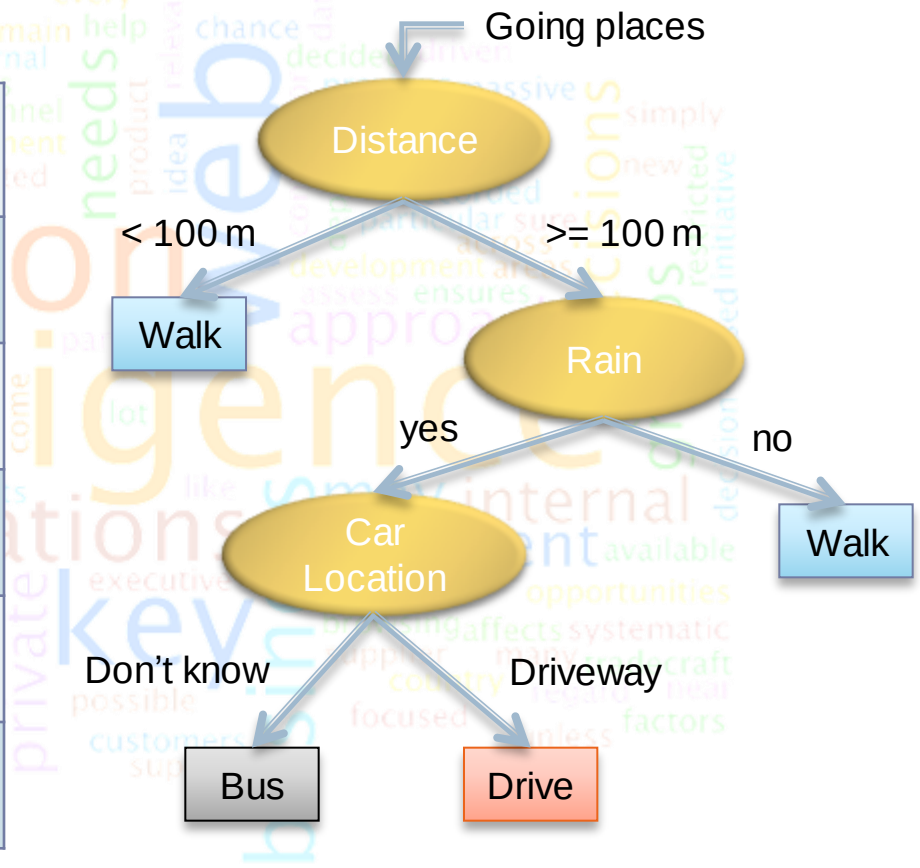
- Example:
  - **How to go to different places.**
    - By walk?
    - By car?
    - By bus?
  - **Parameters:**
    - Wheather
    - Travel distance, …
  - **The data:**
    - A survey over different persons.

Decision Trees: How do they look like?



Programs that build decision trees

Yes	45	…	…	Walk
No	125	…	…	Walk
Yes	190	…	…	Drive
Yes	…	…	…	…
Yes	…	…	…	…
No	…	…	…	…



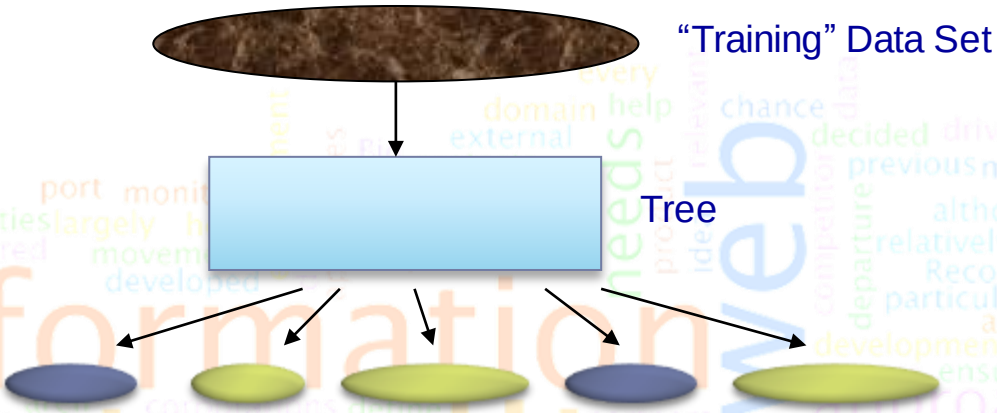
Types of Trees

- **Classification Trees:** : Assign the **non-numeric (or discrete)** target value to each record. For example, tomorrow's weather can be classified (predicted) as one of
  - "sunny“
  - "cloudy“
  - or "rain".
- **Regression Trees:** Estimate the **numeric** target value for each record. For example, tomorrow's maximum temperature would be 27º C.

Types of Trees (2)

- All of these trees have the **same basic structure**
- However, there are a **variety of algorithms for building tree-based models**
- Some of the most popular decision tree algorithms are:
  - ID3 (Quinlan, 1986)
  - CHAID
  - CART
  - C4.5
  - See5/C5.0, etc.
- CART and See5/C5.0 are **widely accepted as some of the best algorithms** for constructing tree-based models for data

Basic Tree Builder



- Trees are constructed by **repeatedly partitioning the training data set** into many **subsets**
- The aim is to **identify subsets** each of which contains **cases with one target value**
- In other words, we want to find subsets such that they are ***purer*** (have less *diversity*) than the original data set

Basic Tree Builder (2)

- **Measuring Purity:** The term **entropy** (used in information theory) indicates the **impurity** of an arbitrary collection of cases (examples)

$$Entrophy(S) = -(p_{pos}) \log_2(p_{pos}) - (p_{neg}) \log_2(p_{neg})$$

- For a Boolean function (target is up or down), where, S is a sample of training cases, (p_{pos}) is a proportion of positive cases (say up values), (p_{neg}) is a proportion of negative cases (say down values)
  - Example:

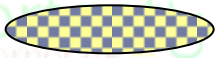
Entropy([15 up, 4 down]) = $-(15/19)\log_2(15/19) - (4/19)\log_2(4/19)$
= 0.742

Basic Tree Builder (3)

- The entropy (impurity) is **0** if **all** members of S belong to the same class (up or down)

or

  - The entropy (impurity) is 1 if S contains an equal number of positive (up) and negative (down) cases



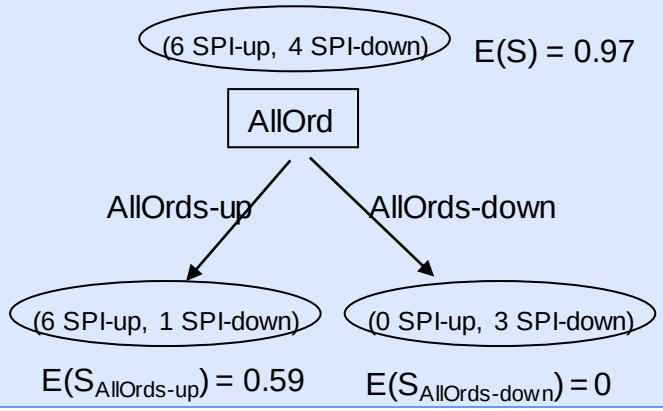
  - If S contains unequal numbers of positive and negative cases, the entropy is between 0 and 1

Basic Tree Builder (4)

- **Information Gain** measures the expected **reduction in entropy** (impurity) when we partition S using an attribute A

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

- **Gain(S, A) = Entropy(S) - weighted average of entropies (impurities) of Subsets**



Gain(S, AllOrd) = E(S) - (7/10)*E($S_{AllOrds-up}$)
- (3/10)*E($S_{AllOrds-down}$)
= 0.97 - (0.7)*(0.59)
- (0.3)*(0)
= 0.557

Basic Tree Builder (5)

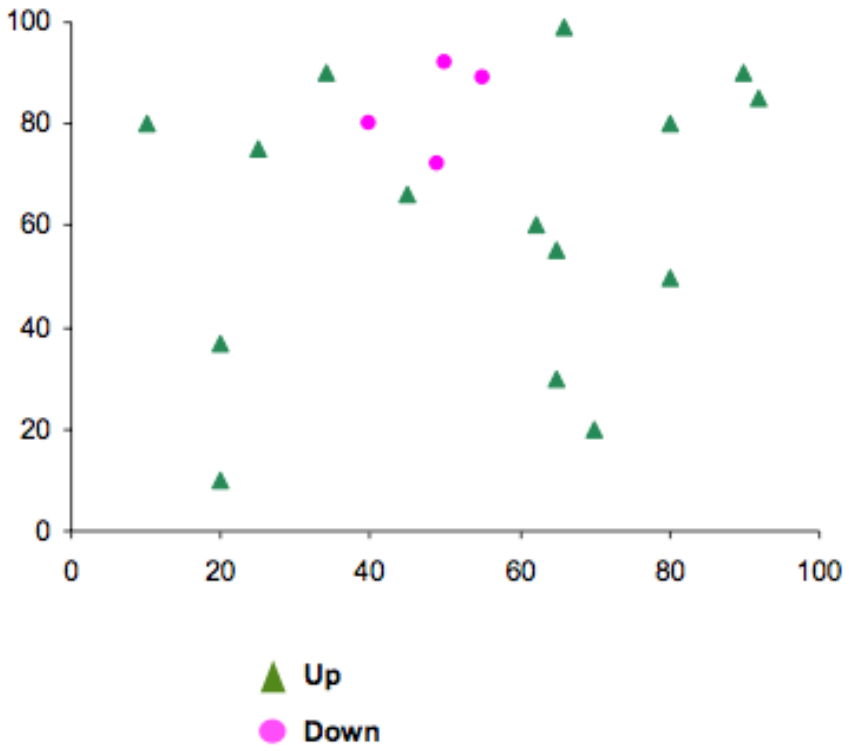
- **Calculate** the information gain for every attribute
- **Select** the attribute that produces **maximum information gain** and partition the data using that attribute
- **Terminate** if a "pure" node is reached or no attribute increases information gain

A Simple Example

- We can visualize the following simple data set using a 2-dimensional graph on the right hand side

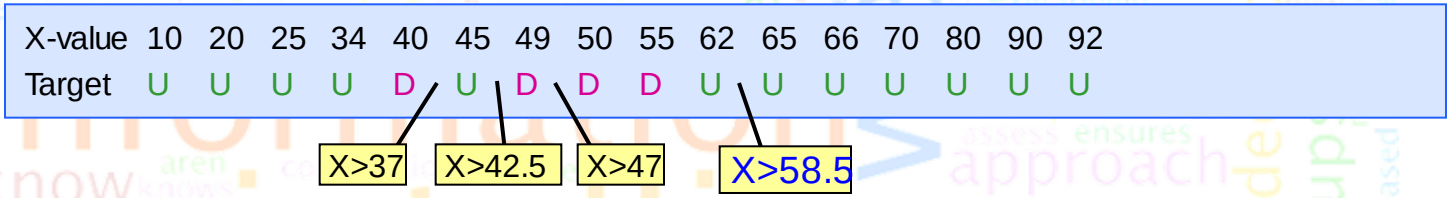
Training Data Set (S)

X-value	Y-value	Target
20	10	Up
70	20	Up
65	30	Up
20	37	Up
80	50	Up
65	55	Up
62	60	Up
45	66	Up
49	72	Down
25	75	Up
10	80	Up
40	80	Down
55	89	Down
90	90	Up
34	90	Up
50	92	Down



A Simple Example (2)

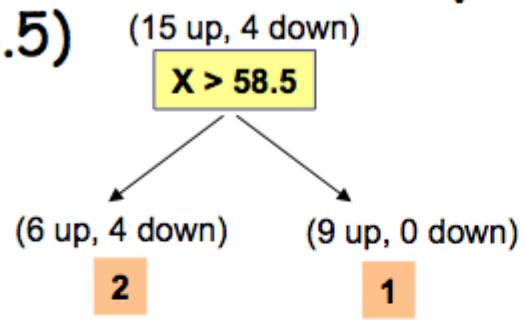
- Calculate the information gain for every possible split position of the attribute X
  - possible split positions for the attribute X

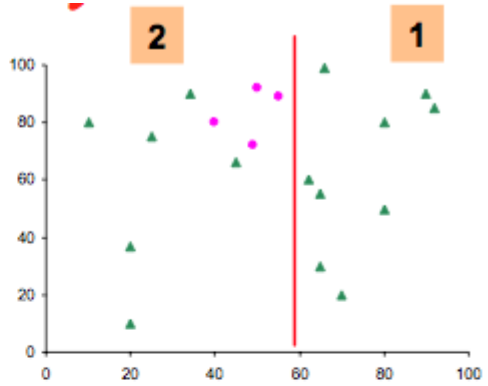


- Similarly, calculate the information gain for every possible split position of the attribute Y
- The **attribute with maximum information gain** (the split position with max information gain) is selected and the data set S is partitioned accordingly

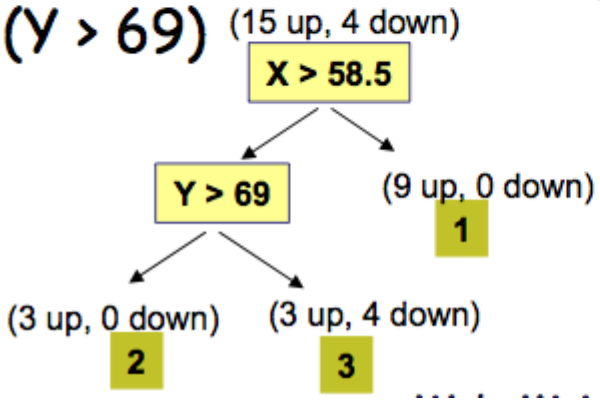
A Simple Example (3)

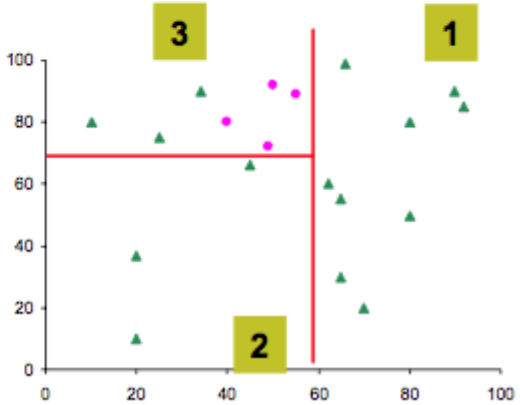
The tree after the **first** split for (X > 58.5)



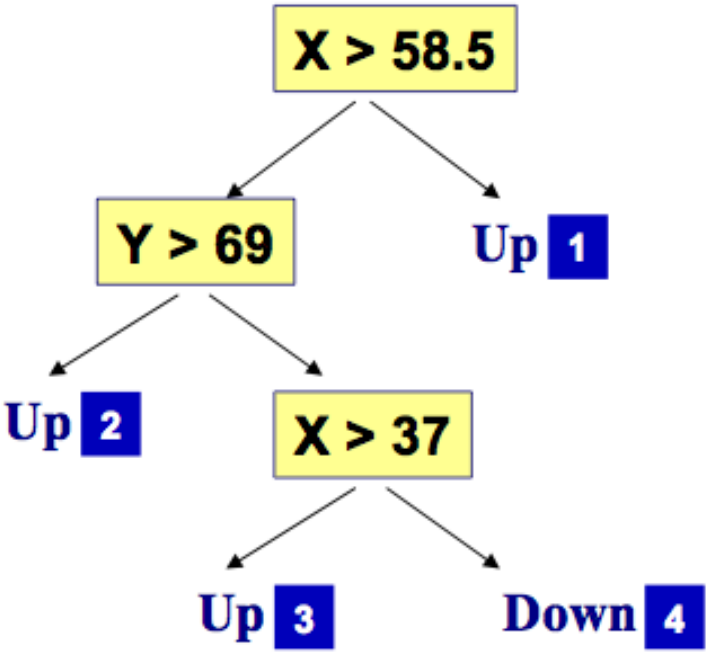


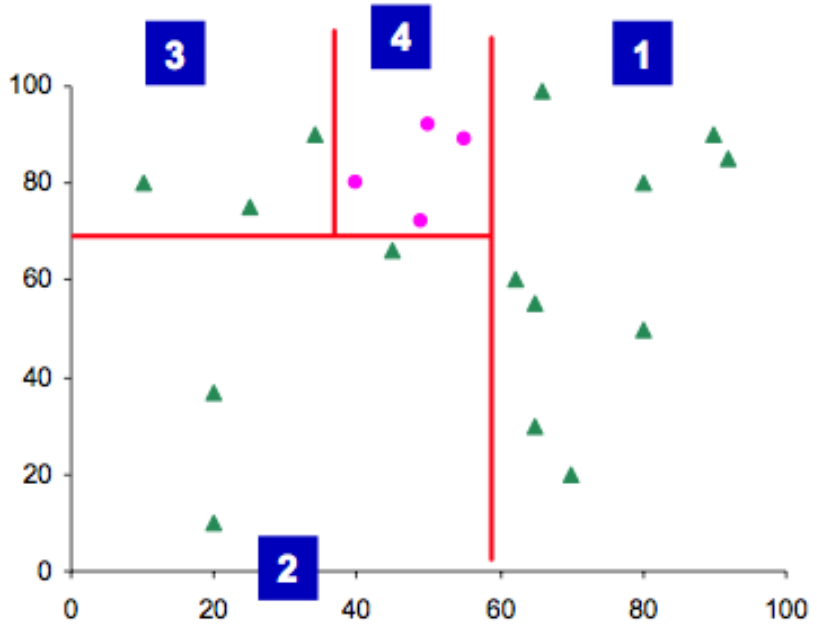
The tree after the **second** split for (Y > 69)





A Simple Example (4)





Rule-1: IF (X > 58.5) THEN Up
Rule-2: IF (X <= 58.5) AND (Y <= 69) THEN Up
Rule-3: IF *(X <= 58.5)* AND (Y>69) AND (X <= 37) THEN Up
Rule-4: IF (X <= 58.5) AND (Y>69) AND (X > 37) THEN Down

What If the Target is Numeric?

- Each leaf node stores either
  - Mean value of cases in the leaf, or
  - A **multivariate linear model** for the cases at that leaf, and this model is used to predict the value
- Example:

```
IF (X <= 38.5) THEN LM1
IF (X >  38.5) AND (Y <= -14 )
     THEN LM2
IF (X >  38.5) AND (Y > -14 )
     AND (Y <= 1.5 ) THEN LM3
IF (X >  38.5) AND (Y > -14 )
     AND (Y > 1.5 ) THEN LM4
```

Models at the leaves:

LM1: A = 2.74 – 0.026Y

LM2: A = 30.7 – 0.362Y

LM3: A = –884 + 0.318Z – 0.253Y

LM4: A = 15.8 – 1.22Y

What If the Target is Numeric? (2)

- Splitting criterion used is **standard deviation** of the target values of cases in the node (as a **measure of the error**)
- The expected **error reduction** (standard deviation reduction) can be calculated using:

$$\Delta error = sdev(S) - \sum_i \frac{|S_i|}{|S|} * sdev(S_i)$$

where S_i are sets that result from splitting

- The attribute which maximizes the expected error reduction is selected

Issues for a Tree Builder

- Selecting a **splitting criterion**. Few different criteria are available:
  - Information Gain, Gini Index, Gain Ratio, etc.
- Number of splits allowed at each level of the tree (2 or 3 or ….)
- Depth / height of the tree
- Avoid **over-fitting** the training data
- Handle **missing values** for attributes
- Estimate error on new (unseen) data

Avoiding Over-Fitting

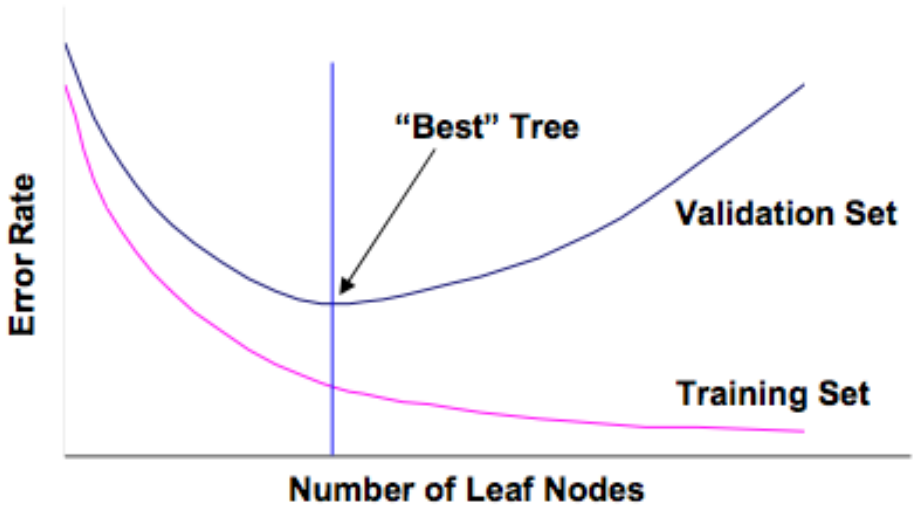
- Selecting the “best” tree:

Data Set is divided in to three sets:

- Training
- Validation

} used for model building

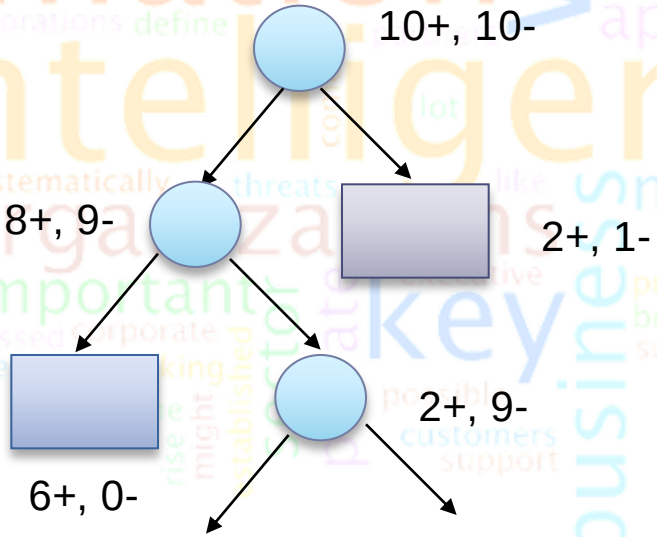
- Test



- **Occam’s razor**: Prefer the simplest hypothesis (i.e., model) that fits the data

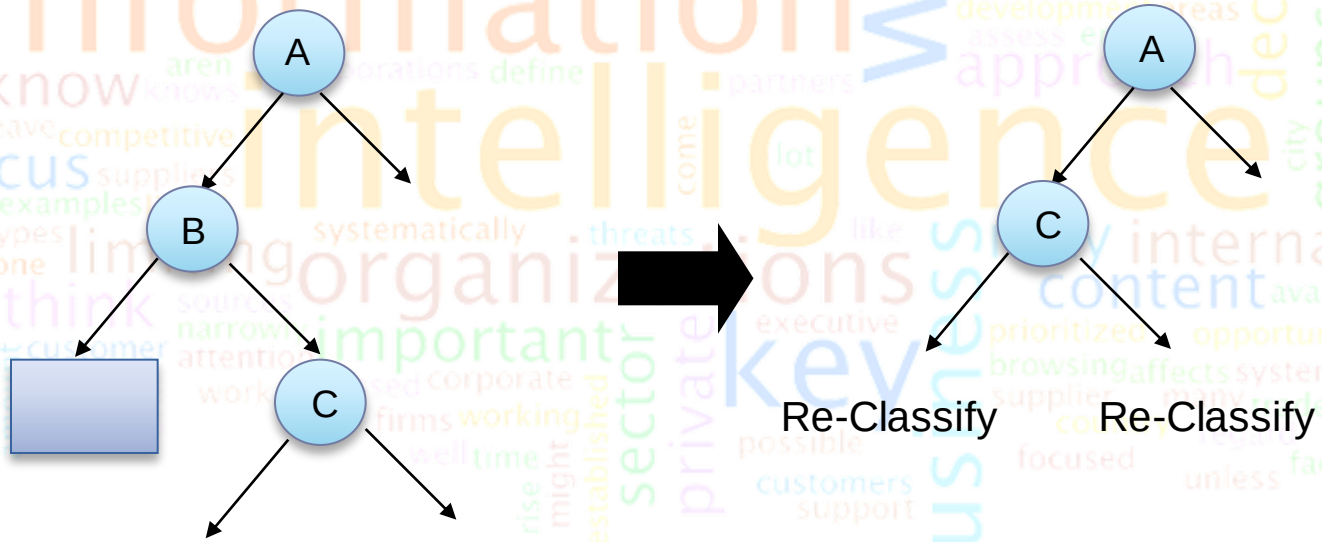
Over-fitting Avoidance by "Pruning"

- **"Forward" pruning**:
  - Stop when the number of cases in a node reach some pre-defined value (or use some statistical test like χ^2)



Over-fitting Avoidance by "Pruning" (2)

- **"Backward" pruning**:
  - Replace nodes by leaves
- Sub tree lifting:



Computational Complexity

- **Assume**: N instances each with M attributes and tree depth **O(log N)**
- At each tree depth, we have to consider, for each attribute, the classification of all N instances. The overall cost for building the tree is thus: **O(MN log N)**
- When pruning, each node has to be considered for replacement by a leaf. The tree can have at most N leaves. For binary trees this would mean at most 2N-1 nodes i.e. O(N). For sub tree lifting, each node is considered for replacement: this is O(N). For each replacement an instance may have to be re-classified at each level of the tree. This is **O(N log N)** reclassifications. A reclassification requires at most **O(log N)** operations.
- **Total cost:** **O(MN log N) + O(N (log N)^2)**

Section 2.5

Support Vector Machines

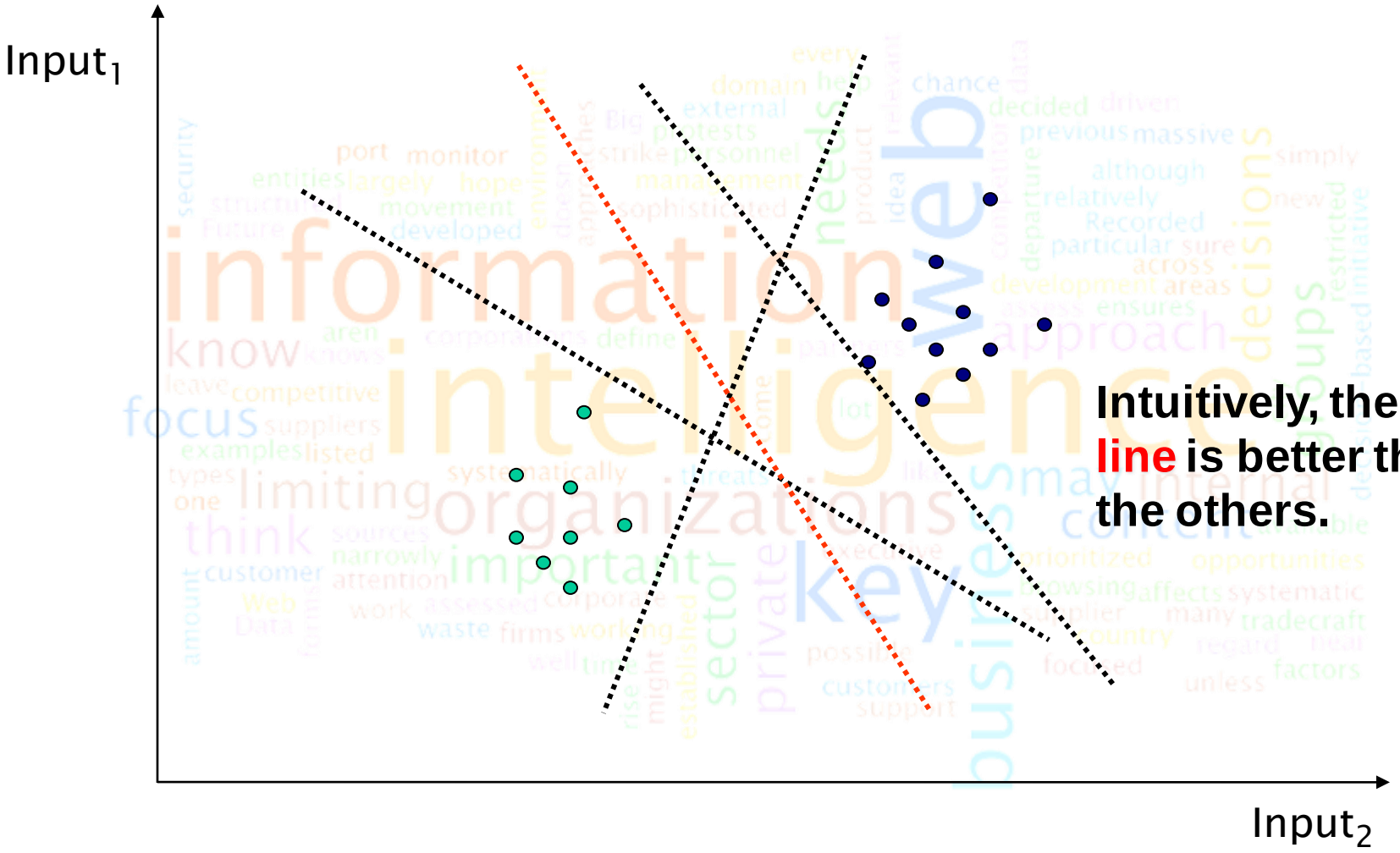
Support Vector Machines

- An **effective machine learning technique**; one the **most important recent discovery** in machine learning
- SVMs are based on Vapnik's work and were introduced in early '90s.
- Decision surface for classification is a **hyperplane** in the **feature** space (*a line in 2D case*).
- **Convert problems into linearly separable problems** by *transforming the input space into a higher dimensional feature space*.

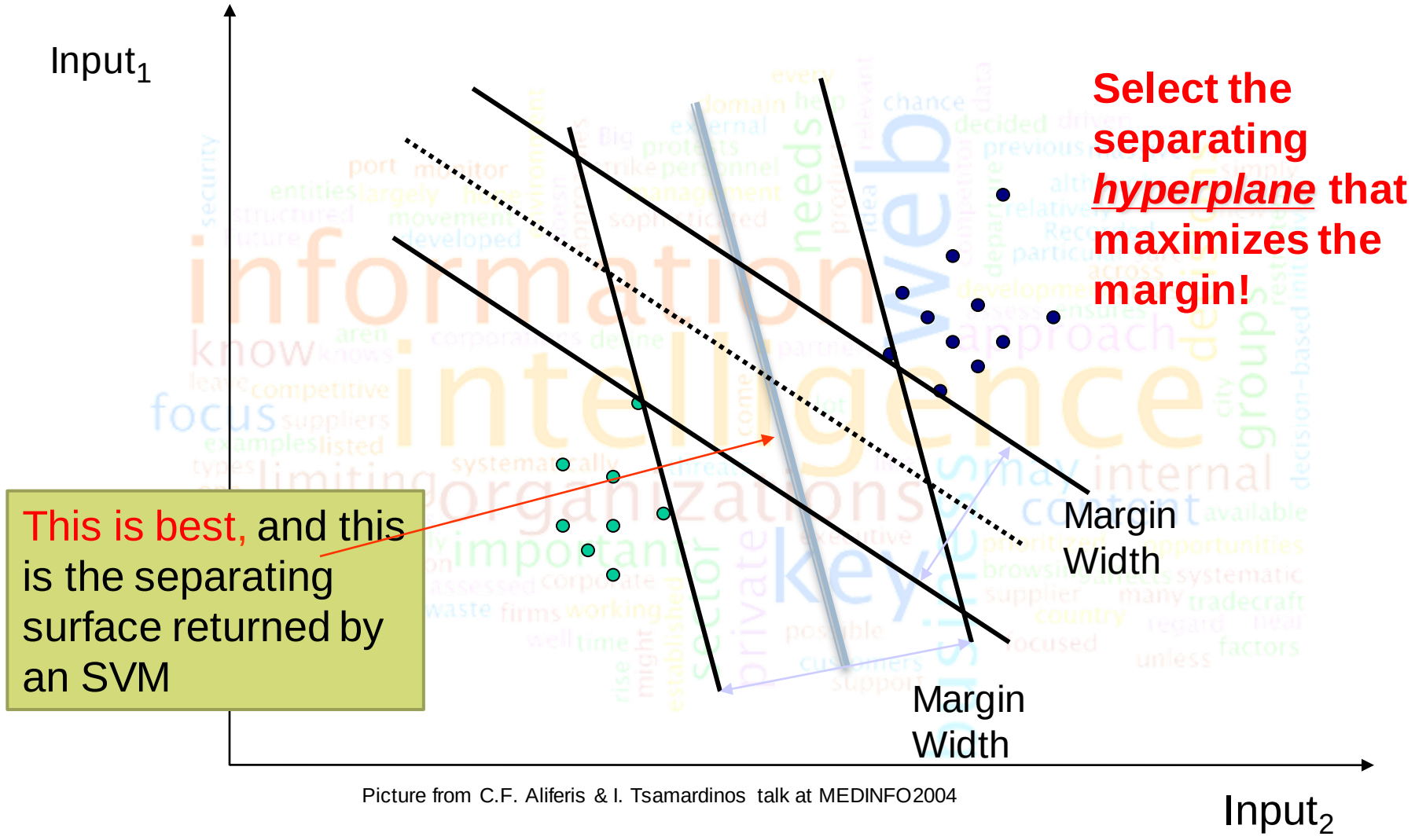
Basic ideas

- Map data onto a high dimensional space via a *kernel function* where *data can be classified with linear decision surfaces*
- Find the "optimal" hyperplane that *maximizes the degree of separation between the two classes* (*maximizes the margin* between the two classes)
- If data are not linearly separable in a given space *find the hyperplane that maximizes the margin and also minimizes the number of misclassifications*

Which Separating Hyperplane to Use?

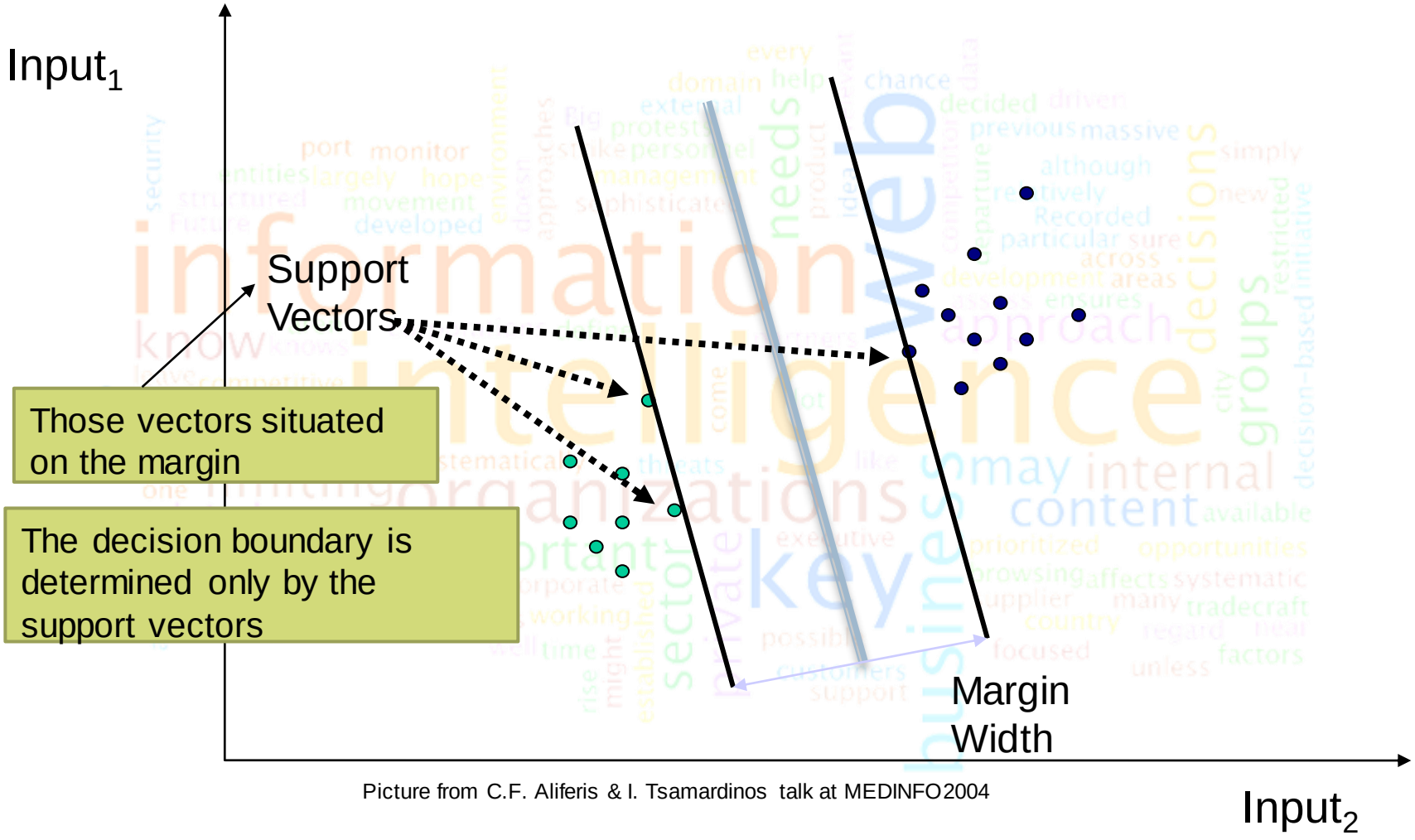


Maximize the Margin



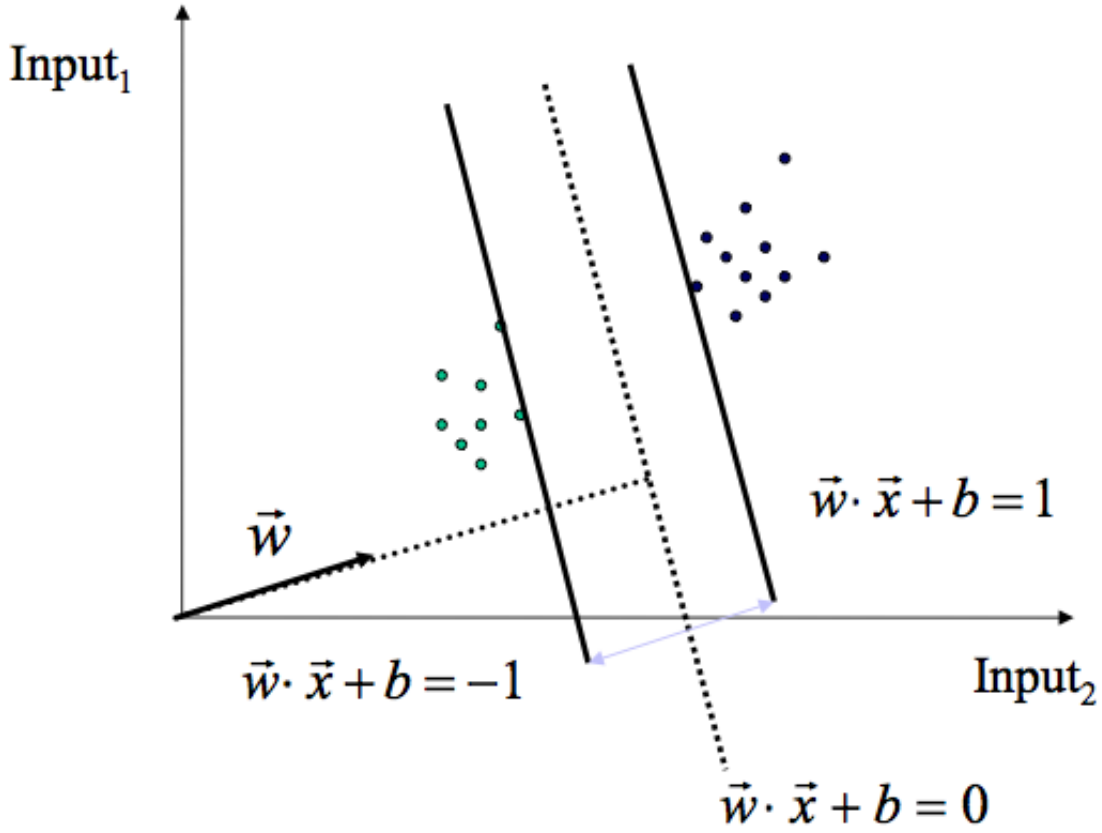
Picture from C.F. Aliferis & I. Tsamardinos talk at MEDINFO2004

What are Support Vectors?



Picture from C.F. Aliferis & I. Tsamardinos talk at MEDINFO2004

Formulation of the Optimization Problem



$$\max \frac{2}{\|w\|}$$

$$s.t.\ (w\cdot x+b)\geq 1,\ \forall x \text{ of class 1}$$

$$(w\cdot x+b)\leq -1,\ \forall x \text{ of class 2}$$

The distance between the origin and the line wx+b=k is k/||w||, where k=1, or -1.

Then *margin* = 2/||w|| and we should maximize it.

Formulation of the Optimization Problem (2)

- If class 1 is “1” and class 2 is “-1” we have:

$$(w * x_i + b) \geq 1, \quad \forall x_i \; with \; y_i = 1$$
$$(w * x_i + b) \leq -1, \quad \forall x_i \; with \; y_i = -1$$

$$y_i(w * x_i + b) \geq 1, \forall x_i$$

- So the problem becomes:

$$\max_{y_i(w*x_i+b)\geq 1, \forall x_i} \frac{2}{\|w\|} \quad \text{or} \quad \min_{y_i(w*x_i+b)\geq 1, \forall x_i} \frac{1}{2}\|w\|^2$$

Linear, Hard-Margin SVM Formulation

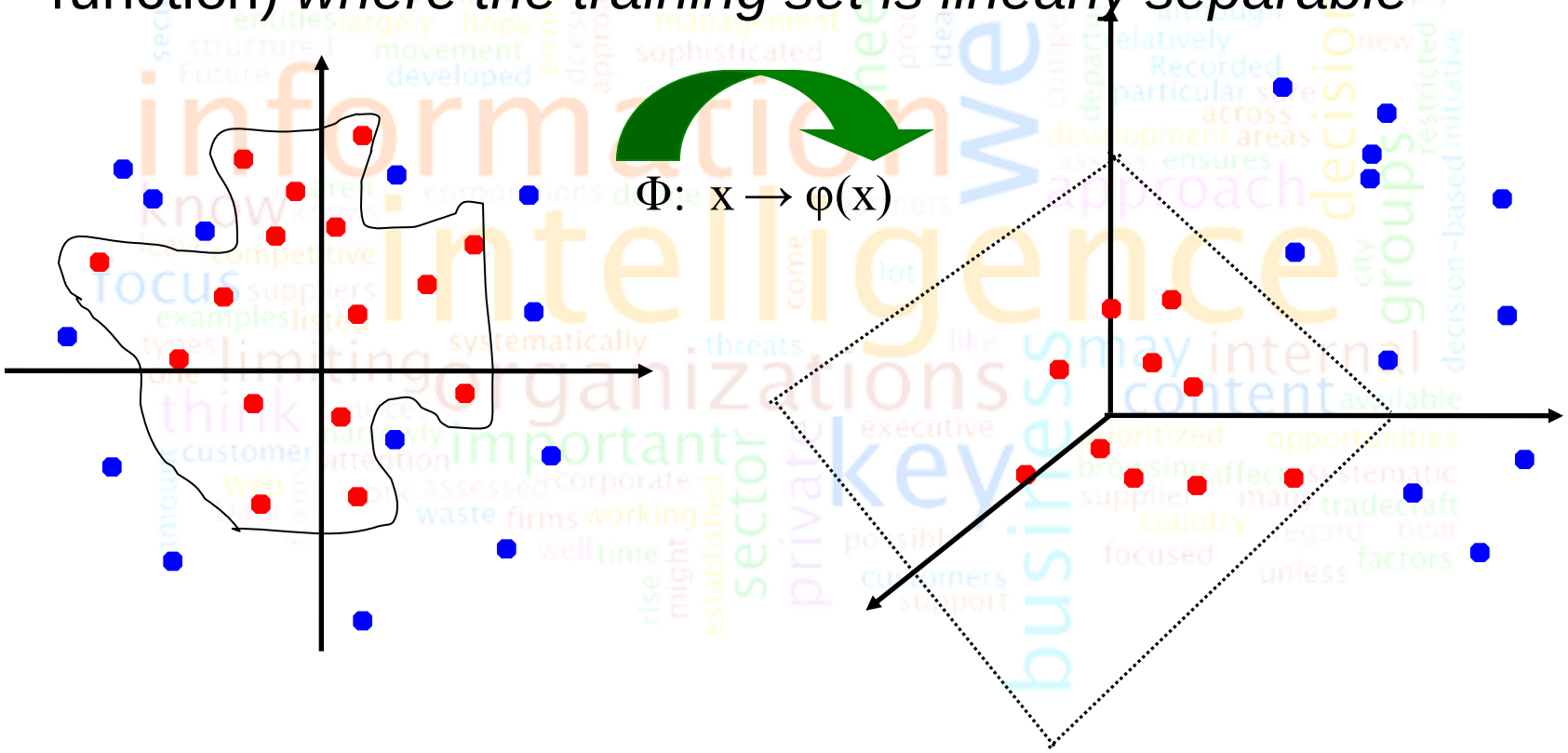
- Find *w,b* that solves

$$\min_{y_i(w*x_i+b)\geq 1, \forall x_i} \frac{1}{2}\|w\|^2$$

- Problem is convex so, there is a unique global minimum value (when feasible)
- There is also a unique minimizer, i.e. *weight and b value that provides the minimum*
- *What if the data is not linearly separable?*
  - Answer: We may *allow some examples to fall within the margin but penalize them* (using a so called soft margin); or see next slide.

How to solve non-linearly separable problems?

- Basic idea: the original input space is mapped to some higher-dimensional feature space (using a kernel function) *where the training set is linearly separable*



Why do SVMs work?

- **The feature space is often very high dimensional**. Why don't we have the curse of dimensionality?
- *A classifier in a high-dimensional space has **many parameters and is hard to estimate**.*
- Vapnik argues that the fundamental problem is not the number of parameters to be estimated. Rather, the problem is about the **flexibility of a classifier**

From C.F. Aliferis & I. Tsamardinos talk at MEDINFO2004

Why do SVMs work?

- The flexibility of a classifier should not be characterized by the number of parameters, but by the flexibility (capacity) of a classifier, formalized by the "VC-dimension" of a classifier.
- The **higher the VC-dimension**, the **more flexible** a classifier is
- In practice**, the VC-dimension may be difficult to be computed exactly**

Structural Risk Minimization (SRM) problem

- SRM means **we should find a classifier that minimizes the sum of training error**
  - Called empirical risk
- As well as a term that is a **function of the flexibility of the classifier**
  - Called model complexity

Choosing the Kernel Function

- **It is the most tricky part of using SVM**. It is *equivalent to choosing the number of hidden nodes for a neural network*.
- In practice, **start with a low degree polynomial kernel function** or **RBF kernel function with a reasonable width**
- Note that **SVM with RBF kernel is closely related to RBF neural networks**, where the centers of the radial basis functions are automatically chosen for SVM

Advantages of SVMs

- Training is relatively easy - No local optima
  - (very good compared to neural networks)
- It scales relatively well to high dimensional data
- The tradeoff *between classifier complexity and error*
  - (empirical risk) can be controlled explicitly
- But we need to choose a "good" kernel function!!!
  - This may not be so easy.

Section 2.6

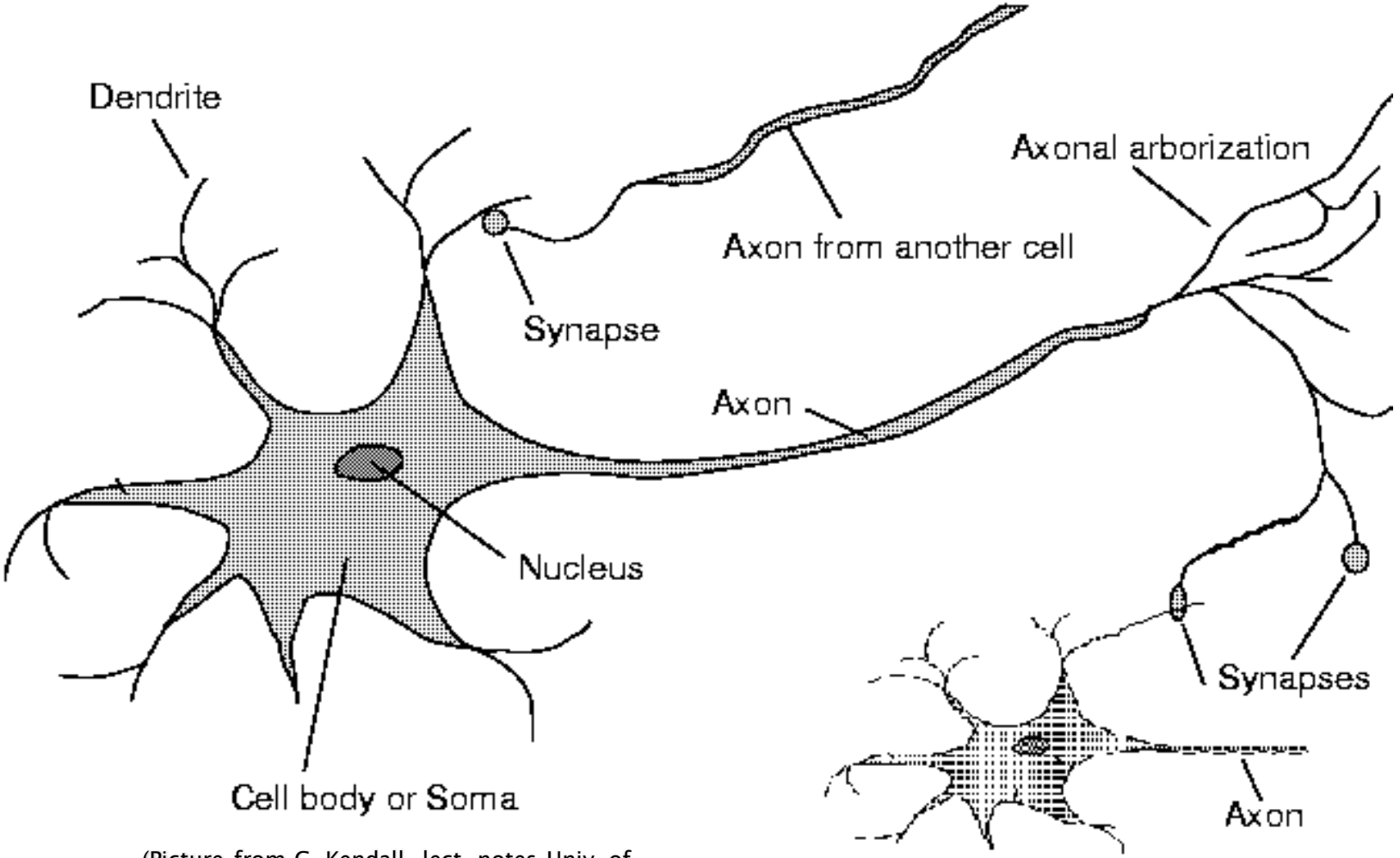
Artificial Neural Networks

Inspired on a biological model ...

What is connectionism/neural networks?

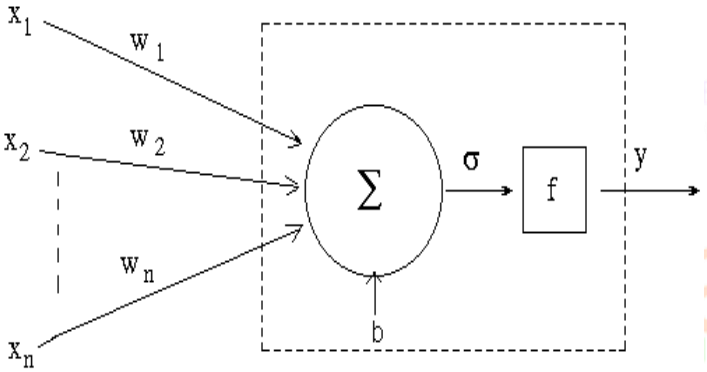
- A **simplified model** of how natural neural systems work. Neural Networks (NNs) **simulate natural** information processing tasks from **human brain**.
- A NN model consists of **neurons and connections between neurons** (synapses).
- Characteristics of Human Brain:
  - **It contains 10^{11} neurons and 10^{15} connections**
  - **Each neuron may connect to other 10,000 neurons.**
  - **Human can perform a task of picture naming in about 500 miliseconds**

Biological Neural Networks

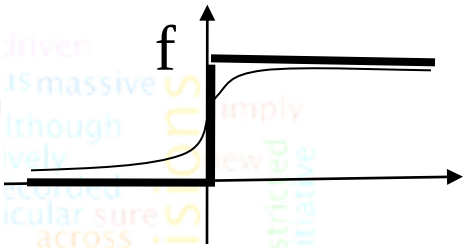


(Picture from G. Kendall, lect. notes Univ. of Nottingham)

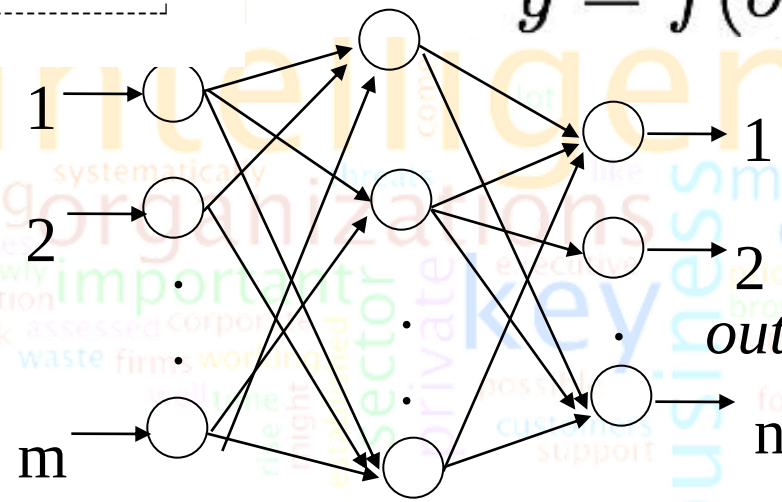
What is an artificial neural network?



$$\sigma = \sum_{i=1}^{n} w_i x_i + b$$



$$y = f(\sigma)$$



$$f : \Re^m \rightarrow \Re^n$$

Artificial Neural Nets (ANNs)

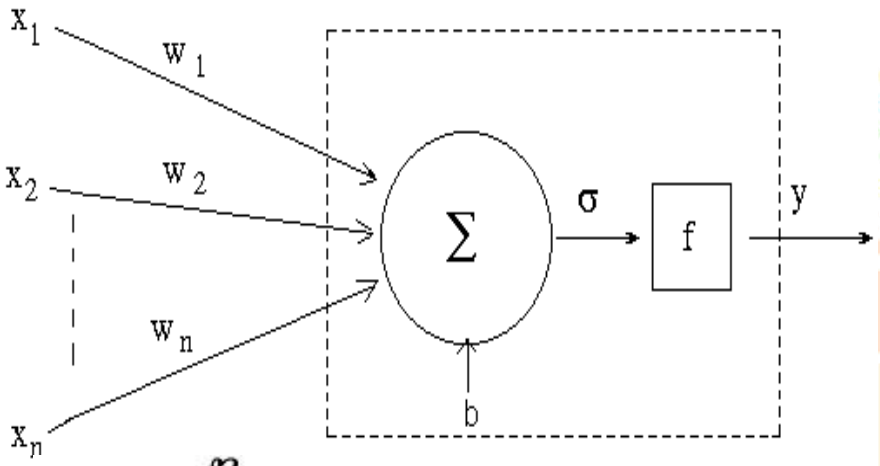
- An **Artificial Neural Network** is an **interconnected assembly** of simple processing elements, **units or nodes** (neurons), whose functionality is inspired by the **functioning of the natural neuron from brain**.
- The processing ability of the neural network is stored in **the inter-unit connection** strengths, **or weights**, obtained by a **process of learning from a set of training patterns**.

Artificial Neural Nets (ANNs)

- The **units** (individual neurons) **operate only locally** on the inputs they receive via connections.

- ANNs undergo some sort of **"training"** whereby the connection **weights are adjusted on the basis of presented data**. In other words, **ANNs "learn" from examples** *(as children learn to recognize dogs from examples of dogs)* and exhibit some **generalization capability** *beyond the training data* (for other data than those included in the training set).

Formal neuron (McCulloch & Pitts)

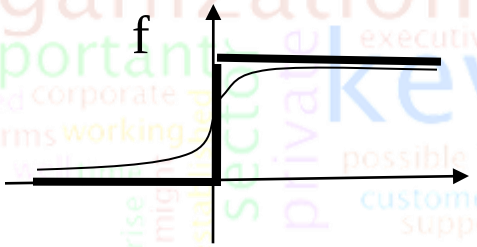


A single neuron has 6 components:

1. Input "x"
2. Weights "w"
3. Bias "b" (Threshold = -b)
4. Activation function "f"
5. Input function σ
6. Output "y"

$$\sigma = \sum_{i=1}^{n} w_i x_i + b$$

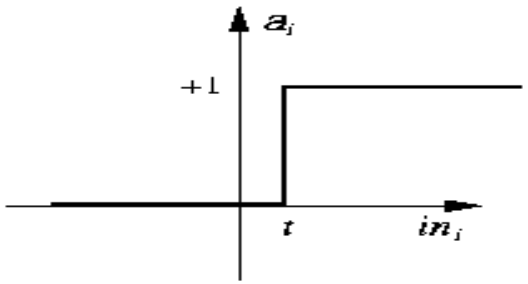
$$y = f(\sigma)$$



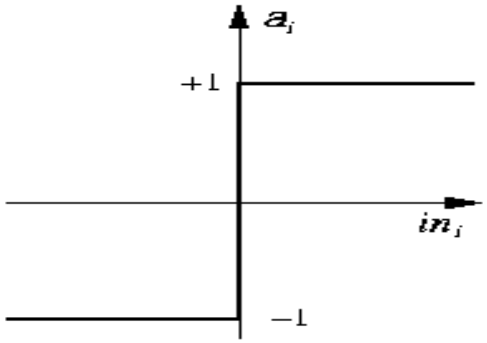
$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

McCulloch & Pitts (1943) recognised as the designers of the first neuron (and neural network) model.

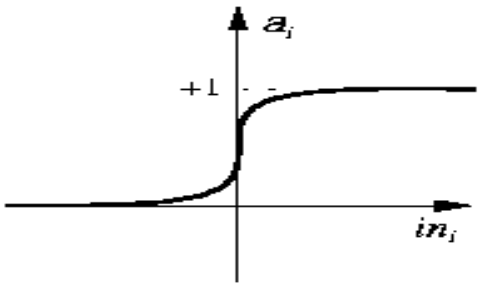
Formal neuron - Activation Functions



Step function



Sign function



Sigmoid function

$$Step_t(x) = \begin{cases} 1 & : \quad x \geq t \\ 0 & : \quad x < t \end{cases}$$

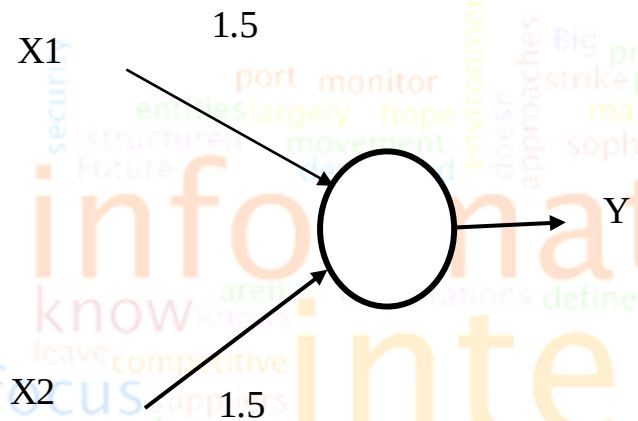
$$Sign(x) = \begin{cases} +1 & : \quad x \geq 0 \\ -1 & : \quad x < 0 \end{cases}$$

$$Sigmoid(x) = \frac{1}{1 + e^{-kx}}$$

$$Id(x) = x$$

$$Lin(x) = k * x$$

A neuron which implements OR function



OR

X1	X2	Y
1	1	1
1	0	1
0	1	1
0	0	0

Threshold = 1

Bias = -1 (Threshold = - Bias)

Mayor clases of Neural Networks

- Backpropagation Neural Networks
  - Supervised learning
- Kohonen Self Organizing Maps
  - Unsupervised learning
- Hopfield Neural Networks
  - Recurrent neural networks
- Radial Basis Function Neural Networks (RBF)
- Neuro-Fuzzy Networks (NF)
- Others: various architectures of recurrent neural networks,
  - Networks with dynamic neurons,
  - Networks with competitive learning, etc.

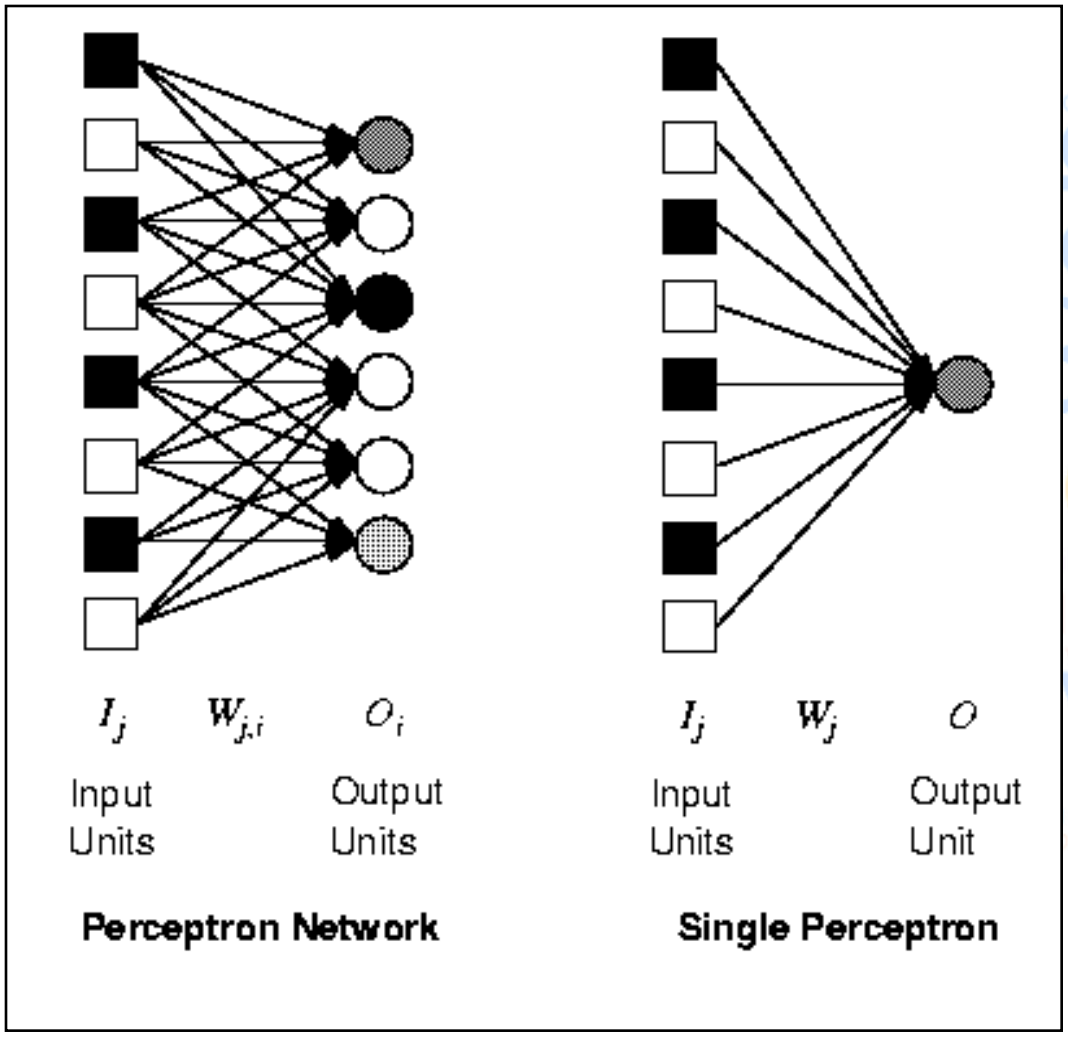
History of NN

- McCulloch & Pitts (1943)
  - Neural networks and artificial intelligence were born, first well-known model for a biological neuron
- Hebb (1949)
  - Hebb learning rule
- Minsky (1954)
  - Neural Networks (PhD Thesis)
- Rosenblatt (1957)
  - Perceptron networks (Perceptron learning rule)
- Widrow and Hoff (1959)
  - Delta rule for ADALINE networks
- Minsky & Papert (1969)
  - Criticism on Perceptron networks (problem of linear separability)
- Kohonen (1982)
  - Self-Organizing Maps

History of NN (II)

- Hopfield(1982)
  - Hopfield Networks
- Rumelhart, Hinton & Williams (1986)
  - Back-Propagation algorithm
- Broomhead & Lowe (1988)
  - Radial Basis Functions networks (RBF)
- Vapnik (1990)
  - Support Vector Machine approach
- In the ’90s
  - Massive interest in neural networks, many NN applications were developed
  - Neuro-Fuzzy networks emerged

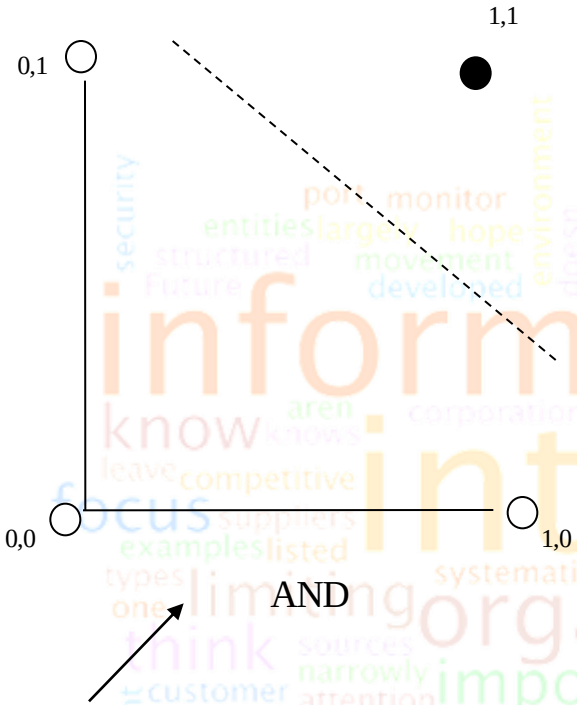
Perceptron

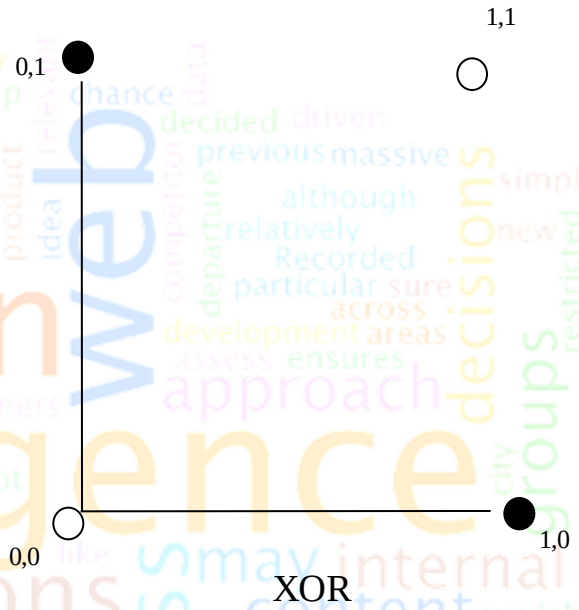


- Synonym for Single-Layer, Feed-Forward Network.
- First Studied in the 50's (Rosenblatt).
- Other networks were known about but the Perceptron was the only one capable of learning and thus all research was concentrated in this

(from G. Kendall, lect. notes Univ. of Nottingham)

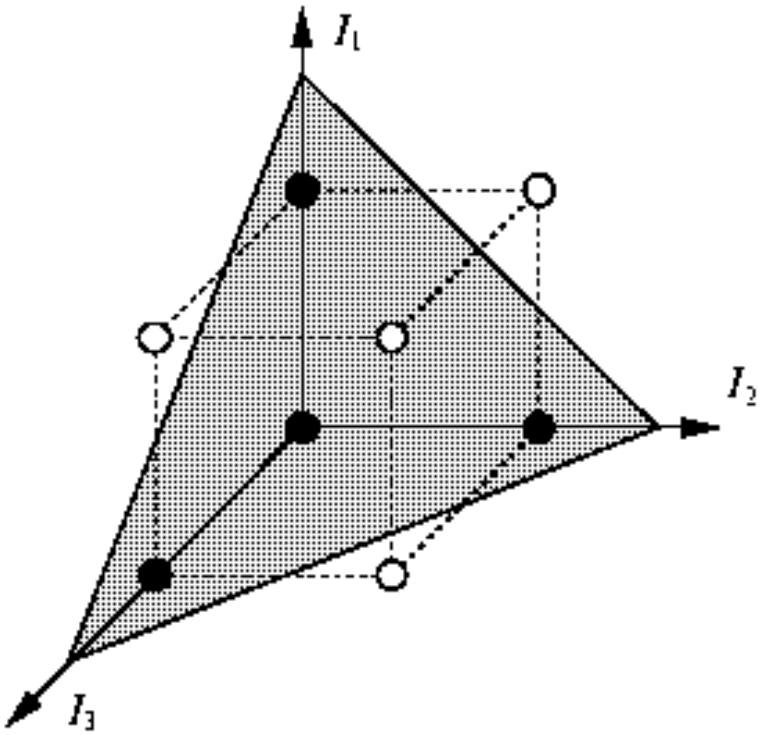
What can perceptrons learn?



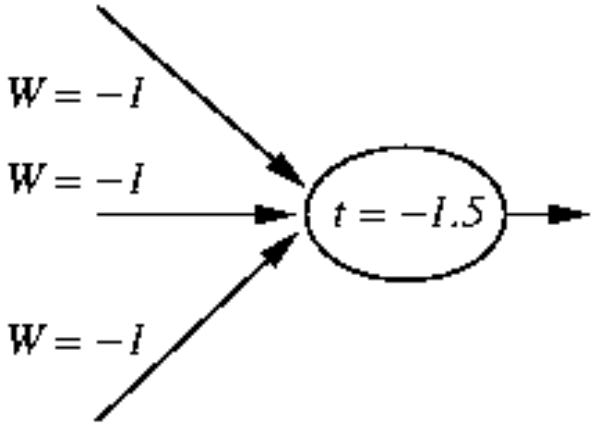


- Functions that can be separated in this way are called *Linearly Separable* (XOR is not Linearly Separable).
- A PERCEPTRON can learn (represent) only Linearly Separable functions.

What can perceptrons represent?



(a) Separating plane



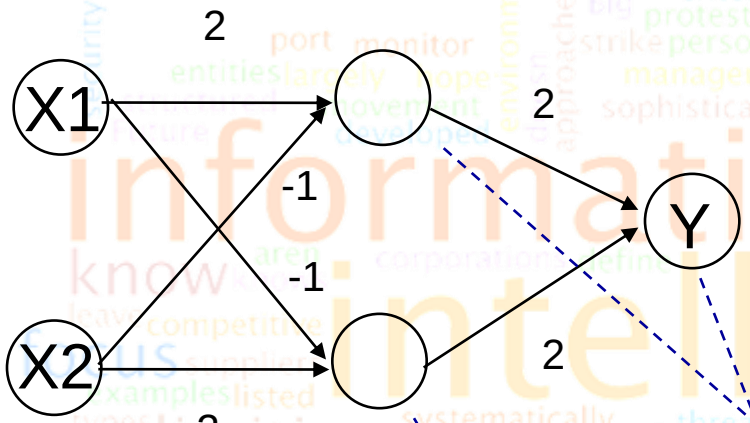
(b) Weights and threshold

Linear Separability is also possible in more than 3 dimensions – but it is harder to visualize

(from G. Kendall, lect. notes Univ. of Nottingham)

XOR problem – Not linearly separable

- One neuron layer is not enough, we should introduce an intermediate (hidden) layer.



Threshold for all nodes = 1.5

XOR

X1	X2	Y
1	1	0
1	0	1
0	1	1
0	0	0

$Y = X_1$ XOR $X_2 = (X_2$ AND NOT $X_1)$ OR $(X_1$ AND NOT $X_2)$

Training a Perceptron

AND

X1	X2	D
1	1	1
1	0	0
0	1	0
0	0	0

Training Dataset { (x(i), d(i)), i=1,…,p}

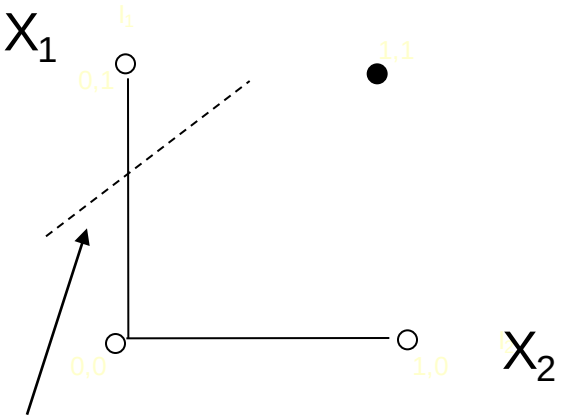
p = 4

Training set = { ((1,1),1), ((1,0),0), ((0,1),0), ((0,0),0) }

The training technique is called Perceptron Learning Rule.

Perceptron Learning

After weight initialization
(First Epoch)



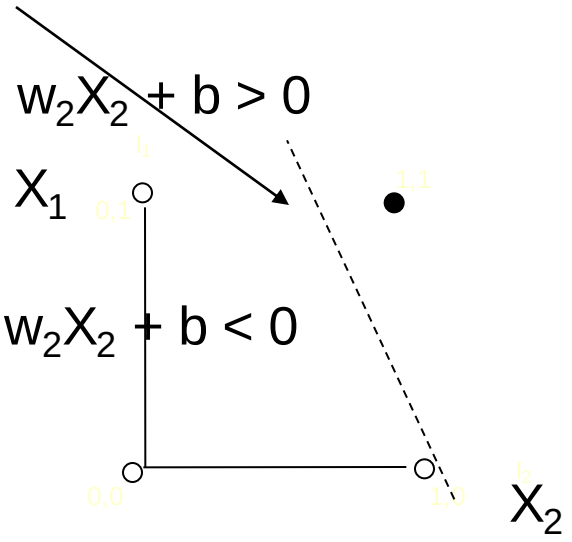
Separation line

$w_1X_1 + w_2X_2 + b = 0$

$w_1X_1 + w_2X_2 + b > 0$

$w_1X_1 + w_2X_2 + b < 0$

At Convergence



Training a Perceptron: Main Idea

- Vectors from the training set are presented to the **Perceptron network one after another** (cyclic or randomly):

  $(\underline{x}(1), d(1)), (\underline{x}(2), d(2)), \ldots, (\underline{x}(p), d(p)),$
  $(\underline{x}(p+1), d(p+1)), \ldots$

- **If the network's output is correct, no change is made**.
- Otherwise, the **weights and biases are updated** using the ***Perceptron Learning Rule***.
- **An entire pass** through all of the input training vectors is called an **Epoch**.
- When such **an entire pass of the training set** has occurred **without error**, training is **complete.**

The Perceptron Learning Rule

1. **Initialize the weights** and **threshold** to ***small random numbers***.
2. At time **step t** present a vector to the neuron inputs and calculate the perceptron **output y(t).**
3. **Update the weights** and **biases** as follows:

$$w_j(t+1) = w_j(t) + \eta(d(t) - y(t))x_j$$

$$b(t+1) = b(t) + \eta(d(t) - y(t))$$

   - d(t) is the desired output
   - y(t) is the computed output
   - t is the step/iteration number
   - η is the gain or step size (Learning Rate), where $0.0 < \eta <= 1.0$
4. Repeat steps 2 and 3 until:
   - The iteration error is less than a user-specified error threshold
   - Or a predetermined number of iterations have been completed.

The perceptron learning algorithm developed originally by F. Rosenblatt in the late 1950s.

Example

$\eta = 1$
$Y = f(\sigma) = Id(\sigma)$

t = 0

…

t = 9

t	INPUTS				d(t)	y(t)	E	WEIGHTS				
	x1	x2	x3	x4				b(t)	w1(t)	w2(t)	w3(t)	w4(t)
0								0	0	0	0	0
1	0	0	0	1	-1	0	1	-1	0	0	0	-1
2	1	1	1	0	1	1	2	0	1	1	1	-1
3	1	1	1	1	1	2	0	0	1	1	1	-1
4	0	0	1	1	-1	0	1	-1	1	1	0	-2
5	0	0	0	0	1	-1	2	0	1	1	0	-2
6	0	1	0	1	-1	-1	0	0	1	1	0	-2
7	1	0	0	0	1	1	0	0	1	1	0	-2
8	1	0	1	1	1	-1	2	1	2	1	1	-1
9	0	1	0	0	-1	2	3	0	2	0	1	-1

Training a perceptron (2)

- **Learning only occurs when an error is made**, **otherwise the weights are left unchanged!!.**
- During training, **it is useful to measure the performance of the network** as it attempts **to find the optimal weight set**.
- A **common error measure used is sum-squared errors** (computed over all of the input vector / output vector pairs in the training set):

$$E = \frac{1}{2} \sum_{i=1}^{p} \|d_i(t) - y_i(t)\|^2$$

- where “p” is the **number of input/output vector pairs** in the training set.
- **η - Learning rate** - Dictates how **quickly the network converges**.
- It is set by a matter of experimentation (usually small – e.g. 0.1)

Two types of network training

- **Sequential mode**
  - *on-line or per-pattern*
  - Weights updated **after each pattern is presented** (**Perceptron** is in this class)
- **Batch mode**
  - *off-line or per-epoch*
  - Weights updated **after all patterns are presented**

Learning

- Training from data set, adaptation
  - Extracts **principles from training data** set in order to **generalize to other data**
- The purpose of learning is to **minimize error**:
  - On the **training data set**
  - On the **testing set** (prediction errors)!!!
- Two main types of Neural Network LEARNING:
  - **Supervised learning**
    - Have a teacher, telling you what is the **output (target)** for a given input pattern.
  - **Unsupervised learning**
    - No teacher, learn by itself.

Adaline Learning (Delta Rule)

- From it we can better **understand the Perceptron learning rule**, and the more general **BackPropagation learning**
- **Adaline learning** was developed by Widrow and Hoff (1960).
- **ADALINE** is an acronym for **ADAptive LInear Neuron**
  - Neurons in the network have ***linear activation functions***
- The **Adaline learning rule**
  - Also known as the **Delta rule** or the **Widrow-Hoff rule**
  - It is a training rule that **minimizes the output error** using (approximate) **gradient descent method**

Adaline Learning

- After all training pattern vectors xi (i=1,…,p) are presented, the correction to apply to the weights is proportional to the error E(t):

$$E(t) = \frac{1}{2}\sum_{i=1}^{p}[d_i(t) - f(\vec{w}(t)\vec{x}_i)]^2$$

- Our purpose is to find the vector w which minimizes E(t). At each step:

$$w(t+1) = w(t) + \Delta w(t)$$

- In gradient descent techniques:

$$\vec{w}(t+1) = \vec{w}(t) - \eta \nabla E(\vec{w})$$

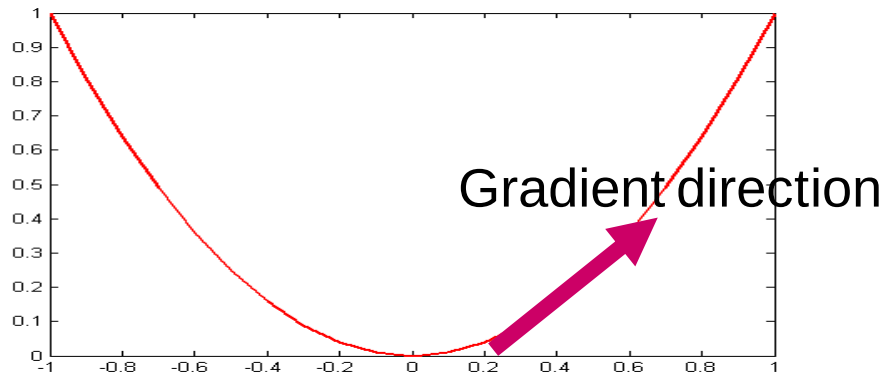
- η >0 learning rate

Adaline Learning

- By analogy, **gradient method** can be **compared with a ball rolling down from a hill**:
  - the ball will **roll down and finally stop at the valley**.
  - Gradient direction is the **direction of uphill** (in the Figure – **E(w) one dimensional case**)
- In a gradient descent algorithm, the ball goes in the **opposite direction to the gradient**, i.e., we have

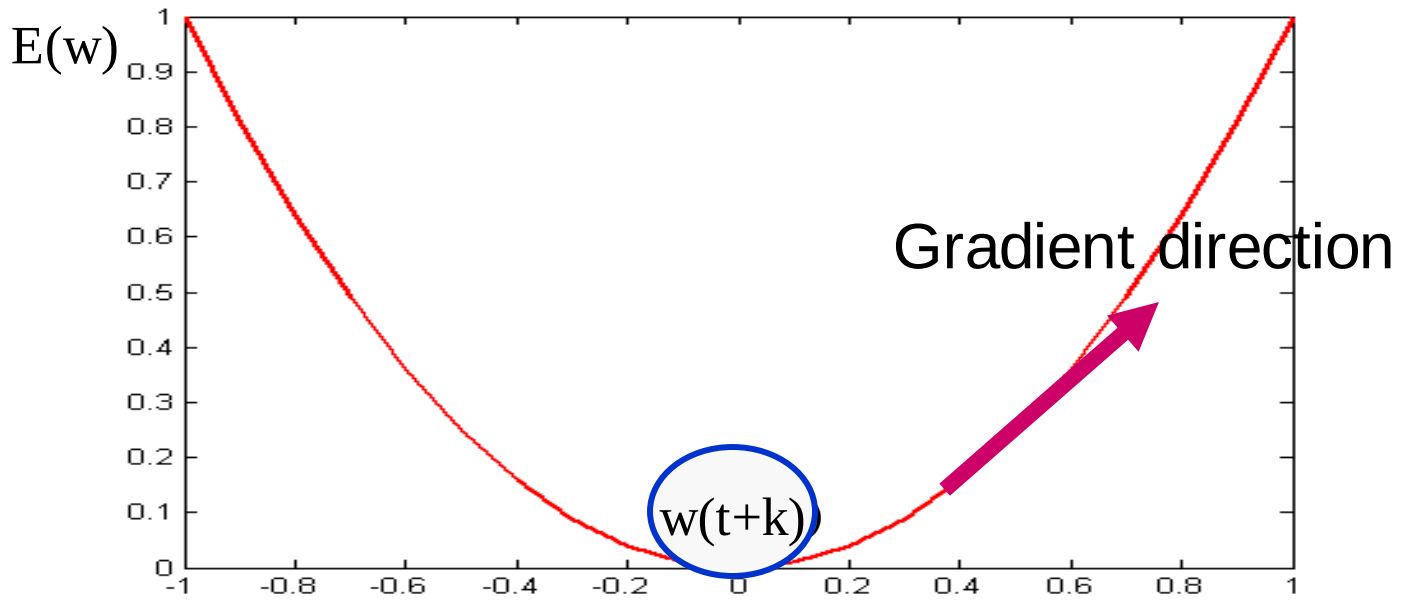
$$\vec{w}(t+1) = \vec{w}(t) - \eta \nabla E(\vec{w})$$

- therefore the ball goes downhill since – E'(w(t)).



Adaline Learning

- Gradually the ball will stop at a (local or global) **minima where the gradient is zero**



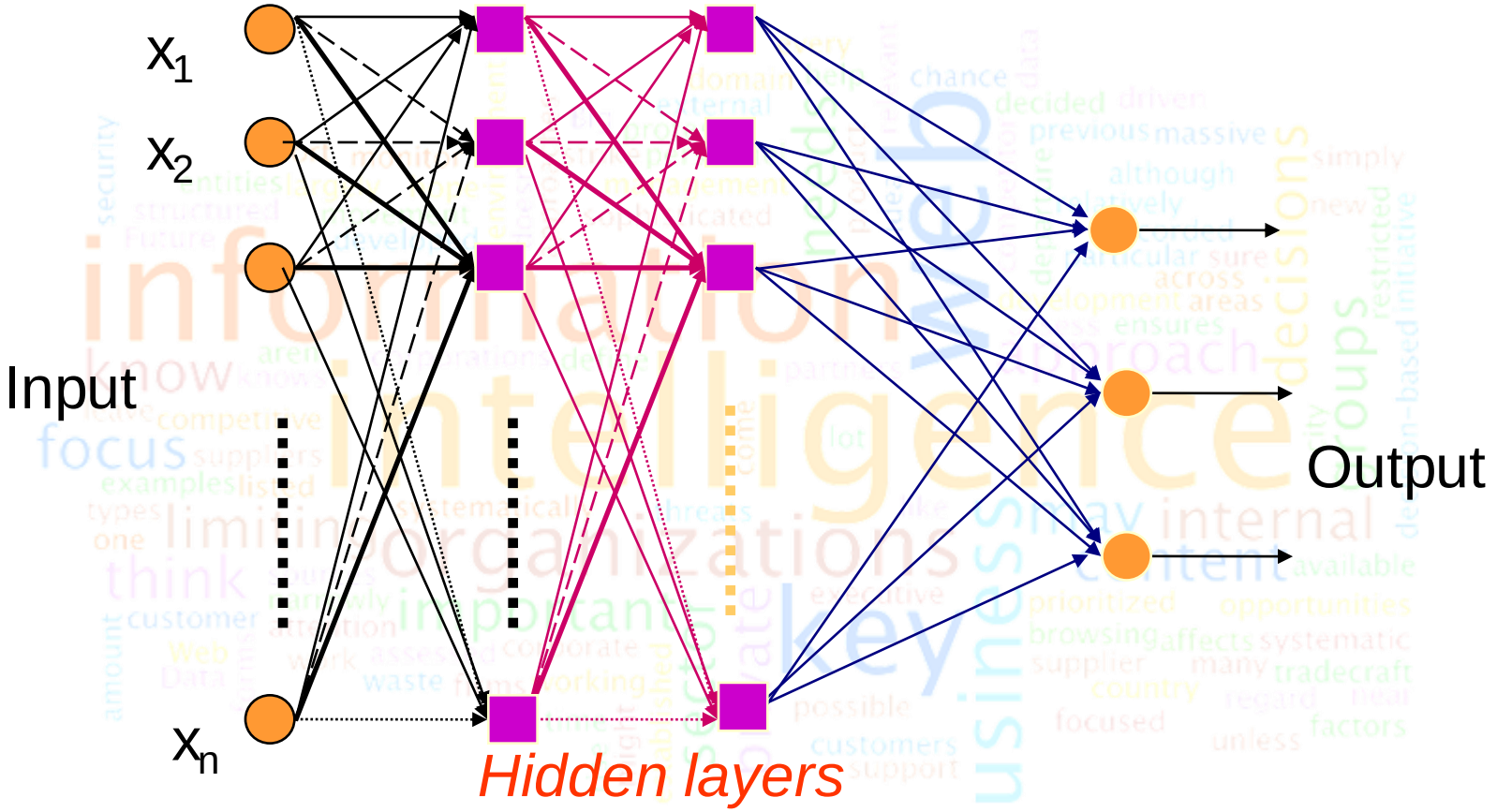
Key differences
between Delta Rule and Perceptron Training Rule

- The **Perceptron training rule** converges **after a finite number of iterations** to a **solution that perfectly classifies the training data**, provided the training examples are linearly separable.
- The **Delta rule** converges only **asimptotically** toward the **minimum error solution**, possibly requiring **unbounded time**, but converges regardless of whether the training data are linearly separable or not.
- The Perceptron rule **updates weights based** on the error in the **thresholded Perceptron output,** whereas the Delta rule **updates weights based** on the error considering a **linear activation function for neurons.**

Backpropagation

- Training algorithm for multilayer neural networks (or MLP – Multi-Layer Perceptron)
- Supervised learning algorithm based on gradient descent
- Also named Generalized Delta Rule Introduced by Rumelhart, Hinton & Williams (1986) Parker (1982), Werbos (1974)
- Require differentiable "activation functions" for neurons, such as sigmoid function
- BP neural networks are the most widely used neural networks
- BP networks can learn any non-linear function.

Example of a four-layer neural network



3 proper neuron layers the first (input layer) is dummy, only transmit the inputs to the next layer

Summary of BP learning algorithm

```
Set learning rate
Set initial weight values (including biases):  W, b

Loop until stopping criteria satisfied:
   For each of the patterns in the training set
       present an input pattern  to input units
       compute output signal for hidden units
       compute output signal for output units

        present Target response to output units
       compute error signal for this pattern
  Compute an overall error for all the patterns
(e.g. mean squared err )
  Update weights at output layer
  Update weights at hidden layer
   Increment t  to t+1 (t –epoch number)
end loop
```

BP has two phases

- **Forward pass phase**
  - feed-forward propagation of input pattern signals through the network, **from inputs towards the network outputs**
- **Backward pass phase**
  - computes 'error signal' - **propagation of error** (difference between actual and desired output values) backwards through network, **starting from output units towards the input units**

Network Training

- Each full presentation of all patterns = 'epoch'
- Usually better to randomize order of training patterns presented for each epoch in order to avoid correlation between consecutive training pairs being learnt (order effects).
- Training set shown repeatedly until stopping criteria are met.
- Selecting initial weight values:
  - Choice of initial weight values is important as this decides starting position in weight space. That is, how far away from global minimum

Error function of BP training

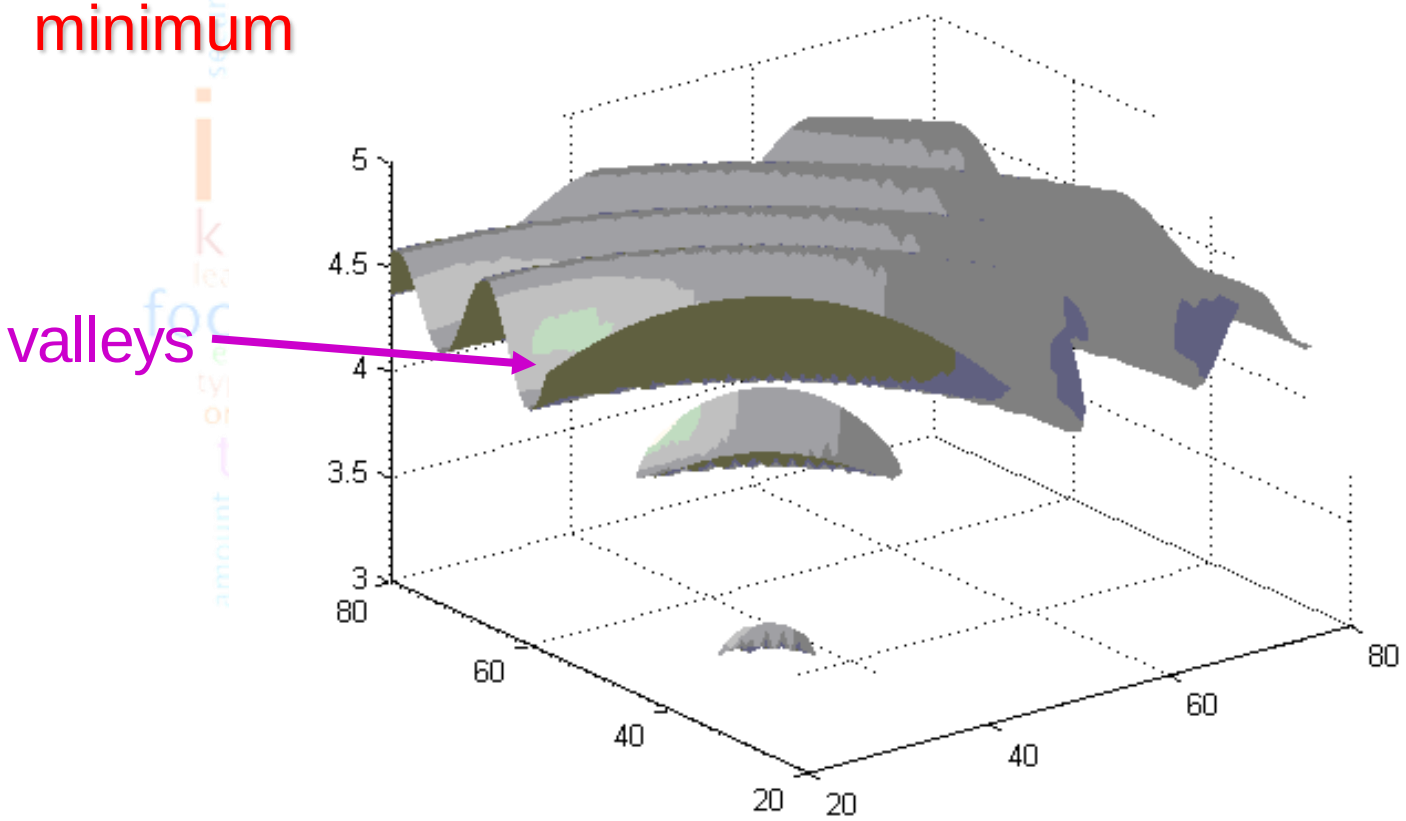
- Aim is to **minimise an error function** over all training patterns by **adapting weights** in MLP.
- **Mean squared error** is typically used:

$$E(t) = \frac{1}{2} \sum_{k=1}^{p} (d_k(t) - y_k(t))^2$$

- p : number of training patterns
- In *single layer Perceptron with linear activation functions* (**ADALINE**), the **error function is simple** and described by a **smooth parabolic surface with a single minimum**.

Error function of BP training

MLP with nonlinear activation functions have *complex error surfaces* (e.g. plateaus, long valleys etc.) with no single minimum



Picture from Jianfeng Feng, lect. notes Univ. of Sussex.

Weights updating in BP

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n)$$

$$\Delta w_{ij}(n) = \eta \delta_o x_{ij} \quad \text{for output layer}$$

$$\Delta w_{ij}(n) = \eta \delta_h x_{ij} \quad \text{for hidden layer}$$

- The values δ_o, δ_h are the gradients at output and respectively hidden layers. For more details on how to calculate these values please see "Machine Learning" - Tom Mitchell (1997)
- The big problem of BP algorithm:
  - Difficulty to cope with local minima and find a global minimum.
- Few improvements were reported:
  - Variable learning rate, momentum, weight decay, use of modified error functions, etc.

Improvement: Momentum

- Method of **reducing problems of instability** while increasing the rate of convergence

- **Modified weight update equation** is:

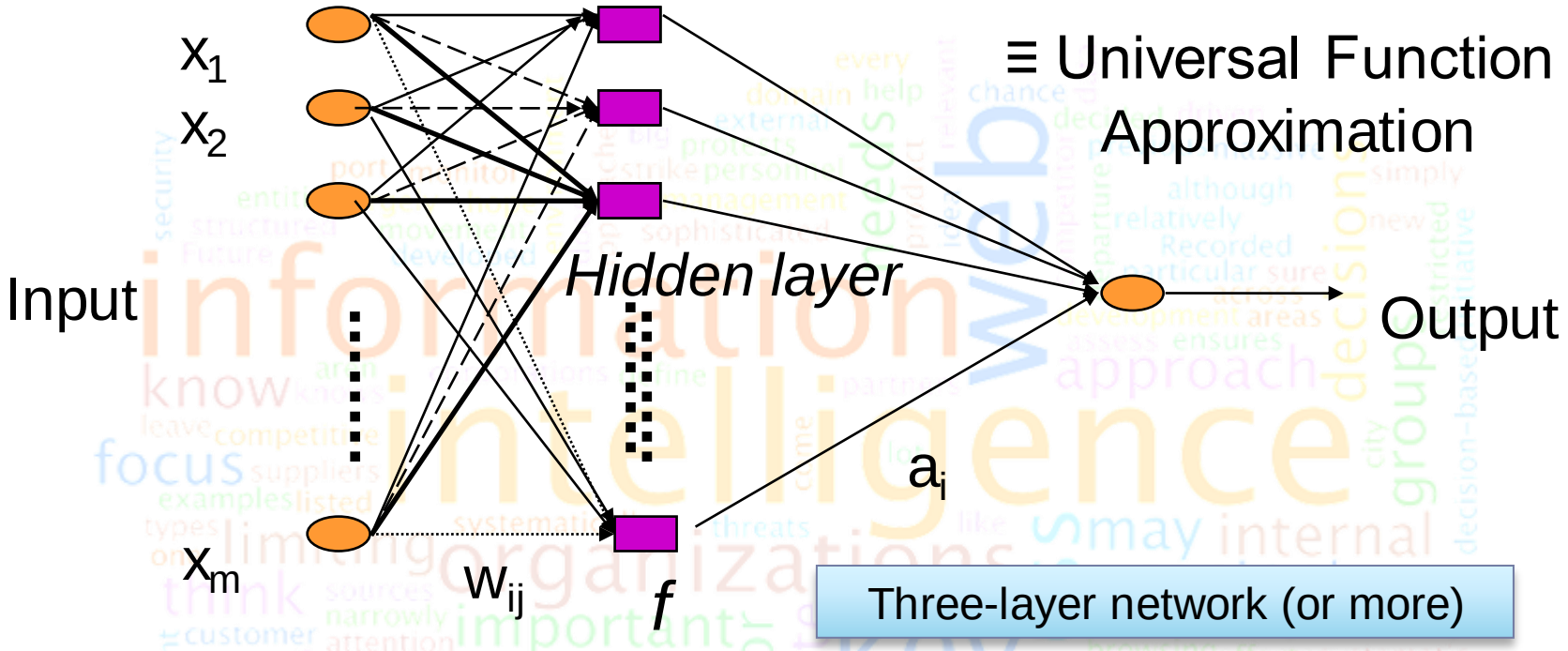
$$w_{ij}(n+1) = w_{ij}(n) + \eta\delta_j(n)x_{ij} + \alpha[w_{ij}(n) - w_{ij}(n-1)]$$

α – Momentum coefficient, $0 <= \alpha < 1$

Effect of momentum term

- If weight changes tend to have the same sign, *momentum term increases* (gradient decreases) – speed up convergence on shallow gradient
- If weight changes tend have opposing signs, *momentum term decreases* and gradient descent slows to reduce oscillations (stabilizes)
- Can help escape when being trapped in local minima
- *Increases the convergence speed with a factor of */(1-*)*

Universal Approximation Theorem



- For any given constant $\varepsilon > 0$ and continuous function *h* $(x_1,\ldots,x_m)$ *with m inputs and n outputs,* there exists a three layer MLP (which computes the function H) with m inputs and n outputs with the property $| h(x_1,\ldots,x_m) - H(x_1,\ldots,x_m) | < \varepsilon$

How do I know if network modifications are needed?

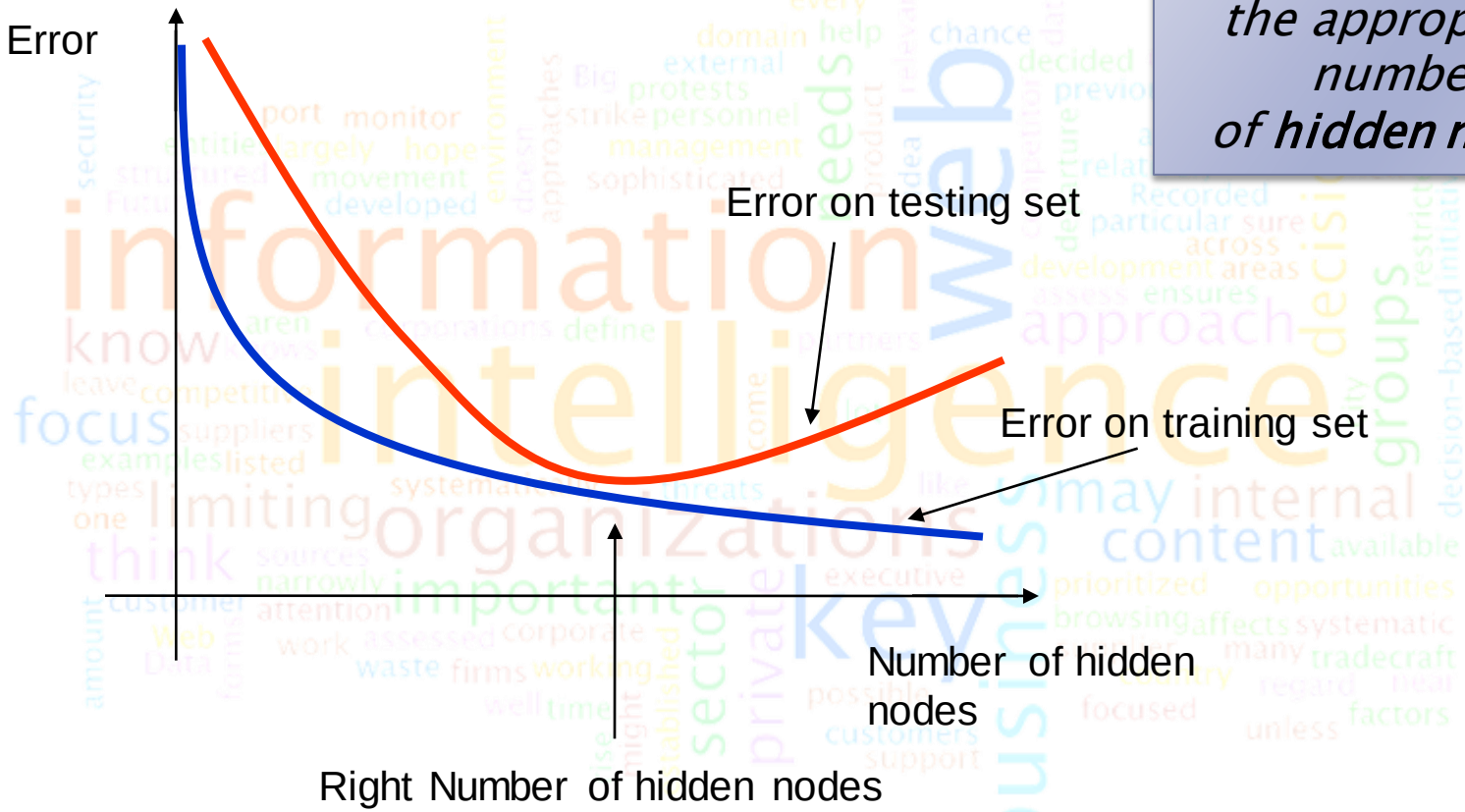
- Low accuracy of training or test data *indicates that a new hidden layer or more hidden nodes are needed*.
  - If the number of hidden nodes exceeds number of inputs and outputs, then add another hidden layer
  - Decrease the total hidden nodes by 50% in each successive hidden layer (e.g., if 10 nodes in first layer, then use 5 in the second layer and 2 in the third layer)
- If NN performs well on the Training set but poorly on Testing set,
  - Then it is treating each record as a special case and has "memorized" the data (lost generalization ability - over fitting).
  - Then use fewer hidden nodes or remove a hidden layer.

What is the best architecture for a Neural Network?

- Divide available data into 2 sets:
  - **Training data set**
    - Used to **train the weights and biases of NN**
  - **Testing data set**
    - Used to **test the performance** of the trained neural network
- If the network **contains more hidden units** (learning parameters) **than necessary** to learn the training set
  - Then the **network will memorize the training patterns**
  - and will **exhibit poor classification abilities for data not contained in the training set** (testing set).
  - That means **the network lost generalization ability - over fitting.**

*the appropriate number of **hidden nodes***

What is the best architecture for a Neural Network?



*the appropriate number of **hidden nodes***

In conclusion, supervised neural networks:

- **Can learn directly from data.**
  - They exhibit **good learning ability** – better than other AI approaches
- Can learn **from noisy or corrupted data**
- **Parallel** information **processing**
- **Computationally fast once trained**
- **Robustness to partial failure** of the network
- **Useful where data are available** and difficult to acquire symbolic knowledge
- Drawback of NN
  - Knowledge captured by a NN through learning (in weights – real numbers) is not in a **familiar form for human beings**, e.g. if-then rules (**NNs are black box structures**).
- **Over fitting issues.**

Self-Organizing Feature Maps (SOFMs)

- Based on Competitive learning
- Neurons compete among themselves to be activated.
- While in "Hebbian learning", several output neurons can be activated simultaneously, in competitive learning, only a single output neuron is active at any time.
- The output neuron that *wins* the "competition" is called the *winner-takes-all* neuron.

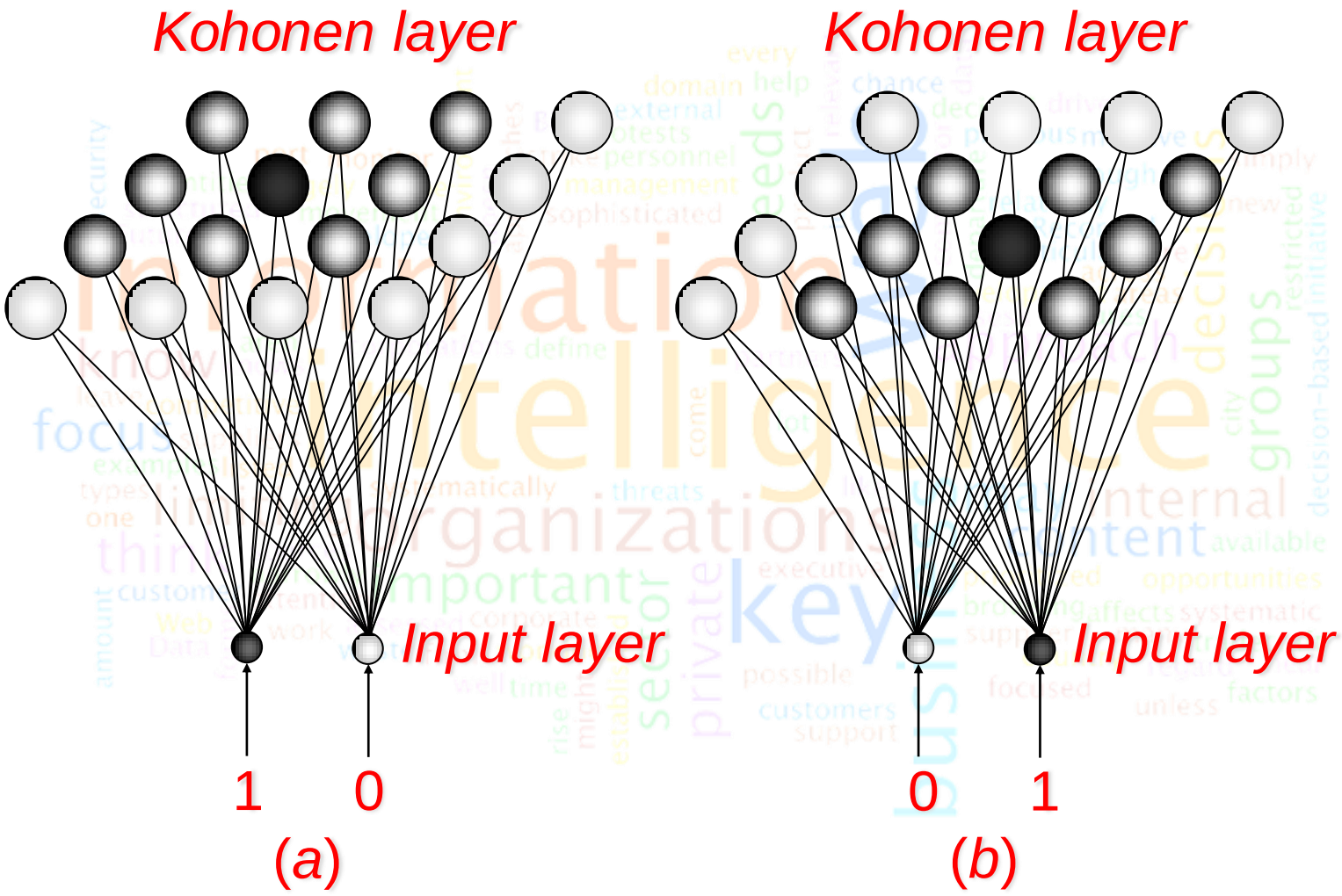
Self-Organizing Feature Maps (2)

- The basic idea of competitive learning was introduced in the early 1970s.
- In the late 1980s, Teuvo Kohonen introduced a special class of artificial neural networks called **Self-Organizing feature Maps**.
- These **maps are based on competitive learning**.

What is a Self-Organizing feature Map?

- **Our brain is dominated by the cerebral cortex**
  - A very complex structure of billions of neurons and hundreds of billions of synapses.
  - The cortex includes areas that are responsible for different human activities (motor, visual, auditory, somatosensory, etc.), and associated with different sensory inputs.
  - We can say that **each sensory input is mapped into a corresponding area of the cerebral cortex**.
  - ***The cortex is a self-organising computational map in the human brain***.

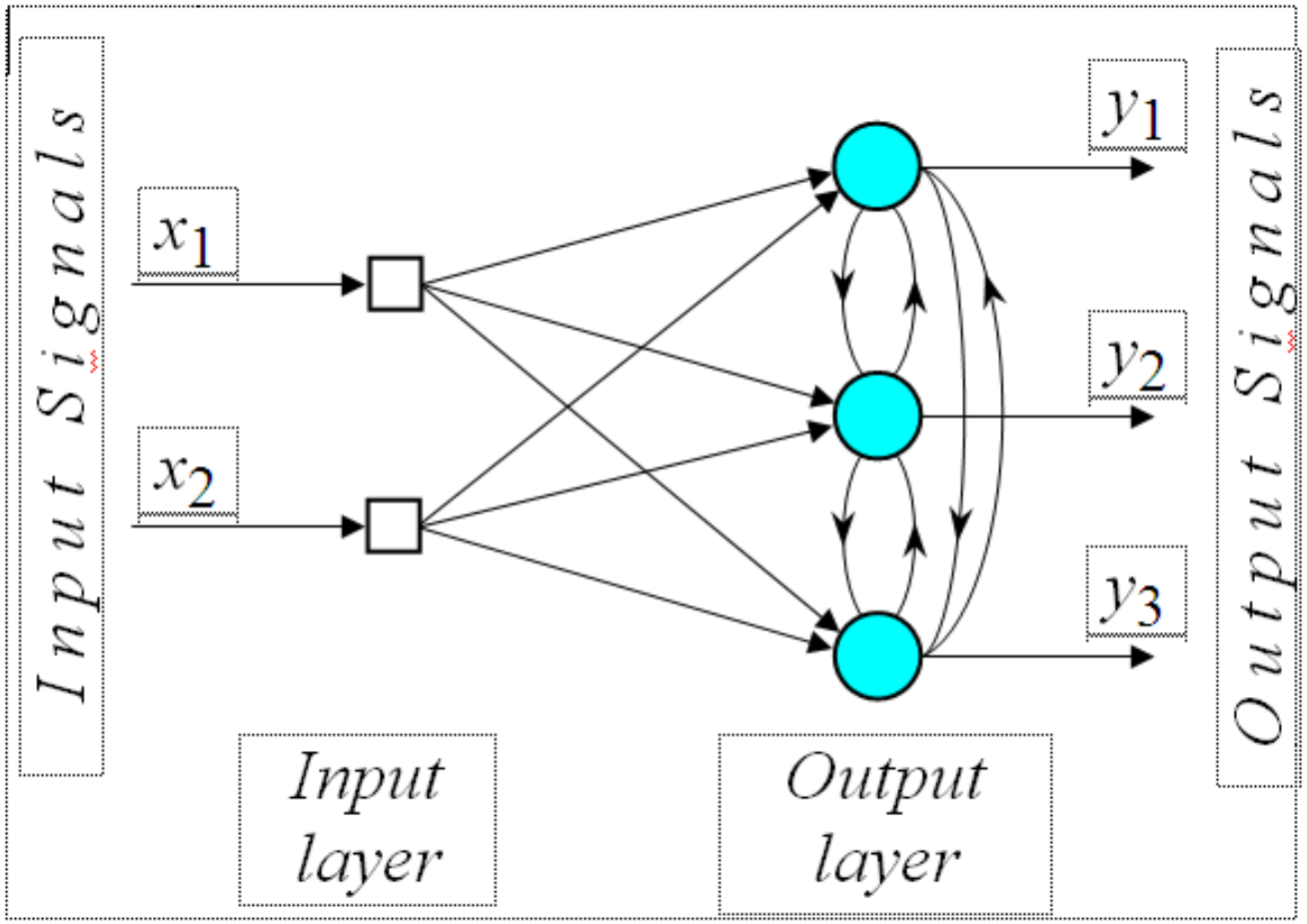
Feature-mapping Kohonen mode



The Kohonen Network

- The Kohonen model provides a topological mapping. It places a fixed number of input patterns from the input layer into a higher-dimensional output or Kohonen layer.
- Training in the Kohonen network begins with the winner's neighbourhood of a fairly large size. Then, as training proceeds, the neighbourhood size gradually decreases.

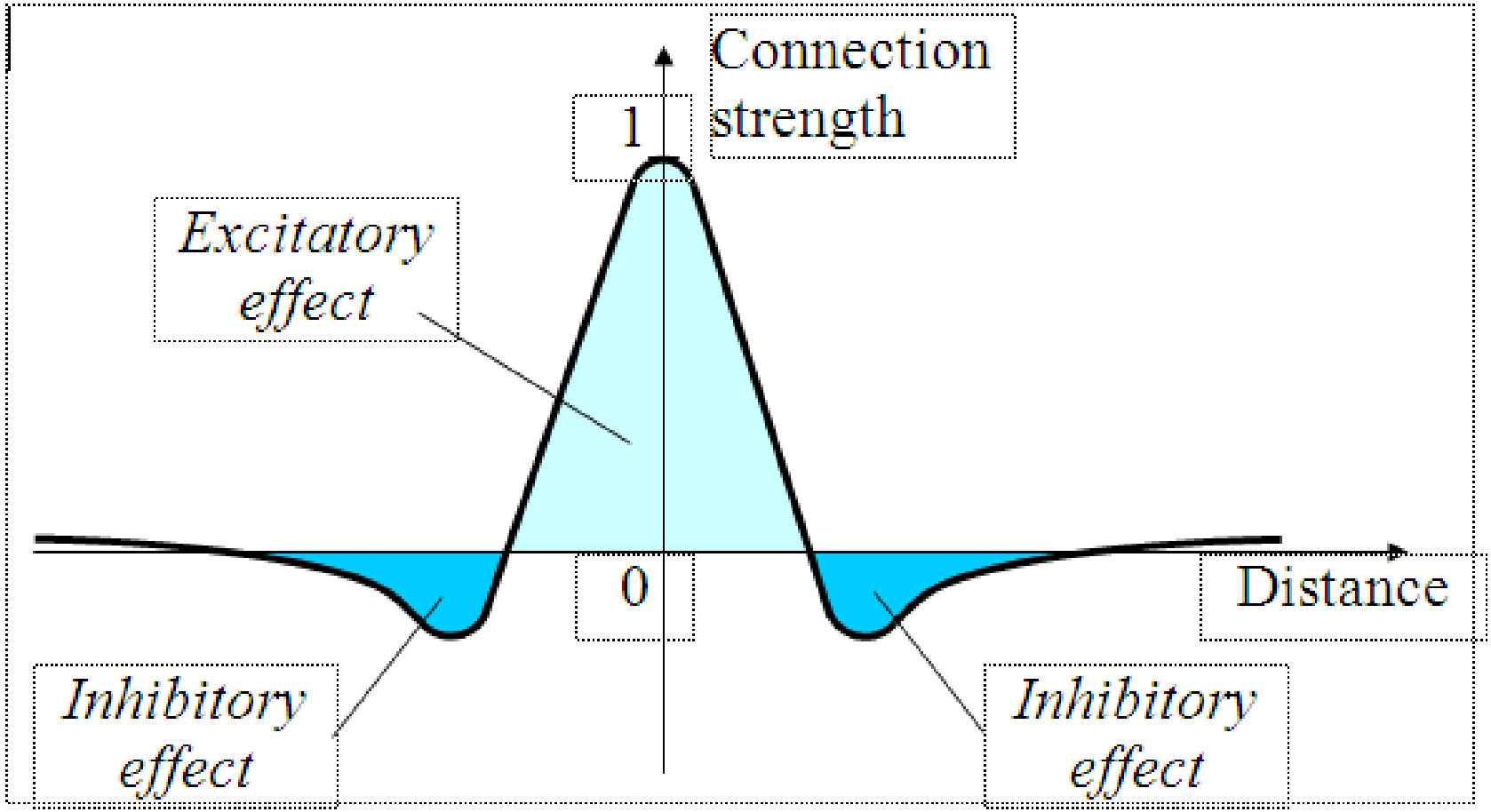
Architecture of the Kohonen Network



Architecture of the Kohonen Network (2)

- The lateral connections are used to create a competition between neurons. The neuron with the largest activation level among all neurons in the output layer becomes the winner. This neuron is the only neuron that produces an output signal. The activity of all other neurons is suppressed in the competition.
- The lateral feedback connections produce excitatory or inhibitory effects, depending on the distance from the winning neuron. This is achieved by the use of a Mexican hat function which describes synaptic weights between neurons in the Kohonen layer.

The Mexican Hat function of lateral connection



Architecture of the Kohonen Network (3)

- In the Kohonen network, a neuron learns by shifting its weights from inactive connections to active ones. Only the winning neuron and its neighbourhood are allowed to learn. If a neuron does not respond to a given input pattern, then learning cannot occur in that particular neuron.
- The competitive learning rule defines the change Δw_{ij} applied to synaptic weight w_{ij} as

$$\Delta w_{ij} = \begin{cases} \alpha(x_i - w_{ij}) & \text{If neuron j wins the competition} \\ 0 & \text{If neuron j loses the competition} \end{cases}$$

where x_i is the input signal and α is the *learning rate* parameter.

Architecture of the Kohonen Network (4)

- The overall effect of the competitive learning rule resides in **moving the synaptic weight vector Wj** of the winning neuron j towards the input pattern X. The matching criterion is equivalent to the **minimum Euclidean distance between vector**s.
- The **Euclidean distance** between a pair of n-by-1 vectors X and Wj is defined by

$$d = \|X - W_j\| = \left[\sum_{i=1}^{n}(x_i - w_{ij})^2\right]^{1/2}$$

Where xi and wij are the ith elements of the vectors X and Wj, respectively.

Architecture of the Kohonen Network (5)

- To identify the winning neuron, jX, that best matches the input vector X, we should apply the following condition:

$$jx = \min_j \|X - W_j\|, j = 1, 2, ...$$

Where m is the number of neurons in the Kohonen layer.

Example

- Suppose, for instance, that the 2-dimensional input vector X is presented to the **three-neuron Kohonen network**,

$$jx = \begin{bmatrix} 0.52 \\ 0.12 \end{bmatrix}$$

- The initial weight vectors, Wj, are given by

$$W_1 = \begin{bmatrix} 0.27 \\ 0.81 \end{bmatrix} W_2 = \begin{bmatrix} 0.42 \\ 0.70 \end{bmatrix} W_3 = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix}$$

Example (2)

- We find the winning (best-matching) neuron j_X using the minimum-distance Euclidean criterion:

$$d_1 = \sqrt{(x_1 - w_{11})^2 + (x_2 - w_{21})^2}$$
$$d_1 = \sqrt{(0.52 - 0.27)^2 + (0.12 - 0.81)^2} = 0.73$$

$$d_2 = \sqrt{(x_1 - w_{12})^2 + (x_2 - w_{22})^2}$$
$$d_2 = \sqrt{(0.52 - 0.42)^2 + (0.12 - 0.70)^2} = 0.59$$

$$d_3 = \sqrt{(x_1 - w_{13})^2 + (x_2 - w_{23})^2}$$
$$d_3 = \sqrt{(0.52 - 0.43)^2 + (0.12 - 0.21)^2} = 0.13$$

- Neuron 3 is the winner and its weight vector W_3 is updated according to the competitive learning rule.

$$\Delta w_{13} = \alpha(x_1 - w_{13}) = 0.1(0.52 - 0.43) = +0.01$$
$$\Delta w_{23} = \alpha(x_2 - w_{23}) = 0.1(0.12 - 0.21) = -0.01$$

Example (3)

- The updated weight vector W_3 at iteration (p + 1) is determined as:

$$W_3(p+1) = W_3(p) + \Delta W_3(p) = \begin{bmatrix} 0.43 \\ 0.21 \end{bmatrix} + \begin{bmatrix} +0.01 \\ -0.01 \end{bmatrix} = \begin{bmatrix} 0.44 \\ 0.20 \end{bmatrix}$$

- The weight vector W_3 of the wining neuron 3 becomes closer to the input vector X with each iteration.

Competitive Learning Algorithm
STEP 1: INITIALIZATION

- Set **initial synaptic weights to small random values**, say in an interval [0, 1], and assign a **small positive value to the learning rate** parameter α.

Competitive Learning Algorithm
STEP 2: ACTIVATION AND SIMILARITY MATCHING

- **Activate the Kohonen network** by applying the input vector X, and find the **winner-takes-all** (best matching) neuron jX at iteration p, using **the minimum-distance Euclidean criterion**

$$\|X - W_j(p)\| = \{\sum_{i=1}^{n}[x_i - w_{ij}(p)]^2\}^{1/2}$$

Where n is the number of neurons in the input layer, and m is the number of neurons in the Kohonen layer.

Competitive Learning Algorithm
STEP 3: LEARNING

- Update the synaptic weights

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

Where $\Delta w_{ij}(p)$ is the weight correction at iteration p.

- The weight correction is determined by the competitive learning rule:

$$\Delta w_{ij}(p) = \begin{cases} \alpha[x_i - w_{ij}(p)], & j \in \Lambda_j(p) \\ 0, & j \notin \Lambda_j(p) \end{cases}$$

Where α is the *learning rate* parameter, and $\Lambda_j(p)$ is the neighbourhood function centred around the winner-takes-all neuron j_X at iteration p.

Competitive Learning Algorithm

STEP 4: ITERATION

- Increase iteration p by **one.**

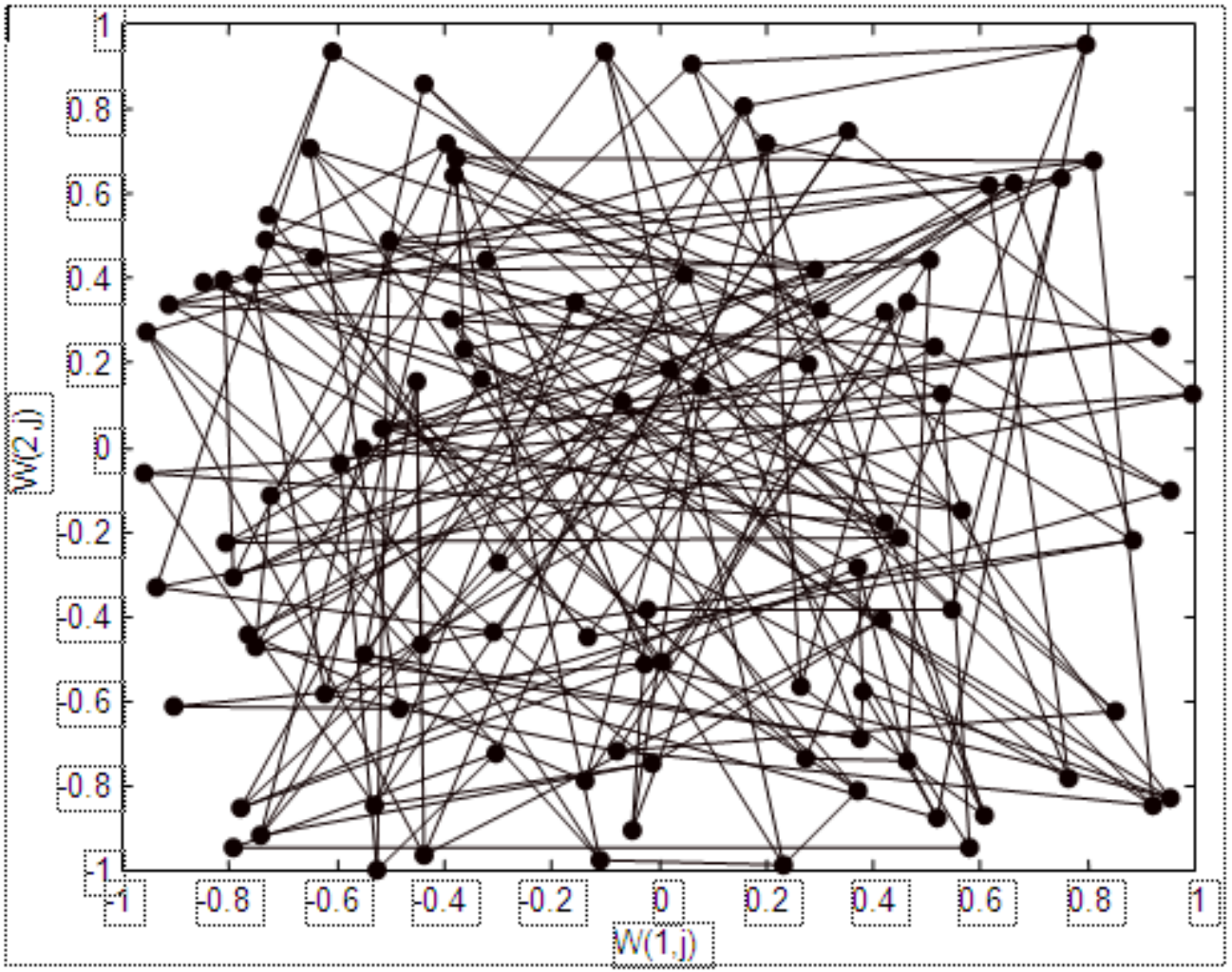
- Go back to Step 2 and continue until the **minimum-distance Euclidean criterion** is satisfied, or **no noticeable changes occur in the feature map**.

Competitive learning in the Kohonen network

- To illustrate competitive learning, consider the Kohonen network with **100 neurons arranged in the form of a two-dimensional lattice** with 10 rows and 10 columns.
- The network is required to classify **two-dimensional input vectors** in each neuron in the network should respond **only to the input vectors occurring in its region.**
- The network is trained with **1000 two-dimensional input vectors** generated **randomly in a square region** in the interval between –1 and +1. The learning rate parameter α is equal to 0.1.

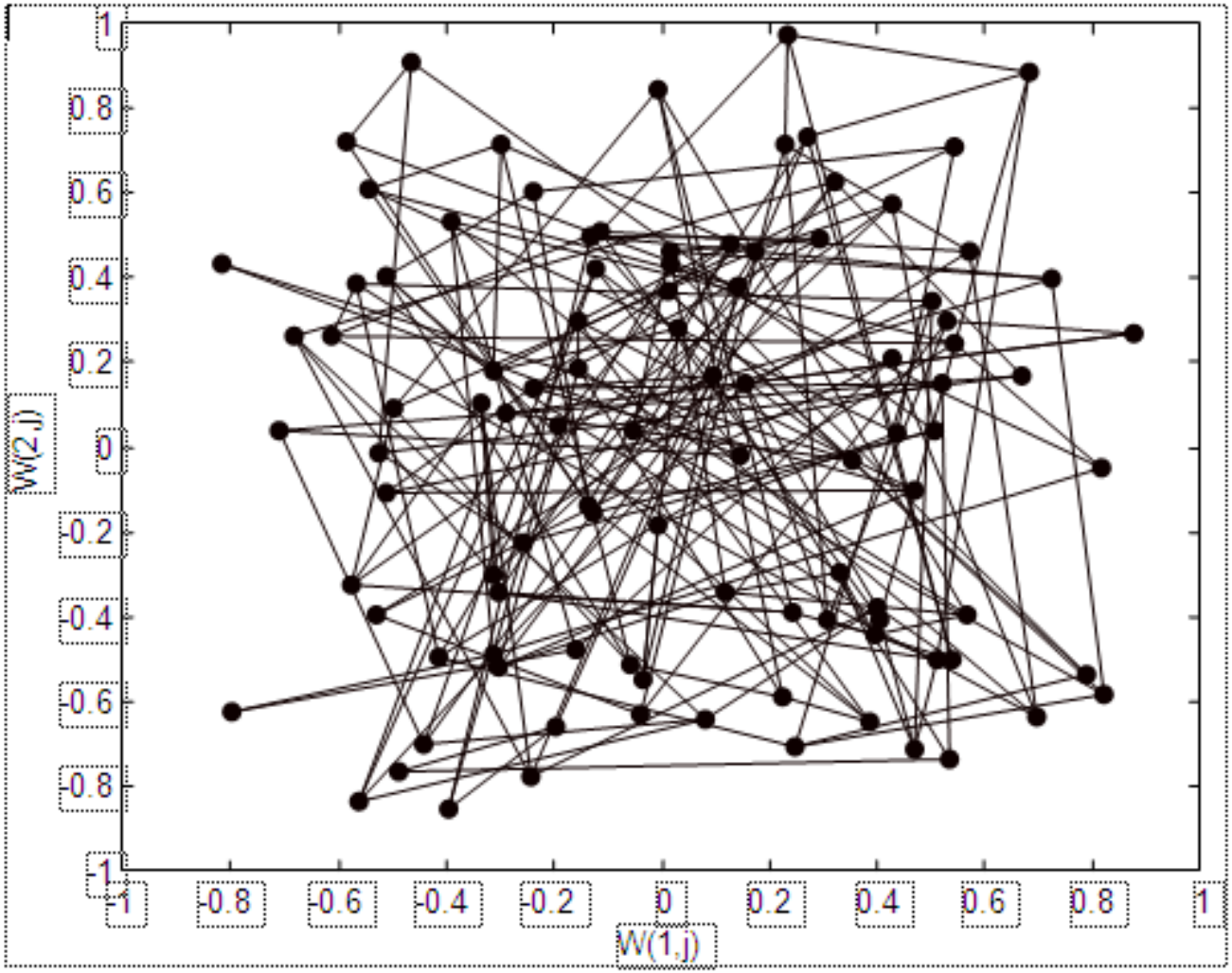
Example of the Evolution

INITIAL RANDOM WEIGHTS

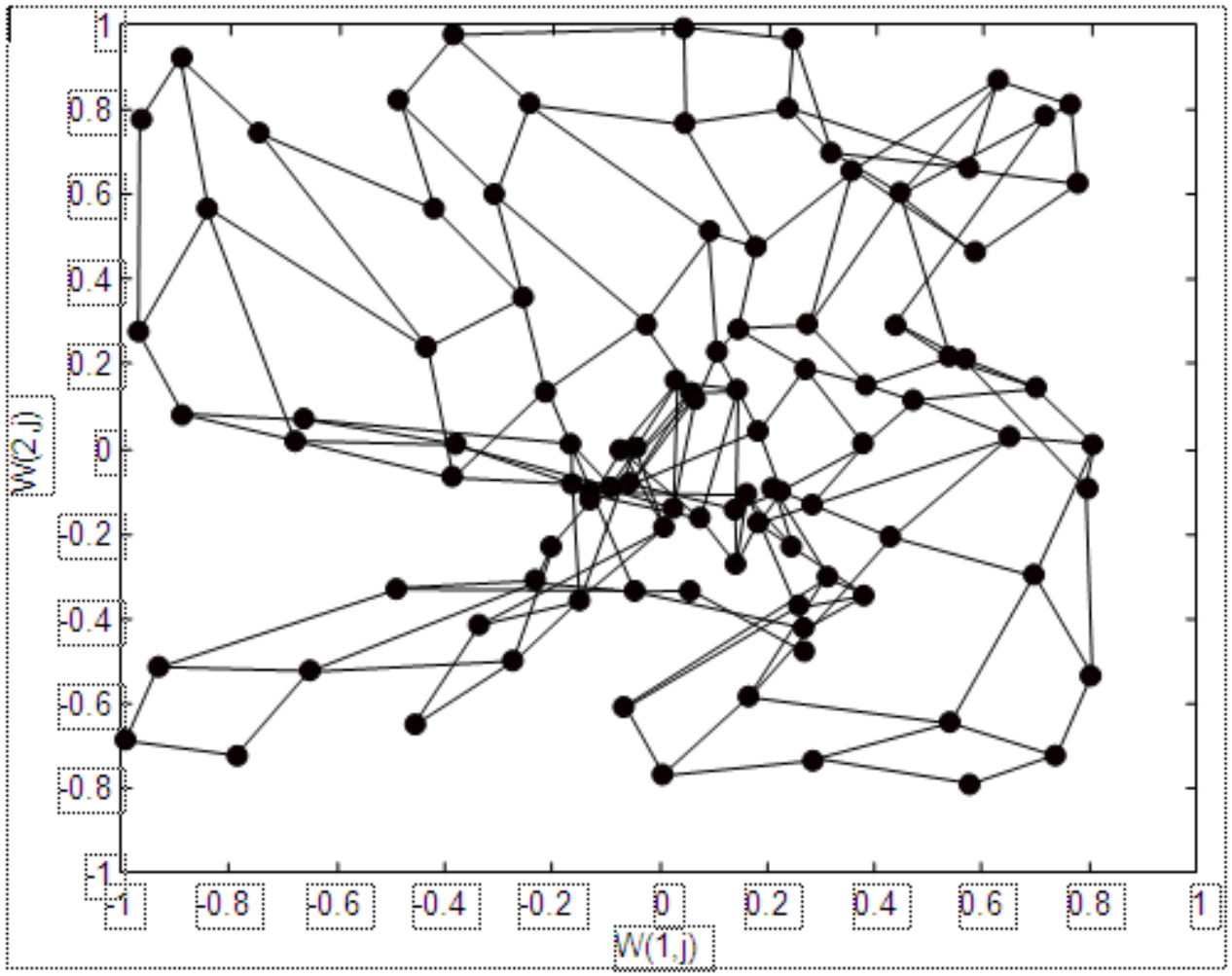


Example of the evolution

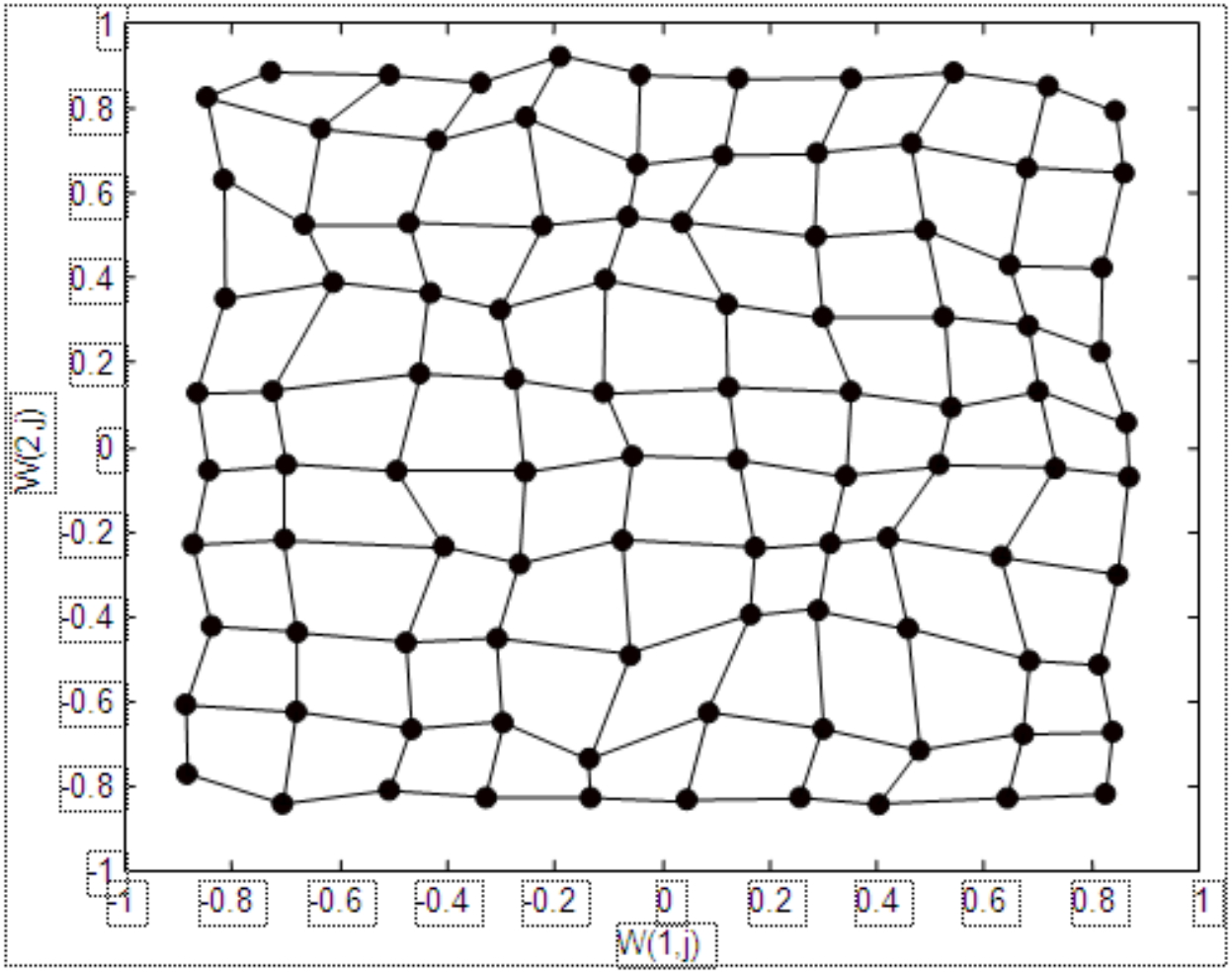
NETWORK AFTER 100 ITERATIONS



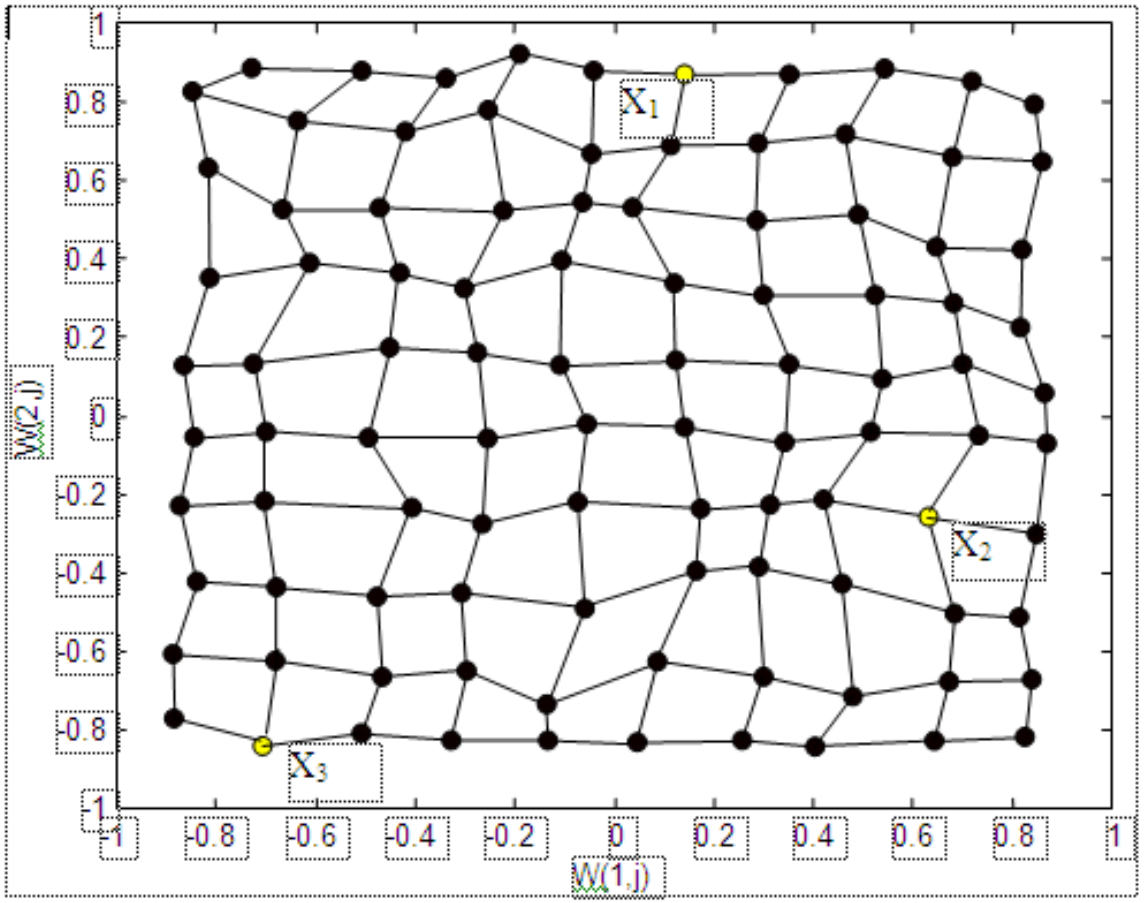
Example of the evolution
NETWORK AFTER 1000 ITERATIONS



Example of the evolution
NETWORK AFTER 10000 ITERATIONS



Example of the responses



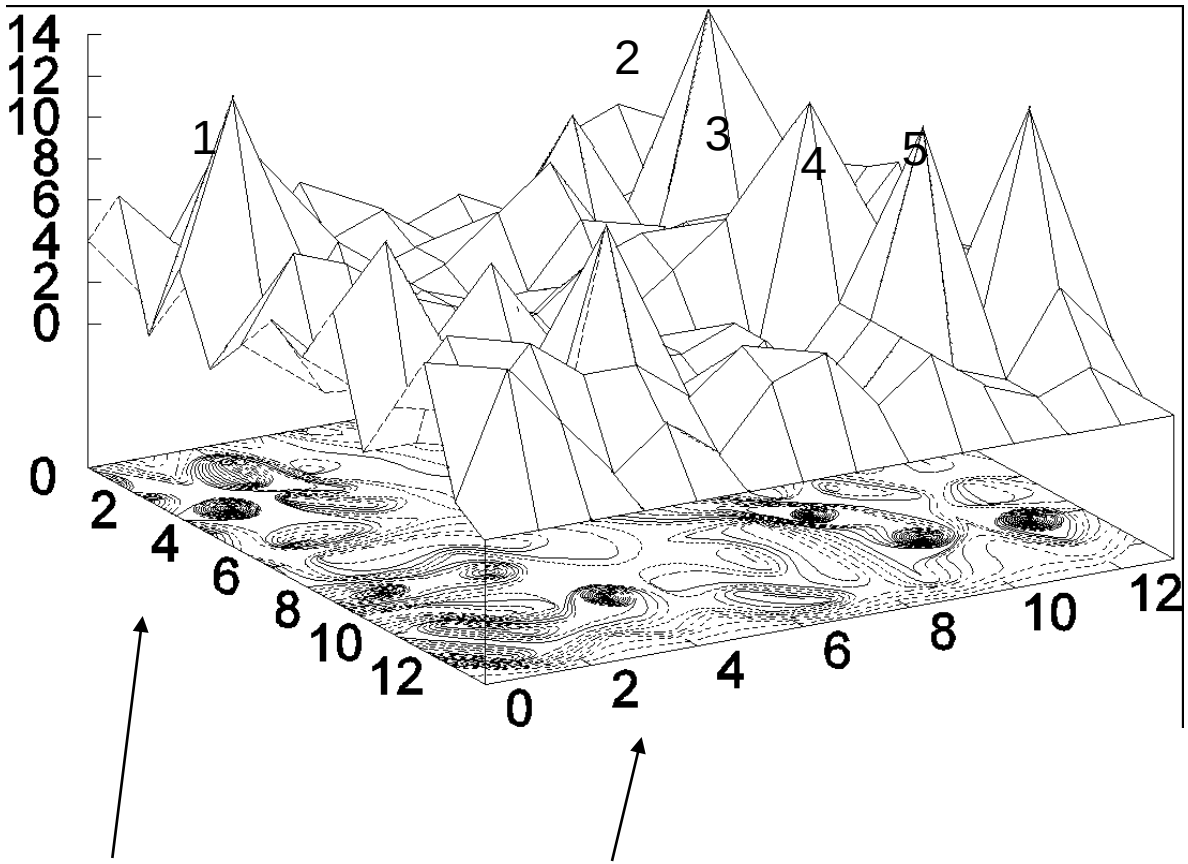
$$\mathbf{X}_1 = \begin{bmatrix} 0.2 \\ 0.9 \end{bmatrix}$$

$$\mathbf{X}_2 = \begin{bmatrix} 0.6 \\ -0.2 \end{bmatrix}$$

$$\mathbf{W}_3 = \begin{bmatrix} -0.7 \\ -0.8 \end{bmatrix}$$

Example of the clusters extraction

Winner frequency



Kohonen map

Five clusters

Summary on Neural Networks
WHAT DID WE LEARN?

- Different **neural network architectures** and learning algorithms:
- **Supervised learning**
  - Perceptron (Perceptron learning rule)
  - Adaline (Delta rule)
  - Feed forward Multi-Layer Perceptron (Back propagation).
- **Unsupervised learning**
  - Competitive learning
  - Kohonen Self-Organizing Maps

Section 2.7

Bayesian Networks

The Bayes framework

- Probability theory predict the **likelihood of different outcomes.**
- **Bayes theorem** give a formula for calculating a *conditional probability based on the reverse condition that sometime is easier*.
- **Conditional probability**:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

The Bayes framework (2)

- ***The Bayes Theorem***

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- Expanding total probability for calculating $P(A=a_i|B)$:

$$P(A = a_i|B) = \frac{P(B|A = a_i)P(A = a_i)}{\sum_k P(B|A = a_k)P(A = a_k)}$$

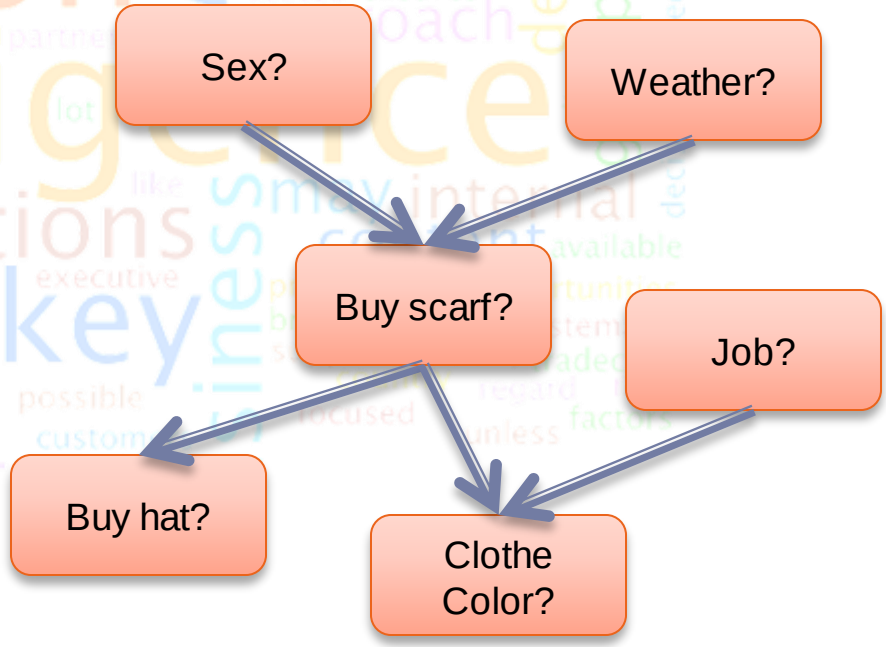
The Bayes framework (3)

- **Bayesian Network**
  - If **P(A|B)** is "**highly probable**" **is equivalent to** [**B=>A**] association Rule
  - A graph of relation [B=>A] is called "Bayesian Network"

- **Example**

If(sex=male & weather=rainy) Then, buy scarf is true with 80% prob.

P(buy scarf=yes|gender=male, weather=rainy)=0.8



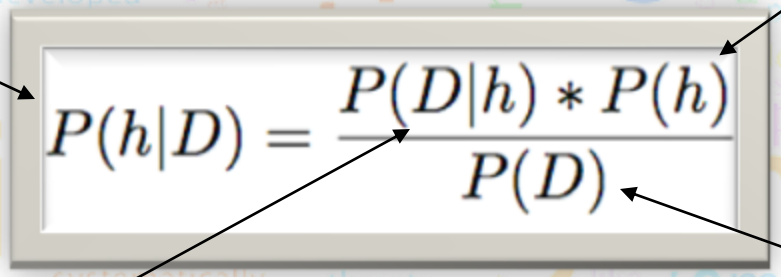
(ML) Algorithms as Bayesian Learners

- So far, we have looked at **algorithm-independent methods** for *evaluating and comparing models.*
- ML algorithms can be seen as **algorithms that seek answers** to one or both of the following:
  - What is the **most probable hypothesis** (model) **given a set of 'training' data**?
  - What is **most probable classification of new data** instances?
- **Bayesian reasoning** provides the basis for answering these questions
  - Many choices made by practical ML algorithms can be better understood within Bayesian setting.

Looking at the Bayes Rule

Probability that hypothesis h holds given the observed training data D

Prior probability of a hypothesis h



Probability of observing data D if hypothesis h holds

Prior probability of observing the training data D

$$P(h|D) \propto P(D|h) \times P(h)$$

Maximum A Posteriori (MAP) Hypothesis

- The **most (maximally) probable hypothesis**, given the data is called the *maximum a posteriori (or MAP) hypothesis*:

$$h_{MAP} = argmax_h P(D|h) \times P(h)$$

Maximum Likelihood Hypothesis

- The term P(D | h) is called the ***'likelihood'*** of the data, given the hypothesis. If **all hypotheses are equally likely**, then the MAP hypothesis reduces to the *maximum likelihood or ML hypothesis*:

$$h_{MAP} = h_{ML} = argmax_h P(D|h)$$

A Different View of MAP

$$h_{MAP} = argmin_h \; -\log_2 P(D|h) \; -\log_2 P(h)$$

bits required to encode D given h using an optimal code

bits required to encode h using an optimal code

The Minimum Description Length (MDL) Principle

1. Choose a scheme for encoding hypotheses (C1) and for encoding data given a hypothesis (C2)
2. The ***MDL hypothesis*** is:

$$h_{MDL} = argmin_h [L_{C_1}(h) + L_{C_2}(D|h)]$$

If C_1 and C_2 are optimal, then $h_{MDL} = h_{MAP}$

Example

Given:

A	B	C
m	b	t
m	s	t
g	q	t
h	s	t
g	q	t
g	q	f
g	s	f
h	b	f
h	q	f
m	b	f

Figure out:

$A = m \quad B = q \quad C = ?$

$A \in \{m, g, h\}$

$B \in \{b, s, q\}$

$C \in \{t, f\}$

Example

$$h_{MAP} = argmax_h P(D|h) \times P(h)$$

$$c = \arg\max_{c_j} \Pr(C = c_j) \prod_{i=1}^{|A|} \Pr(A_i = a_i \mid C = c_j)$$

$$\Pr(C = c_j) = \frac{\text{number of examples of class } c_j}{\text{total number of examples in the data set}}$$

$$\Pr(A_i = a_i \mid C = c_j) = \frac{\text{number of examples with } A_i = a_i \text{ and class } c_j}{\text{number of examples of class } c_j}.$$

Example

$\Pr(C = t) = 1/2, \quad \Pr(C = f) = 1/2$

$\Pr(A=m \mid C=t) = 2/5$
$\Pr(A=m \mid C=f) = 1/5$
$\Pr(B=b \mid C=t) = 1/5$
$\Pr(B=b \mid C=f) = 2/5$

$\Pr(A=g \mid C=t) = 2/5$
$\Pr(A=g \mid C=f) = 2/5$
$\Pr(B=s \mid C=t) = 2/5$
$\Pr(B=s \mid C=f) = 1/5$

$\Pr(A=h \mid C=t) = 1/5$
$\Pr(A=h \mid C=f) = 2/5$
$\Pr(B=q \mid C=t) = 2/5$
$\Pr(B=q \mid C=f) = 2/5$

$$\Pr(C = t)\prod_{j=1}^{2} \Pr(A_j = a_j \mid C = t) = \frac{1}{2} \times \frac{2}{5} \times \frac{2}{5} = \frac{2}{25}$$

$$\Pr(C = f)\prod_{j=1}^{2} \Pr(A_j = a_j \mid C = f) = \frac{1}{2} \times \frac{1}{5} \times \frac{2}{5} = \frac{1}{25}$$

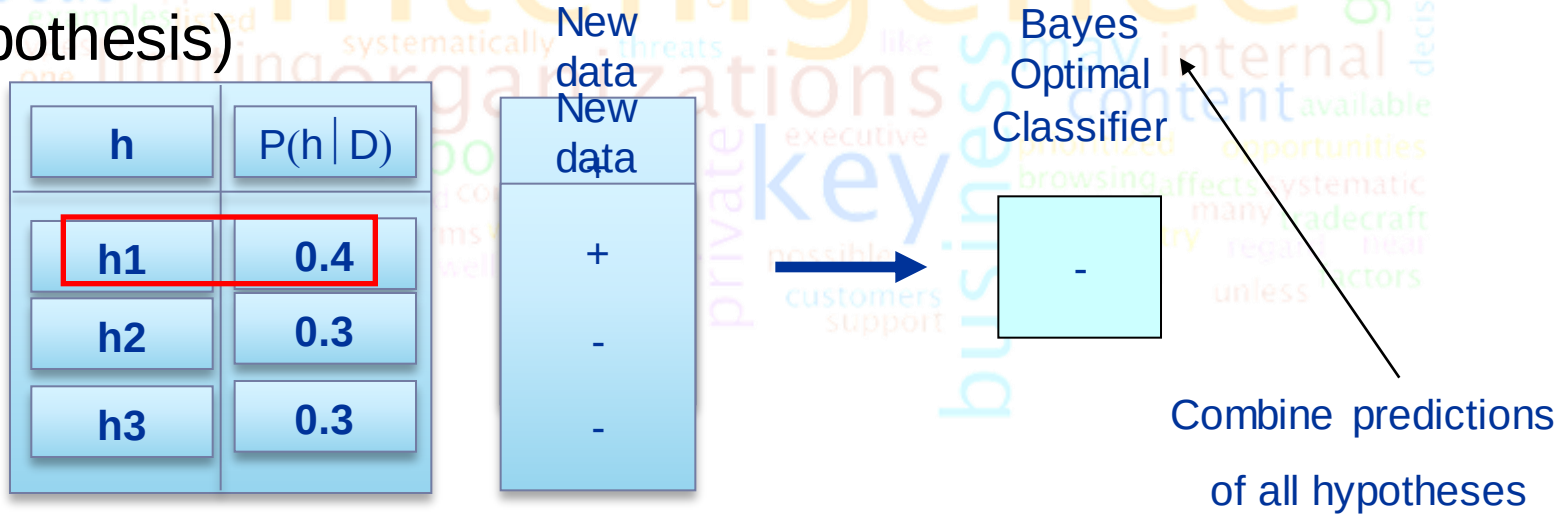
Final answer: h_{MAP} is t

Machine Learning Algorithms as MAP Learners

- Many choices made by particular ML algorithms translate to particular assumptions within the Bayesian framework for identifying $\mathbf{h_{MAP}}$
  - A NN that tries to find the **model that minimises MSE on the training set** is equivalent to a **Maximum Likelihood hypothesis** *under the assumption about noise in the training data*. Recall that this means that it is a **MAP hypothesis that assumes all hypotheses** (models) are **equally likely**.
  - A **Classification Tree** that tries to balance the size of the tree with the errors made on the training set can be seen as an **algorithm that is employing the MDL principle**. The result is some approximation to the **MAP hypothesis**.

Is the MAP Hypothesis Best?

- For a given problem (given the space of hypotheses and prior information),
  - *Will the MAP hypothesis result in the lowest error when predicting new data?*
- **Surprisingly: No!** The *'Bayes Optimal Classifier'* is one that classifies the new data instance by combining the predictions of *all* the hypotheses (not just the MAP hypothesis)



Section 2.8

K-Nearest Neighbour (KNN)

K-Nearest Neighbour Learning

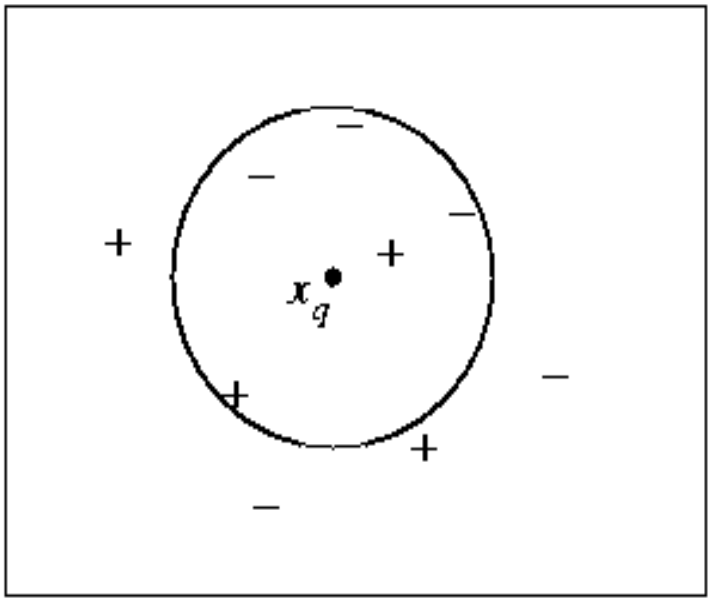
- 1-Nearest Neighbour:

  Given a query (test) instance x_q,
  - Locate the **nearest training example** x_n and assign $f(x_q) = f(x_n)$
- K-Nearest Neighbour:

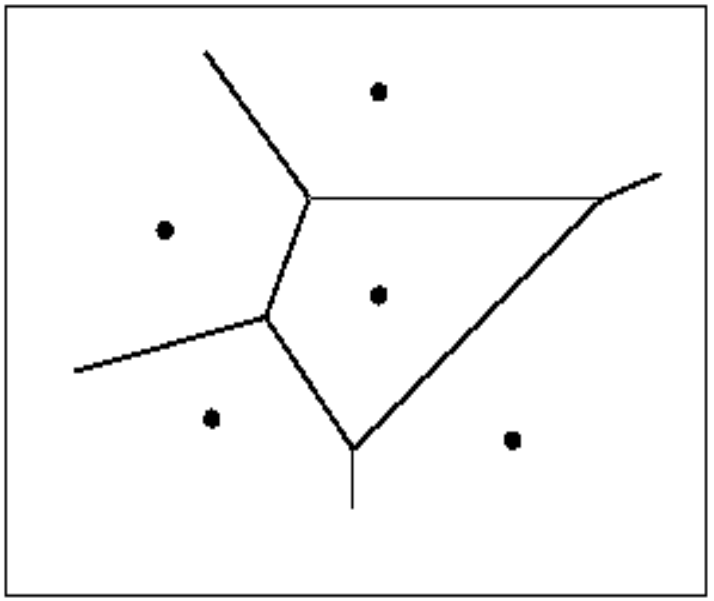
  Given a query (test) instance x_q,
  - Locate the **k-nearest training examples**
  - If the **target function has discrete values**, then return the **most common value** of f among the k nearest training examples;
  - If **continuous-valued target function**, then return the **mean of the f values** of the k nearest training examples.

$$f(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

K-Nearest Neighbour - Example



5-NN classifies x_q as -, 1-NN as



Voronoi diagram

- If the target function is continuous-valued, kNN calculates the mean of the k nearest neighbours

How to measure distance between examples

- We need a **distance measure** in order to **know who are the neighbours**
- Assume that we have *M* attributes (features). Then one example point $\mathbf{x_i}$ is described by a feature vector $\mathbf{<x_{i1}, x_{i2}, \ldots, x_{iM}>}$.
- The distance between two points $\mathbf{x_i}$ and $\mathbf{x_j}$ is usually defined in terms of **Euclidean distance**:

$$d(x_i, x_j) = \sqrt{\sum_{f=1}^{M} [x_{if} - x_{jf}]^2}$$

When should KNN be used?

- **Use no more than, say, 20 attributes per instance**
- *Advantages:*
  - Training is very **fast**
  - Can learn **complex target functions**
  - Do **not lose any information** contained in the training set
- *Disadvantages:*
  - During classification, kNN has to calculate the **distances between the test case and *all* training examples**
  - Some attributes may be **irrelevant**

Version of kNN:
Distance-Weighted KNN

- Weight **nearer neighbours** more heavily:

$$f(x_q) = \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \quad \textit{where} \quad w_i = \frac{1}{d(x_q, x_i)^2}$$

- This allows us to use *all training examples* instead of just k (**Sheppard's method**), and we call this a global learning method. If only the nearest examples are used, we have a local learning method.

Section 2.9

Clustering and K- Means

Clustering

- Is the process of grouping objects with **similar characteristics**.
  - We need a **similarity measure and a neighbourhood definition**.
- **A similarity measure is not a distance function!**
- In a **supervised** learning process
  - The classification **is known a priori**.
- In a **Unsupervised** learning process
  - Che classification **is NOT known a priori**.

Clustering

- **Cluster**
  - A collection of **similar physical/abstract objects**.
    - (Similar within the same cluster, dissimilar to objects in other clusters)
- *Clustering*
  - *process of grouping a set of objects into clusters*
    - It is an **unsupervised learning** method.
    - It is a **common and important ML task**
    - It has many applications:
      - stand-alone classification tool
      - preprocessing tool for other ML algorithms.

Concepts in Clustering

- First we should define a
  - **distance (similarity measure)** between points
- A good clustering is one where:
  - high intra-cluster similarity
    - the sum of distances between objects in the same cluster are minimized,
  - low inter-cluster similarity
    - while the distances between different clusters are maximized
- Clusters can be evaluated using:
  - **"internal measures"** that are related to the inter/intra cluster distance
  - **"external measures"** that are related to how representative are the current clusters to "true" classes. This is typically highly subjective.
- ***Centroid*** – centre of the cluster (i.e. mean point of the cluster)
- ***Medoid*** - the most centrally located (or most representative) object in a cluster

Distance measures – Examples

- **Euclidean distance** $d(x_i, x_j) = \sqrt{\sum_{f=1}^{nof} (x_{i,f} - x_{j,f})^2}$
- **Minkowski distance** $d(x_i, x_j) = \sqrt[p]{\sum_{f=1}^{nof} (x_{i,f} - x_{j,f})^p}$
- **Manhattan distance** $d(x_i, x_j) = \sum_{f=1}^{nof} | x_{i,f} - x_{j,f} |$

where **nof** is the **n**umber **o**f **f**eatures

Inter/Intra Cluster Similarity Measures

- **Intra-cluster similarity measure**
  - Sum/Min/Max/Avg of the absolute/squared distances between:
  - All pairs of points in the cluster OR
    - Between the **"centroid"** and all points in the cluster OR
    - Between the **"medoid"** and all points in the cluster
- **Inter-cluster similarity measure**
  - Sum the (squared) distance between all pairs of clusters, where the distance between two clusters is defined as:
    - Distance between their centroids/medoids (i.e. spherical clusters)
    - Distance between the closest pair of points belonging to the clusters (i.e., chain shaped clusters)

Is clustering a difficult problem?

- Given n points, and say we would like to cluster them into k clusters
  - How many possible cluster???
- **Answer:** Too many
  - → exponential in terms of n and k
  - → **we can not test exhaustively all possibilities**.
- **Solution:** Iterative optimization algorithms
  - → **Start with an initial clustering and iteratively improve it**
  - → (see K-means)

Classical clustering methods

- Partitioning methods
  - Construct various partitions and then evaluate them by some criterion
  - *k-Means (and EM), k-Medoids*
- Hierarchical methods
  - Create a hierarchical decomposition of the set of objects using some criterion
  - *agglomerative, divisive*
- Model-based clustering methods
  - A model is hypothesized for each of the clusters and the idea is to find the best fit of that model to each other

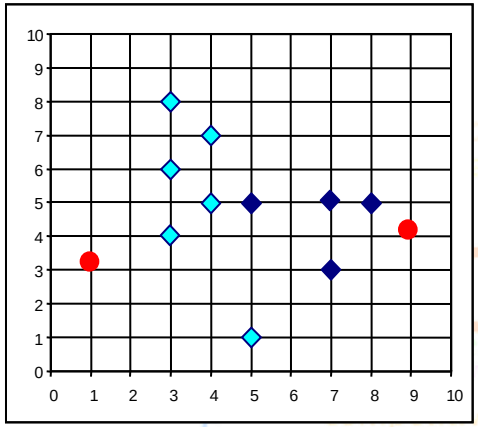
Non-Classical clustering methods

- **Fuzzy clustering algorithms**
  - Fuzzy C-means is the most popular one
- **Neural networks** have been used for clustering
  - Self-Organizing Maps (SOMs – Kohonen, 1984)
  - Adaptive Resonance Theory (ART) networks (Carpenter & Grossberg, 1990)
- **Evolutionary algorithms** based clustering
- **Simulated annealing** based clustering

K-means Algorithm

- First we should specify k
  - the **number of clusters we want to find out**
  - Each cluster will be represented by the **center of the cluster**. Iteratively minimize the objective function (distance to clusters).
- Algorithm:
  1. **Randomly pick k points** (inside the hypervolume containing the pattern set) as the **"centroids"** of the k clusters we want
  2. For each pattern in the data set, **assign the pattern to the cluster** with the **closest centroid**
  3. **Recompute the cluster centroids** using the ***current cluster memberships***
  4. If there is **no (or minimal) change** in the identified clusters between two consecutive iterations **stop, otherwise go to step 2**

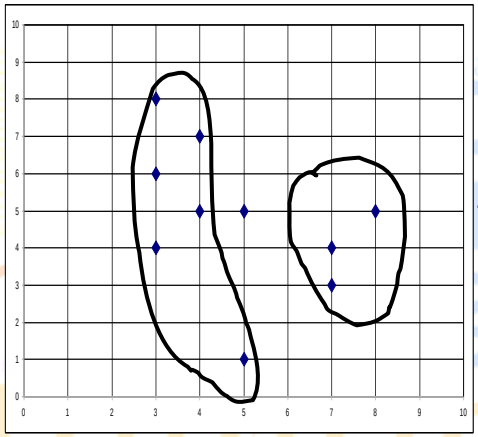
Example of K-Means



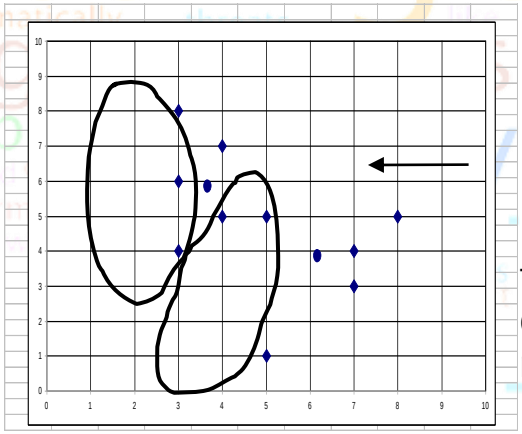
K=2

Arbitrarily choose K object as initial cluster center

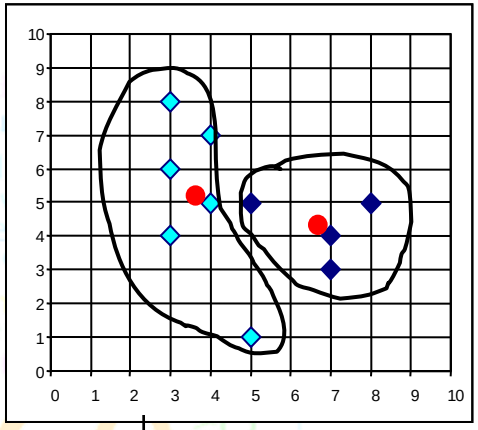
Assign each objects to most similar center



reassign

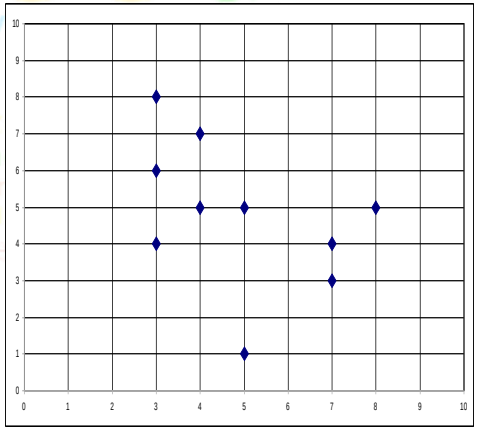


Update the cluster means



Update the cluster means

reassign



Example of K-Means (2)

- Objects: 1, 2, 5, 6,7 (1-dimensional objects)
- We want to find 2 clusters (k=2). Numerical difference is used as distance.
- K-means:
  - Randomly select 5 and 6 as centroids;
  - => Two clusters {1,2,5} and {6,7}; meanC1=8/3, meanC2=6.5
  - => {1,2}, {5,6,7}; meanC1=1.5, meanC2=6
  - => no change.
  - *Aggregate dissimilarity*
    - sum of squared distances between each point (in all clusters)
    - and *its* cluster center--(intra-cluster distance)

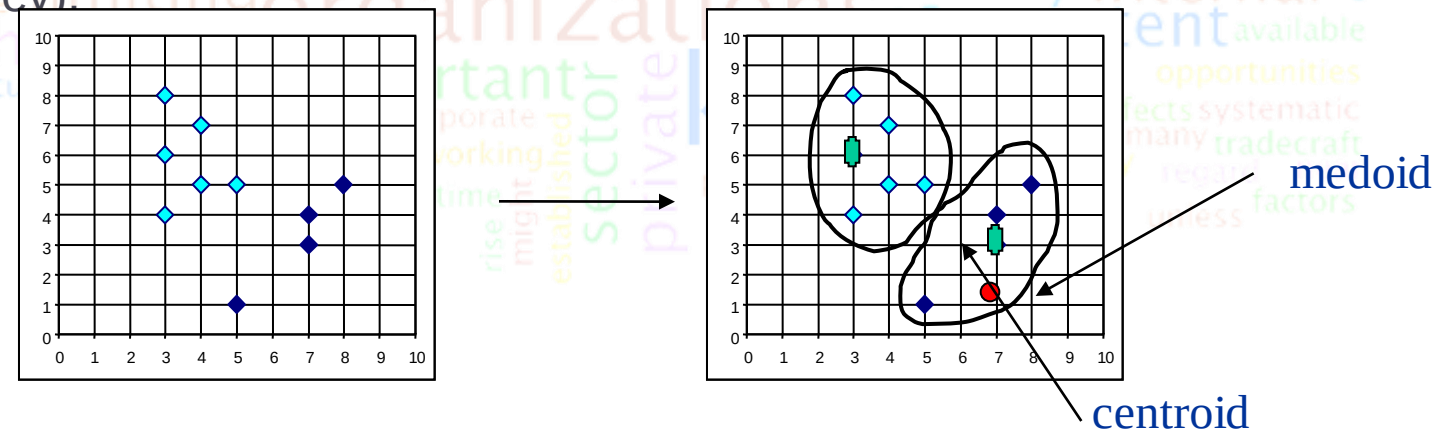
$|1-1.5|^2 = 0.5^2 + 0.5^2 + 1^2 + 0^2 + 1^2 = 2.5$

Problems of K-means

- We need to specify k in advance!
  - Solution: May need to try out several k
- Tends to go to local minima that depend on the selection of starting centroidsSolution: Run the algorithm with different starting points
- Assumes clusters are spherical in vector space (Euclidean topology), could be foils, donuts and others shapes!!
- Sensitive to coordinate changes, weighting, etc.
- K-means is sensitive to “outliers” (does not recognize them) an object with an extremely large value (outlier) may substantially distort the distribution of the data

Problems of K-means (2)

- Outlier problem can be handled by **K-medoid or neighborhood**-based algorithms
- **K-Medoids method:**
  - Use the **medoids** (the **most centrally located** object in a cluster) instead of computing the **mean** value of the objects in a cluster (centroids) as a reference point for that cluster. See for example PAM (Partitioning Around Medoids) algorithm - (Kaufman & Rousseeuw, 1987 Wiley).



Variants of K-means

- **Recompute the centroids after every change** (or few changes), rather than after all the patterns are re-assigned
  - *Improves the convergence speed*
- Starting centroids (seeds) may determine to converge to local minima, as well as the rate of convergence
  - Use **heuristics** to pick good seeds
  - Run **K-means M times** and **pick the best clustering obtained**

Another Approach: Partition Clustering

1. Divide n object into k group called partitions

2. A partition $P=\{p_1,\ldots,p_k\}$ of a set $X=\{x_1,\ldots,x_n\}$, satisfy:
   a) $p_i \subseteq X$, $p_i \neq \emptyset$
   b) $X = \bigcup p_i$
   c) $p_i \cap p_j = \emptyset \quad i \neq j$

Another Approach: Hierarchical Clustering

- Same data but ...
  - There are a **tree like structure** to **decide how to perform the aggregation**.
- This involves **more information than partition clustering**
  - Because we could **split clusters in different types**.
- The final clustering are perform by
  - the leaves of the tree structure.