

Machine Learning

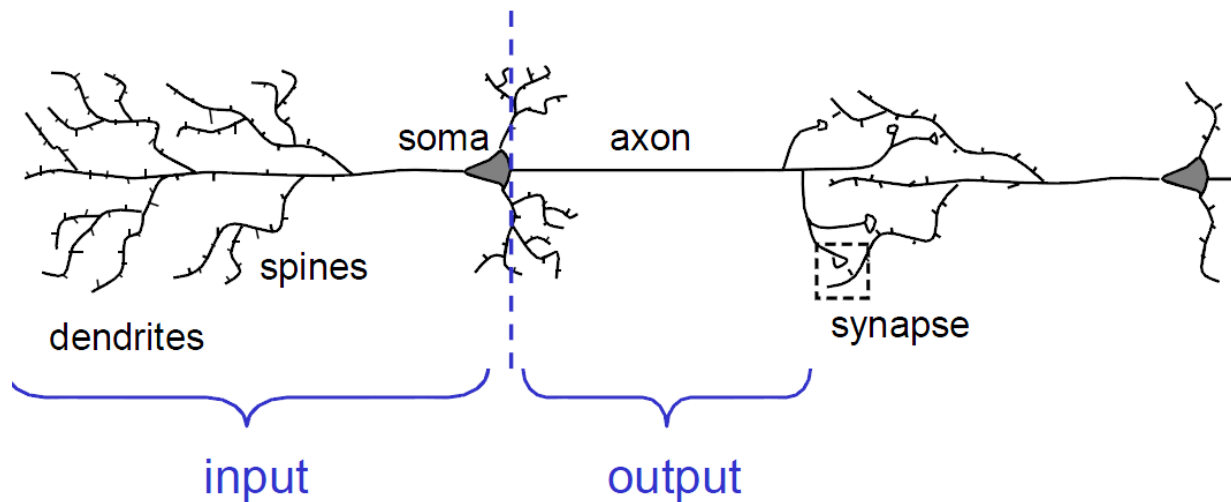
Syllabus

- Introduction
- Mathematics Preliminaries
- Bayesian Learning
- Linear Models for Classification
- Linear Models for regression
- Decision Trees
- Neural Networks
- Instance Based Learning
- Ensemble Learning
- SVM
- Unsupervised Learning

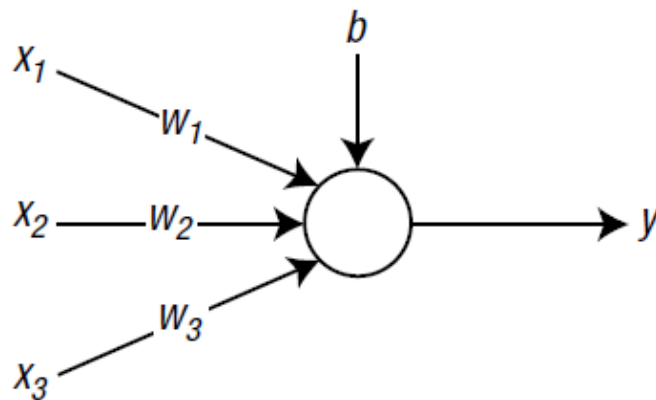
Supervised Learning

Neural Networks:

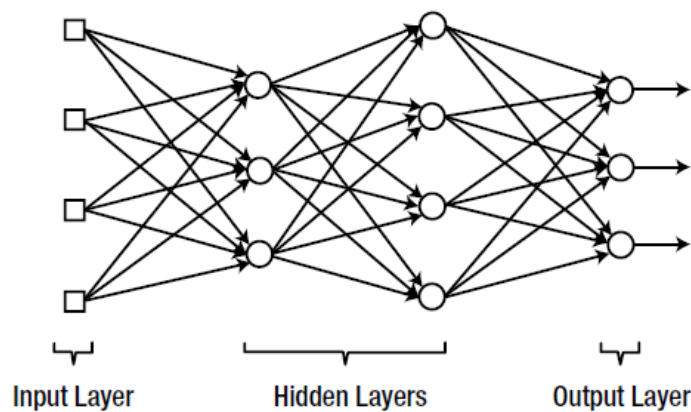
- The central idea of *artificial neural networks* (ANN) is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features.
- This machine learning method originated as an algorithm trying to mimic neurons and the brain.
- The brain has extraordinary computational power to represent and interpret complex environments.
- ANN attempts to capture this mode of computation.
- Specifically, ANN is a formalism for representing functions, inspired from biological systems and composed of parallel computing units, each computing a simple function.



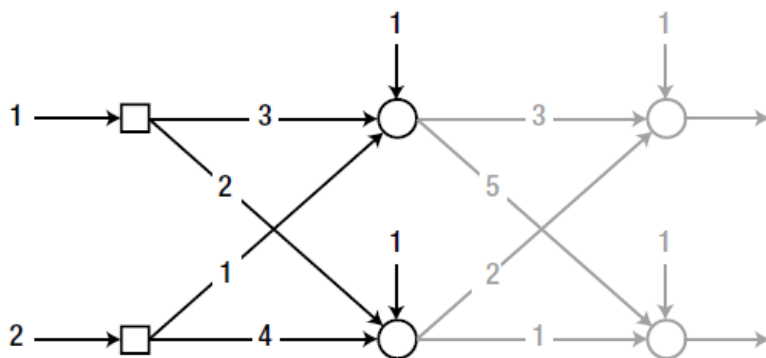
- Parts of the neuron: dendrites, soma (body), axon, synapses



The circle and arrow of the figure denote the node and signal flow, respectively. x_1 , x_2 , and x_3 are the input signals. w_1 , w_2 , and w_3 are the weights for the corresponding signals. Lastly, b is the bias, which is another factor associated with the storage of information. In other words, the information of the neural net is stored in the form of weights and bias. The input signal from the outside is multiplied by the weight before it reaches the node. Once the weighted signals are collected at the node, these values are added to be the weighted sum. The weighted sum of this example is calculated as follows: $v = wx + b = [(w_1 * x_1) + \dots] + b$. Finally, the node enters the weighted sum into the activation function and yields its output. The activation function determines the behavior of the node. $y = j(v)$ of this equation is the activation function. Many types of activation functions are available in the neural network.



Example:



The first node of the hidden layer calculates the output as:

Weighted sum: $v = (3 \times 1) + (1 \times 2) + 1 = 6$

Output: $y = j(v) = v = 6$

In a similar manner, the second node of the hidden layer calculates the output as:

Weighted sum: $v = (2 \times 1) + (4 \times 2) + 1 = 11$

Output: $y = j(v) = v = 11$

The weighted sum calculations can be combined in a matrix equation as follows:

$$v = \begin{bmatrix} 3 \times 1 + 1 \times 2 + 1 \\ 2 \times 1 + 4 \times 2 + 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 11 \end{bmatrix}$$

The weights of the first node of the hidden layer lay in the first row, and the weights of the second node are in the second row. This result can be generalized as the following equation:

$v = Wx + b$ where x is the input signal vector and b is the bias vector of the node.

Output = $f(u) = v = [41 \ 42]$;

Training of NN-Delta rule:

The systematic approach to modifying the weights according to the given information is called the learning rule. Consider a single-layer neural network, as shown in Figure.

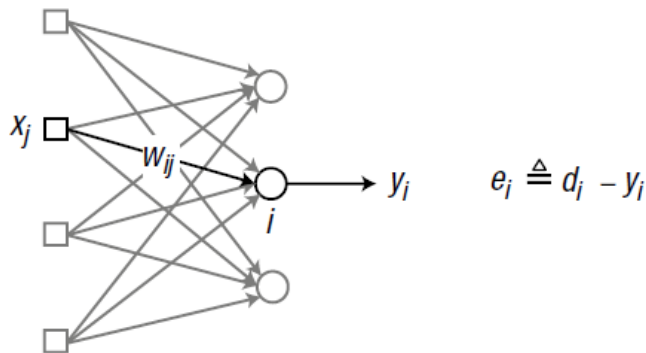


Figure 2-11. A single-layer neural network

In the figure, d_i is the correct output of the output node i . "If an input node contributes to the error of the output node, the weight between the two nodes is adjusted in proportion to the input value, x_j and the output error, e_i ."

This rule can be expressed in equation as: $new\ w_{ij} = old\ w_{ij} + \alpha e_i x_j$

where

x_j = The output from the input node j , ($j=1, 2, 3$)

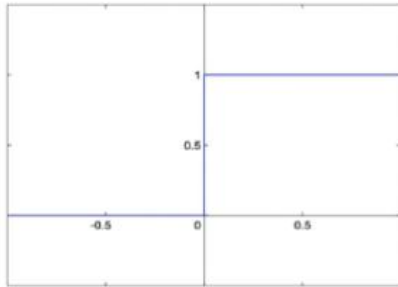
e_i = The error of the output node i

w_{ij} = The weight between the output node i and input node j

α = Learning rate ($0 < \alpha \leq 1$)

The learning rate, α , determines how much the weight is changed per time. If this value is too high, the output wanders around the solution and fails to converge. In contrast, if it is too low, the calculation reaches the solution too slowly.

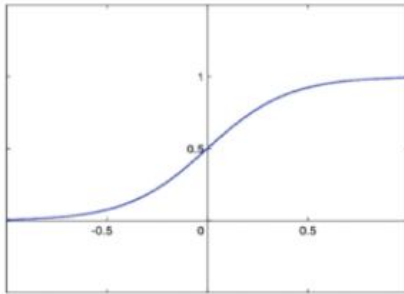
Activation functions:



Step function

The output is a certain value A1, if the input sum is above a certain threshold and A0 if the input sum is below a certain threshold.

When we want to classify an input pattern into one of two groups, we can use a binary classifier with a step activation function.



Sigmoid function

Has the property of being similar to the step function, but with the addition of a region of uncertainty.

Sigmoid functions in this respect are very similar to the input-output relationships of biological neurons.

Sigmoid function

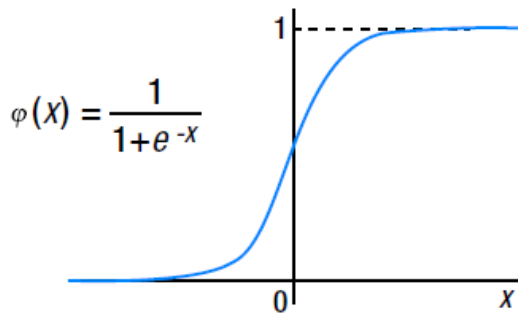


Figure 2-14. The sigmoid function defined

ReLU :

ReLU function is defined as

$$\varphi(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

$$= \max(0, x)$$

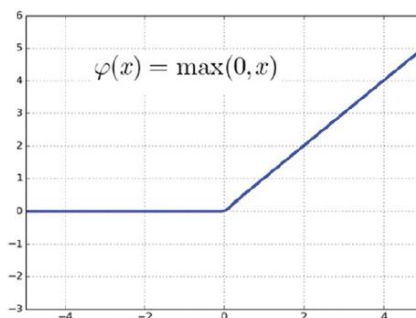


Figure 5-3. The ReLU function

Although the neural network is applicable to both classification and regression, it is seldom used for regression. In the application of the neural network to classification, the output layer is

usually formulated differently depending on how many groups the data should be divided into. For binary classification, a single output node is sufficient for the neural network

Decision Hyperplanes and Linear Separability

If we have two inputs, then the weights define a decision boundary that is a one dimensional straight line in the two dimensional **input space** of possible input values.

If we have n inputs, the weights define a decision boundary that is an $n-1$ dimensional **hyperplane** in the n dimensional input space:

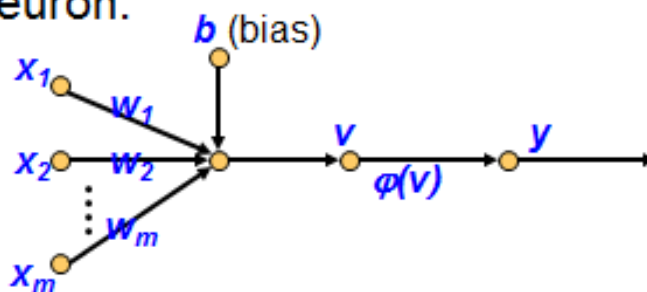
$$w_1x_1 + w_2x_2 + \dots + w_nx_n - q = 0$$

This hyperplane is clearly still linear (i.e. straight/flat) and can still only divide the space into two regions. We still need more complex transfer functions, or more complex networks, to deal with XOR type problems.

Problems with input patterns which can be classified using a single hyperplane are said to be **linearly separable**. Problems (such as XOR) which cannot be classified in this way are said to be **non-linearly separable**.

Perceptron: Neuron Model

- Uses a non-linear (McCulloch-Pitts) model of neuron:



- ϕ is the **sign** function:

$$\phi(v) = \begin{cases} +1 & \text{IF } v \geq 0 \\ -1 & \text{IF } v < 0 \end{cases} \quad \text{Is the function } \text{sign}(v)$$

Perceptron Learning

For simple Perceptrons performing classification, we have seen that the decision boundaries are hyperplanes, and we can think of **learning** as the process of shifting around the hyperplanes until each training pattern is classified correctly. Somehow, we need to formalise that process of “shifting around” into a systematic algorithm that can easily be implemented on a computer.

The “shifting around” can conveniently be split up into a number of small steps. If the network weights at time t are $w_{ij}(t)$, then the shifting process corresponds to moving them by an amount $Dw_{ij}(t)$ so that at time $t+1$ we have weights $w_{ij}(t+1) = w_{ij}(t) + Dw_{ij}(t)$. It is convenient to treat the thresholds as weights, as discussed previously, so we don't need separate equations for them.

Perceptron Learning Rule

1. Initialize weights at random
2. For each training pair/pattern $(\mathbf{x}, \mathbf{y}_{\text{target}})$
 - Compute output y
 - Compute error, $\delta = (y_{\text{target}} - y)$
 - Use the error to update weights as follows:
$$\Delta w = w - w_{\text{old}} = \eta * \delta * x \quad \text{or} \quad w_{\text{new}} = w_{\text{old}} + \eta * \delta * x$$

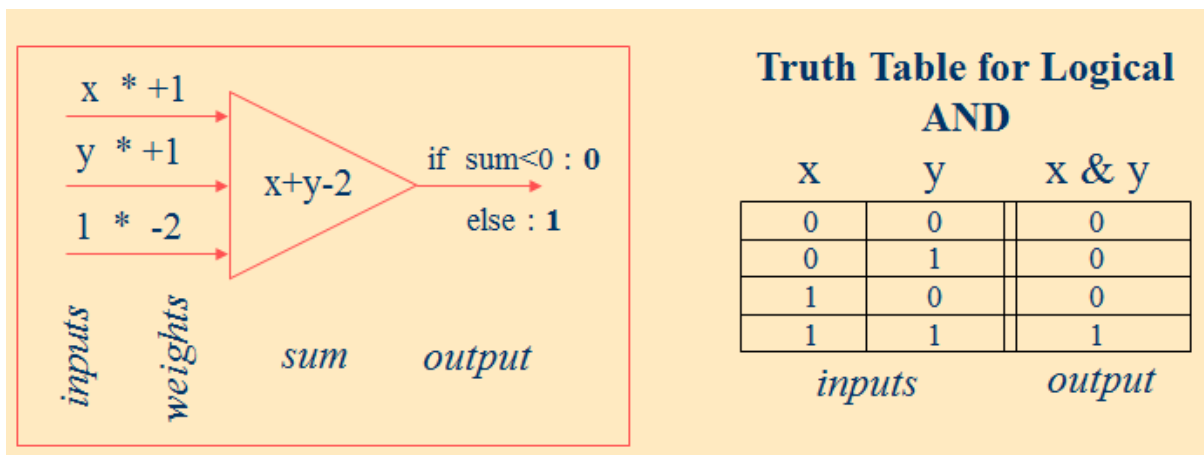
where η is called the **learning rate** or **step size** and it determines how smoothly the learning process is taking place.
3. Repeat 2 until convergence (i.e. error δ is zero)

The **Perceptron Learning Rule** is then given by

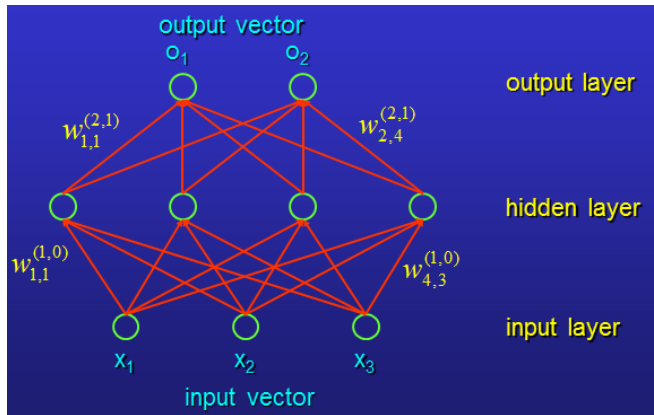
$$w_{\text{new}} = w_{\text{old}} + \eta * \delta * x$$

where

$$\delta = (y_{\text{target}} - y)$$

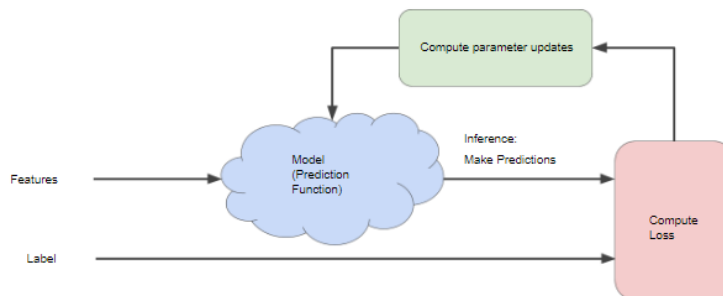


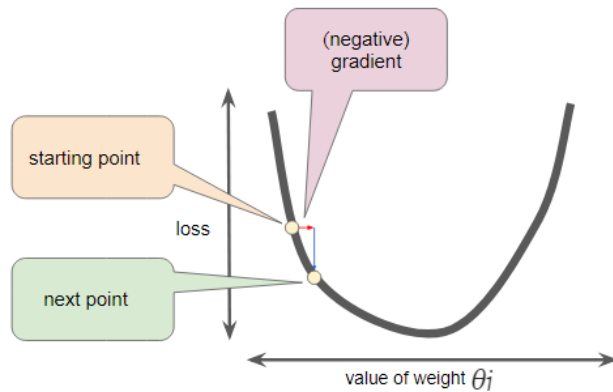
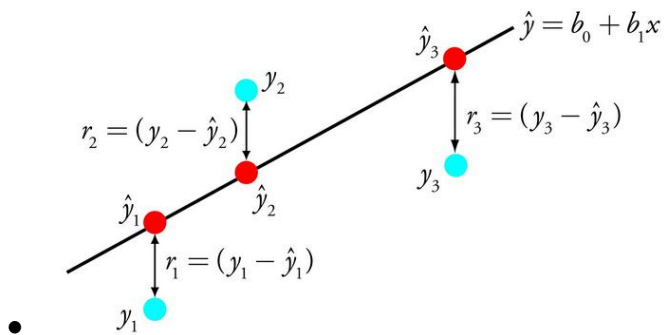
Multilayer Perceptron:



Gradient Descent:

- Perceptron learning rule is for linearly separable data, separating the input space into two by the hyper-plane $W^T X + b = 0$.
- Gradient descent is for data which are not.
- ANN can be seen as a regression without the threshold, as is the case with gradient descent, searching for the optimal weight vector (hypothesis vector in the hypotheses space)
- Suppose we are using Regression, predicting sales (y) for amount spent in ads (x). Predicted value is $\hat{y} = \theta_0 + \theta_1 x_1$



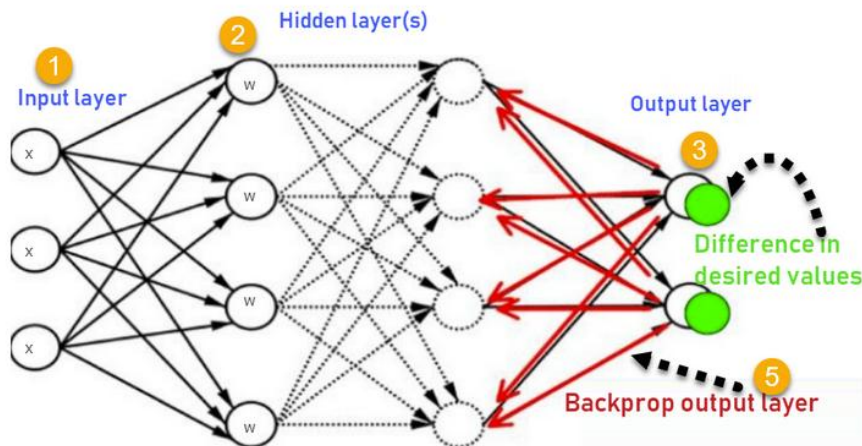


$$o(\vec{x}) = \vec{w} \cdot \vec{x} \quad E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d) (-x_{id}) \end{aligned}$$

Backpropagation:

- BP networks propagate the error backwards for training the network and minimizing the errors



- The activation function in BP is Sigmoid function $F(a)=1/[1+\exp(-a)]$
- Weight training is $W_{ji}(t+1)=W_{ji}(t)+\Delta W_{ji}$.

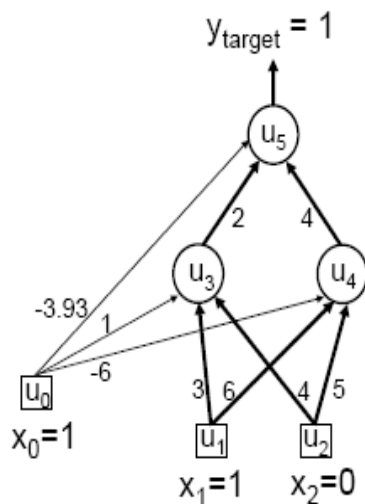
$$\Delta W_{ji} = \eta \delta_j O_i$$

$$\delta_j = O_j(1 - O_j)(T_j - O_j) \quad [\text{output units}]$$

$$\delta_j = O_j(1 - O_j) \sum_k \delta_k W_{kj} \quad [\text{hidden units}]$$

BackPropogation-XOR gate:

- Construct a MLP with 2 input neurons, one hidden neuron, one output neuron and a bias for both layers with the weights
- $W_{13}=0.02, W_{14}=0.03, W_{12}=-0.02, W_{23}=0.01, W_{24}=0.02, W_{1b}=-0.01, W_{2b}=-0.01$
- Note: Numbering from top to bottom.
- Activation fn is sigmoid.



Current state:

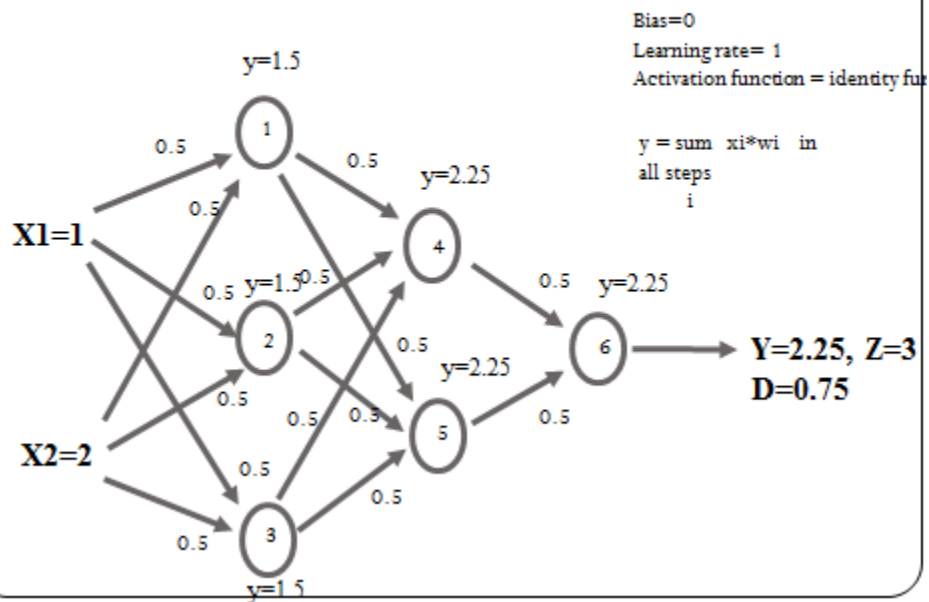
- Weights on arrows e.g. $w_{13} = 3, w_{35} = 2, w_{24} = 5$
- Bias weights, e.g. bias for unit 4 (u_4) is $w_{04} = -6$

Training example (e.g. for logical OR problem):

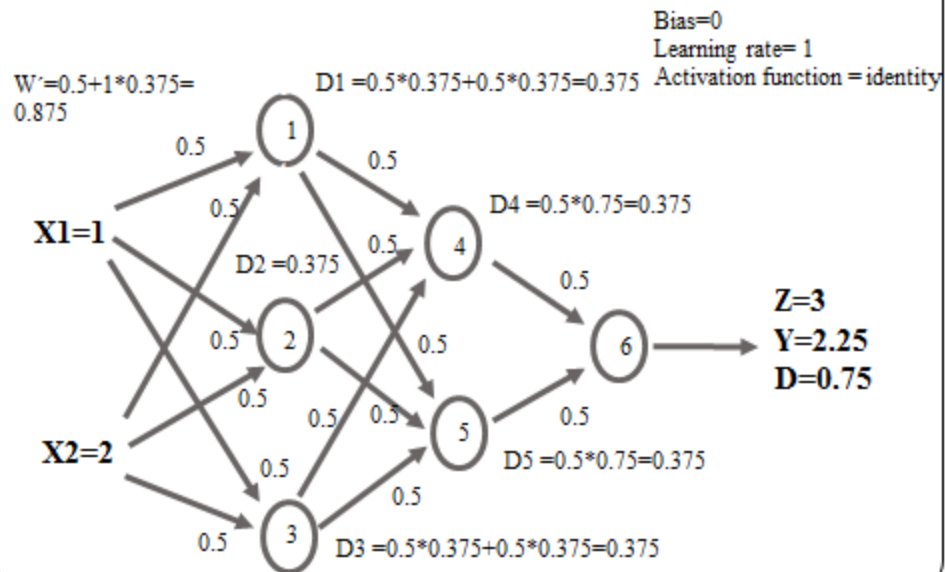
- Input pattern is $x_1=1, x_2=0$
- Target output is $y_{\text{target}}=1$

i	j	w_{ij}	δ_j	y_i	Updated w_{ij}
0	3	1	0.0043	1.0	1.0004
1	3	3	0.0043	1.0	3.0004
2	3	4	0.0043	0.0	4.0000
0	4	-6	0.1225	1.0	-5.9878
1	4	6	0.1225	1.0	6.0123
2	4	5	0.1225	0.0	5.0000
0	5	-3.92	0.1225	1.0	-3.9078
3	5	2	0.1225	0.9820	2.0120
4	5	4	0.1225	0.5	4.0061

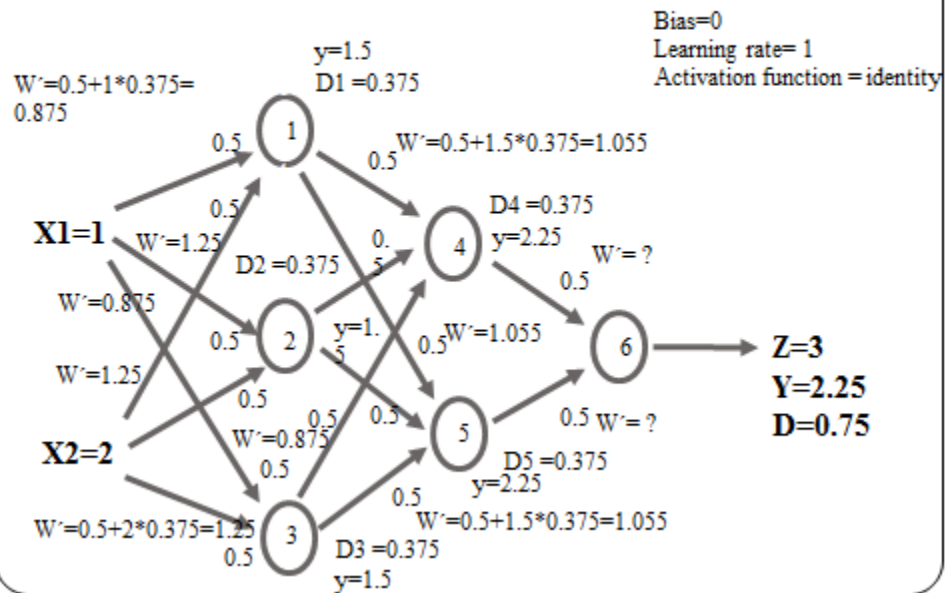
Example: Feed forward



Example: Backpropagation of error



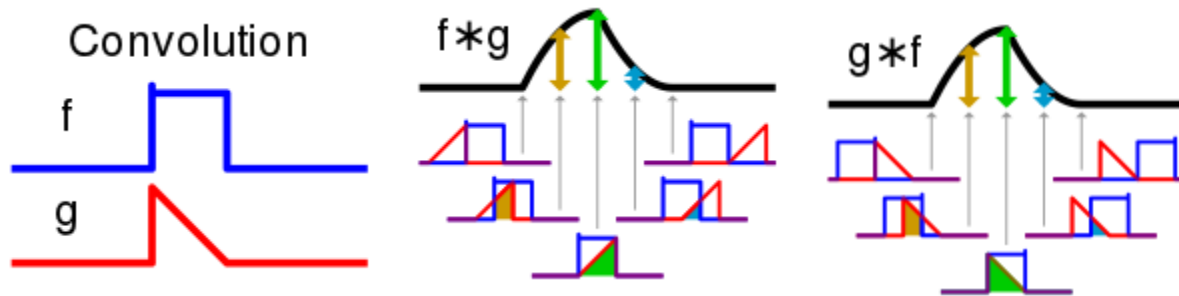
Example: Re-calculation of weights



Convolution:

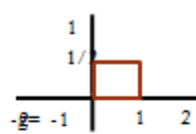
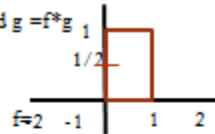
- Convolution is mathematical operation of two functions $f(x)$ and $g(x)$, defined as

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(a)g(x-a)da$$

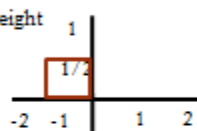


Convolution Example

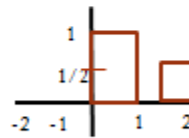
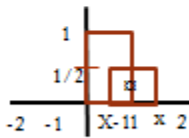
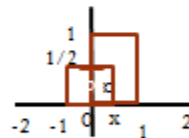
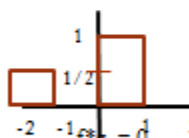
Compute the convolution of f and $g = f * g$



Reflect the weight function g



Slide g



Result

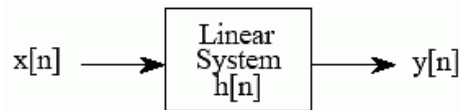
$$f(x) * g(x) = \begin{cases} x/2 & 0 \leq x \leq 1 \\ 1 - x/2 & 1 \leq x \leq 2 \\ 0 & \text{elsewhere} \end{cases}$$

- Convolution is mathematical operation of two functions $f(x)$ and $g(x)$, defined as

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(a)g(x-a)da$$

- In signal processing, it is combining two signals to create a third signal. Systems here are described by their Impulse response.
- if we know a system's impulse response, then we can calculate what the output will be for any possible input signal. This means we know *everything* about the system.
- If the system being considered is a *filter*, the impulse response is called the **filter kernel**, the **convolution kernel**, or simply, the **kernel**.
- In image processing, the impulse response is called the **point spread function**.

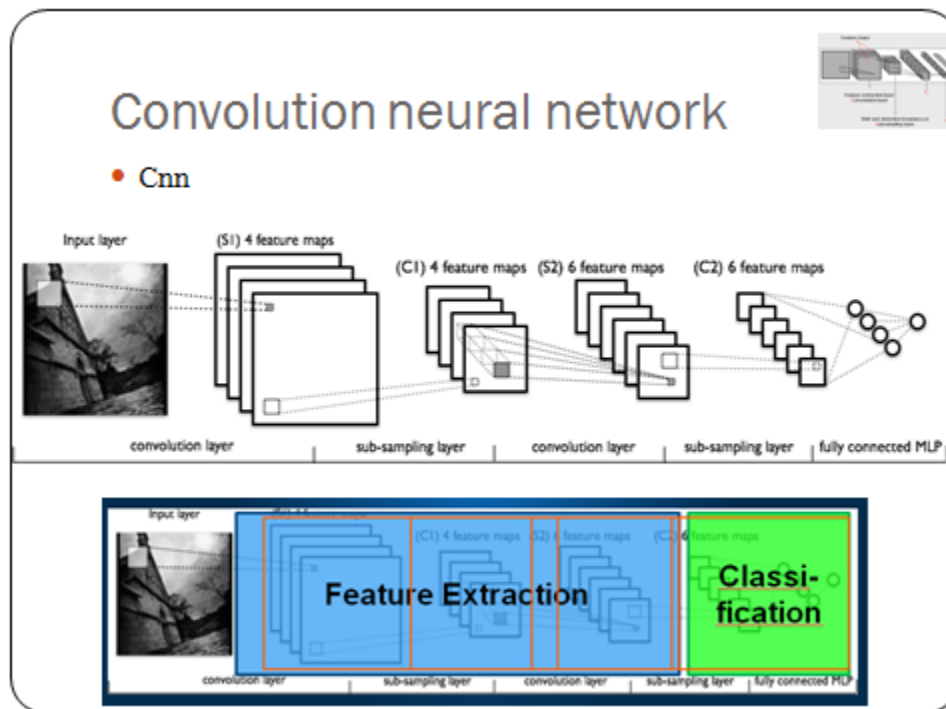
In Linear systems, convolution is used to describe the relationship between three signals of interest: the input signal, the impulse response, and the output signal



$$x[n] * h[n] = y[n]$$

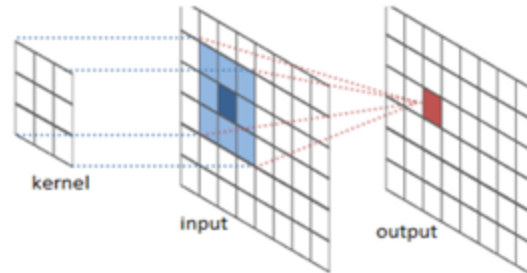
Convolution neural network:

- A simple MLP has general issues like
- Number of parameters is large
- Invariance is not addressed
- Topology of input data is ignored
- Usually black-white patterns
- CNN's try to address these by being locally sensitive, feedforward net which can be trained by a BPN, extracting topological properties of an image



Convolution neural network

- Convolution
- Sparse connectivity
- Parameter sharing



- Filter
- Slide or Stride
- (Subsampling) Pooling
- Padding

Convolution neural network

- Suppose, we want to detect beaks of birds,

1	-1	-1
-1	1	-1
-1	-1	1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Dot product

→

3

-1

If stride=2

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

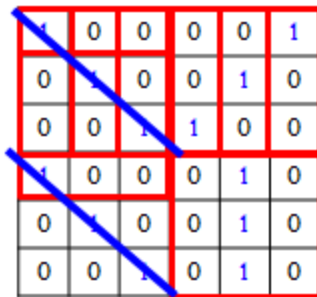
6 x 6 image

3

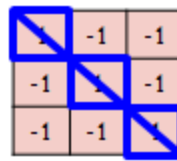
-3

Convolution

stride=1



6 x 6 image

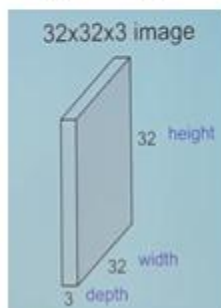


Filter 1

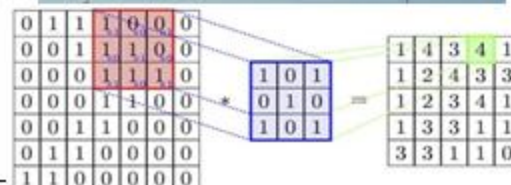


Convolution neural network

- Feature Learning Phase:
- The feature learning phase in a CNN consists of an arbitrary number of pairs of Convolution and Pooling layers
- Input Image



Filter+Convolved Image



- The **fully connected layers** takes as input, a flattened array representing the activation maps of high level features from earlier layers and outputs an N dimensional vector.
- If a Softmax activity function is used, each number in this N dimensional vector represents the probability of a certain class.
- For example, if the resulting vector for a digit classification program is [0, 0.1, 0.1, 0.75, 0, 0, 0, 0, 0, 0.05], then this represents a 10% probability that the image is a 1, a 10%

probability that the image is a 2, a 75% probability that the image is a 3, and a 5% probability that the image is a 9.

Convolution neural network

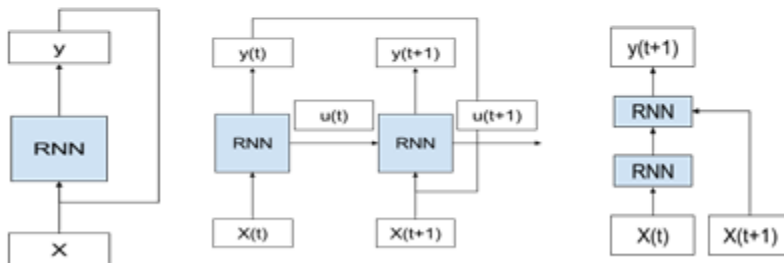
Padding

- Sometimes ,the sliding process can miss to apply the filter to some input array element.In such cases, Padding is employed to include all input.

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Recurrent Neural Networks

- RNN's are Temporal dynamic ANN's.
- RNN has memory(persistent state).
- RNN's have cycles or feed-forward acycles.
- They are very much suited for Time series data.



Instance Based Learning:

- Parametric
 - Naïve Bayes
 - MLE
- Non-parametric
 - k Nearest Neighbour Learning(kNN)
 - Locally weighted regression(LWR)
 - Radial Basis Functions (RBF)

KNN:

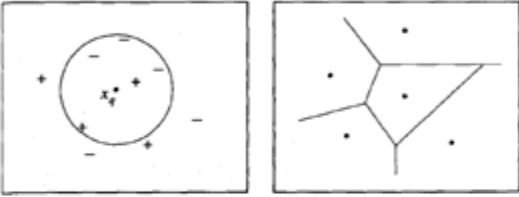
KNN is a **non-parametric, lazy** learning algorithm based on feature similarity.

- Algorithm
- Step1: Select a number k with every new instance.
- Step2: Find the distance between the new instance and the stored data to find out the k closest ones.
- Step3: From the selected k closest ones, find the most common class and assign that to the new instance.
- Distance (similarity) measure - value indicating how similar data points are .
- distance = inverse of similarity.
- Distances:
 - for all objects x_i, x_j , $\text{dist}(x_i, x_j) \geq 0$, $\text{dist}(x_i, x_j) = \text{dist}(x_j, x_i)$
 - for any object x_i , $\text{dist}(x_i, x_i) = 0$
 - $\text{dist}(x_i, x_j) \leq \text{dist}(x_i, x_k) + \text{dist}(x_k, x_j)$
- Distance measures:
 - The vectors(instances) are assumed to be present in an Euclidean space like .
- Euclidean:
$$\sqrt{\sum_{f=1}^{|features|} (x_{i,f} - x_{j,f})^2}$$
- In this method, all training examples are stored. Whenever a query instances arrives, it is compared with every stored for the distance and based on chosen k, a local group is arrived at before interpolation.
- So, the issues are storing, retrieving, distance calculation , speed.
- Another major issue is that the Distance is based on all the attributes/features of the instance. When we have a large number of features and if only a few are relevant, we could have errors. This is termed as *Curse of Dimensionality*.

- Curse of Dimensionality is addressed sometimes by weighting the axes using validation methods.

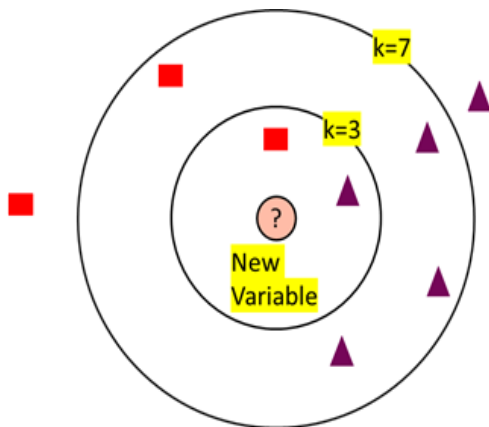
knn

- **KNN** is a **non-parametric, lazy** learning algorithm based on feature similarity.



- The shape of this decision surface induced by 1-NEAREST NEIGHBOUR over the entire instance space.
- The decision surface is a combination of convex polyhedra surrounding each of the training examples.
- For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example.
- Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the *Voronoi diagram* of the set of training examples

- Algorithm
- Step1: Select a number k with every new instance.
- Step2: Find the distance between the new instance and the stored data to find out the k closest ones.
- Step3: From the selected k closest ones, find the most common class and assign that to the new instance.



knn

- Example3:
- Suppose we wish to find out the size of
- t shirt of a person based on his height
- and weight, based on the following data

Height	Weight	T-shirt size	Distance
158	38	M	4.2+26+1
158	39	M	3.603331
158	43	M	3.603331
160	39	M	2.236068
160	60	M	1.41+21+
163	60	M	2.236068
163	61	M	2
160	64	L	3.162278
163	64	L	3.603331
165	61	L	4
165	62	L	4.123106
165	65	L	3.603331
168	65	L	3.062238
168	62	L	7.071068
168	66	L	3.602325
170	63	L	9.219544
170	64	L	9.456333
170	68	L	11.40175
161	61	?	

Locally weighted regression:

- LWR is a generalization of the knn method.
- We chose a Local set, form of regression like linear quadratic etc and then regress using the local set of the query instance.
- The phrase "locally weighted regression" is called
 - **local** because the function is approximated based *a* only on data near the query point,
 - **weighted** because the contribution of each training example is weighted by its distance from the query point, and
 - **regression** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.
- Actually, locally weighted regression in which the target function f is approximated near \mathbf{x} , is using a linear function of the form

$$\hat{f}(\mathbf{x}) = w_0 + w_1 a_1(\mathbf{x}) + \dots + w_n a_n(\mathbf{x})$$

- As before, $a_i(\mathbf{x})$ denotes the value of the i th attribute of the instance \mathbf{x} and the coefficients $w_0 \dots w_n$, are found using the local set selected which minimizes the error

in fitting such linear functions.

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$

- Recall from linear regression, for a given data set (x_i, y_i) , we try to find a function(hypothesis) y such that $y=mx+c$ where m and c are the “parameters” we estimate.
- In general we write this as $h_\theta = \theta_0 + \theta_1 x$
- Our objective is, given a training set, how do we pick, or learn, the parameters θ ?
- To make $h(x)$ close to y , we define the cost function and minimize it , based on theta.

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2.$$

- The minimizing of the cost function is using methods like gradient descent.
- If we assume, we have a single data point,

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2. \quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta).$$

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_{\theta}(x) - y)^2 \\
&= 2 \cdot \frac{1}{2} (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_{\theta}(x) - y) \\
&= (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^n \theta_i x_i - y \right) \\
&= (h_{\theta}(x) - y) x_j
\end{aligned}$$

Locally weighted regression

- In the original linear regression algorithm, to make a prediction at a query point x (i.e., to evaluate $h(x)$), we would:
 1. Fit θ to minimize $\sum_i (y(i) - \theta^T x(i))^2$.
 2. Output $\theta^T x$.
- In contrast, the locally weighted linear regression algorithm does the following:
 1. Fit θ to minimize $\sum_i w(i) (y(i) - \theta^T x(i))^2$.
 2. Output $\theta^T x$.

Locally weighted regression

- Here, the $w(i)$'s are non-negative valued weights. Intuitively, if $w(i)$ is large for a particular value of i , then in picking θ , we'll try hard to make $(y(i) - \theta^T x(i))^2$ small. If $w(i)$ is small, then the $(y(i) - \theta^T x(i))^2$ error term will be pretty much ignored in the fit. A standard choice for the weights is

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$$

- The τ is called the bandwidth parameter which controls the rate at which the weight changes.
- Note that, if $|x(i) - x|$ is small, then $w(i)$ is close to 1; and if $|x(i) - x|$ is large, then $w(i)$ is small.

Locally weighted regression

- Let $x=5$, $x^1=4.9$ and $x^2=3$. Then the weights are,

$$w^1 = \exp\left[\frac{-(4.9-5)^2}{2(0.5)^2}\right] = 0.9802$$

$$w^2 = \exp\left[\frac{-(3-5)^2}{2(0.5)^2}\right] = 0.000335$$

- The cost function is

$$J(\theta) = 0.9802 * (\theta^T x^1 - y^1)^2 + 0.000335 * (\theta^T x^2 - y^2)^2$$

- We choose θ which minimizes the J .
- Note the exponential decrease in the weights based on the distance of the training instances from the query point.

- In parametric regression (general) once we estimate θ , we don't use the data anymore and also we make assumptions about the distributions of the data.
- In non-parametric, for every new query x , we calculate θ from a locally distributed data subset. Also, we store the original data for future and there is no assumption about the original data's distribution.

Radial Basis Function:

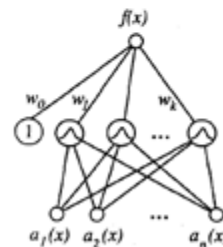
- $f(x) = w_0 + \sum_{u=1}^k w_u K_u(d(x_u, x))$
- $K_u(d(x_u, x))$ is a **kernel function** that decreases as the distance $d(x_u, x)$ increases (usually Gaussian)
- k is a user-defined constant that specifies the number of kernel functions to be included.
- Each kernel is local but the f can be global.

$$K_u(d(x_u, x)) = e^{-\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

- A Gaussian kernel is

Radial Basis function

- The function given by Equation
- $f(x) = w_0 + \sum_{u=1}^k w_u K_u(d(x_u, x))$
- can be viewed as describing a two-layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values.
- An example radial basis function
- (RBF) network is illustrated in Figure



- Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ_u^2 : that define its kernel function $K_u(d(x_u, x))$. Second, the weights w , are trained to maximize the fit of the network to the training data, using the global error criterion. Because the kernel functions are held fixed during this second stage, the linear weight values w , can be trained very efficiently.

SVM

Linearly separable:

- Margin
- Maximal margin
- Soft margin
- Outliers
- Hyper-plane
- Kernel-Polynomial, Radial(Rbasis)
- VC Dim
- SVM's offer solution to binary classifications which are complex.
- SVMs are important because of
 - (a) theoretical reasons:
 - Robust to very large number of variables and small samples
 - Can learn both simple and highly complex classification models
 - Employ sophisticated mathematical principles to avoid over-fitting.
 - (b) superior empirical results

SVM:

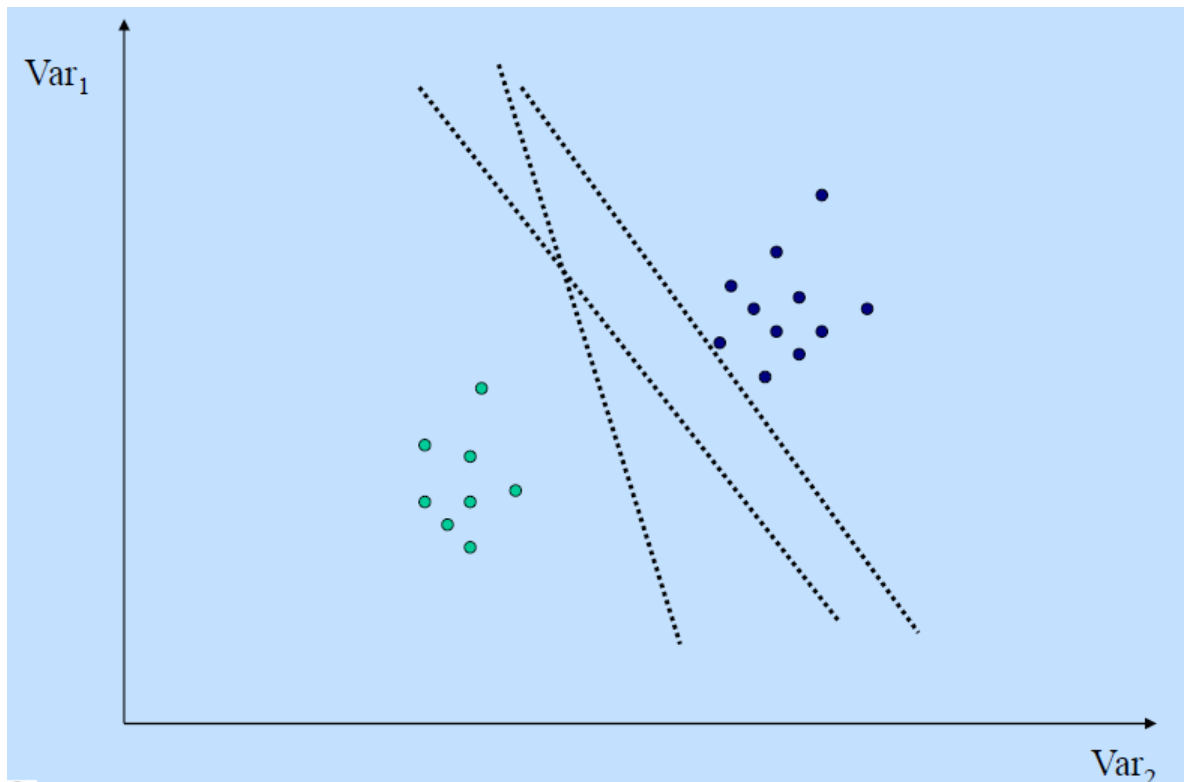
Decision surface: a hyperplane in **feature** space

- One of the most important tools in the machine learning toolbox
- In a brief:
 - map the data to a predetermined very high-dimensional space via a kernel function
 - Find the hyperplane that maximizes the margin between the two classes
 - If data are not separable -find the hyperplane that maximizes the margin and minimizes the (weighted average of the) misclassifications

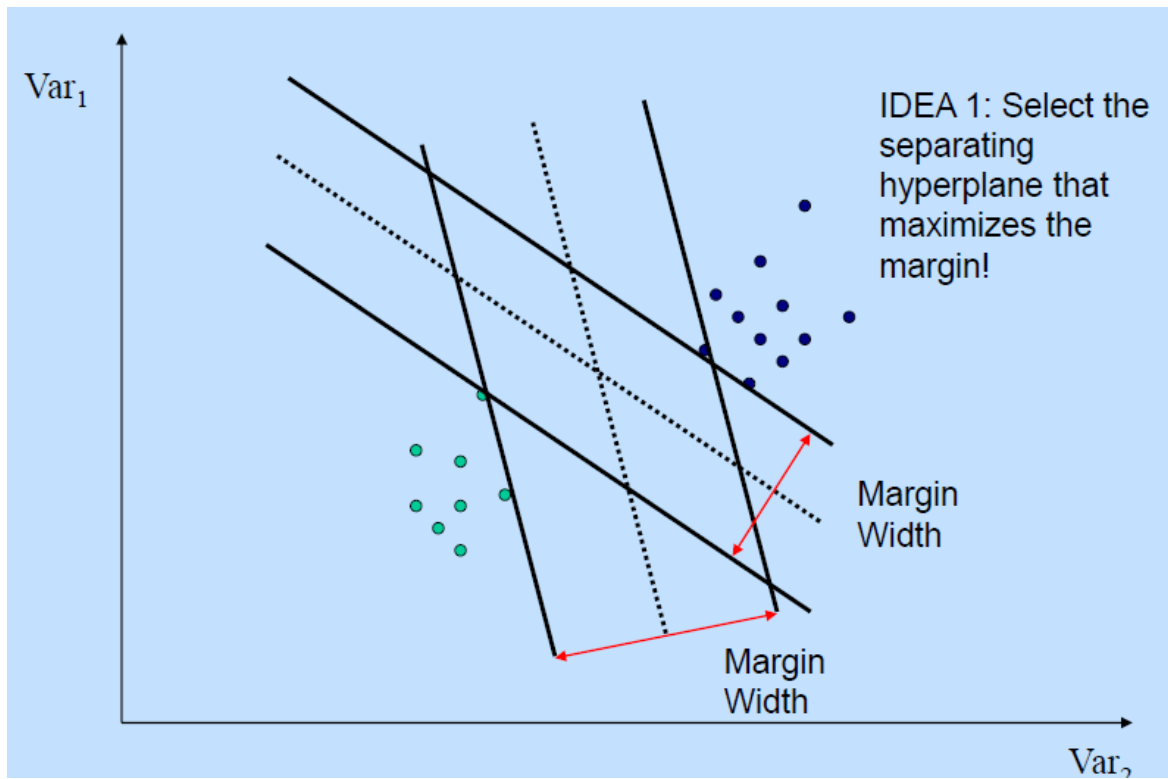
Three main ideas:

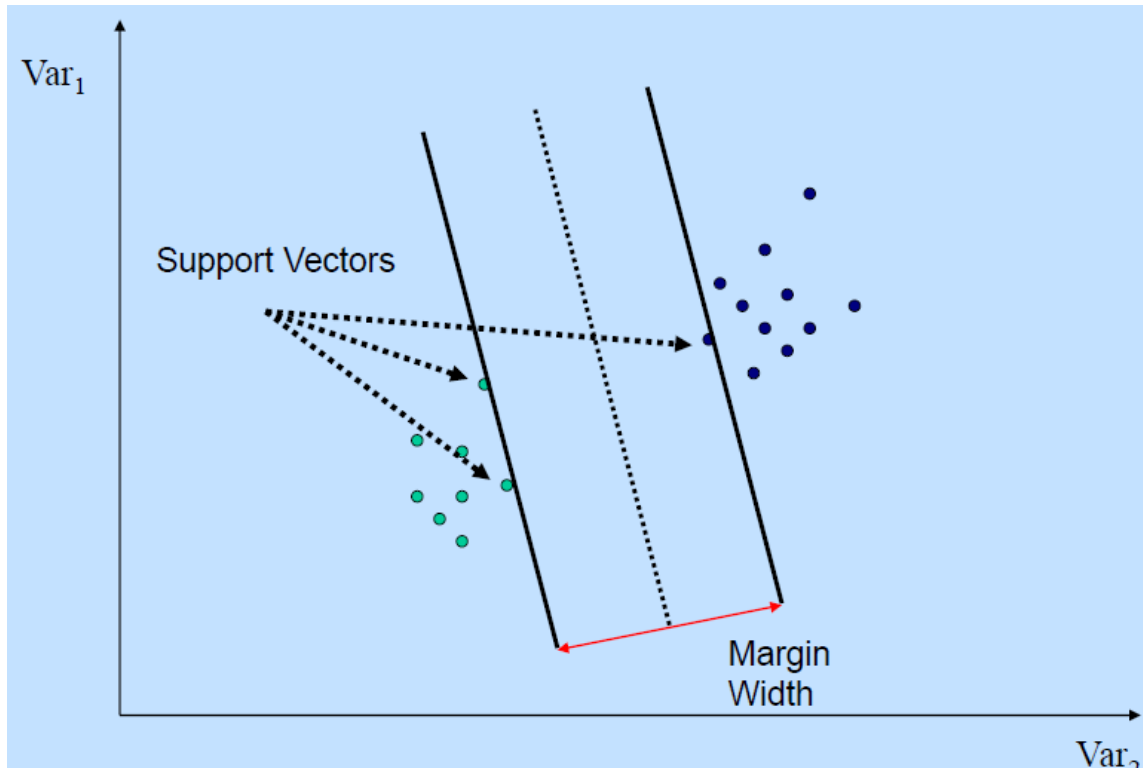
1. Define what an optimal hyperplane is (taking into account that it needs to be computed efficiently): maximize margin
2. Generalize to non-linearly separable problems: have a penalty term for misclassifications
3. Map data to high dimensional space where it is easier to classify with linear decision surfaces: reformulate problem so that data are mapped implicitly to this space

Alternative regression lines(classifiers)

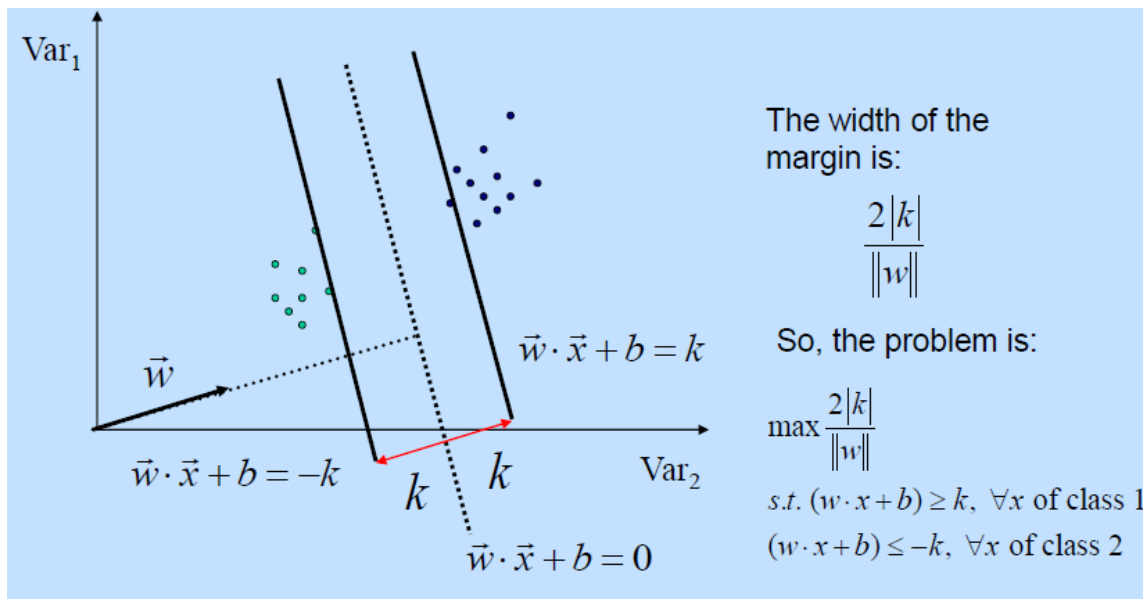


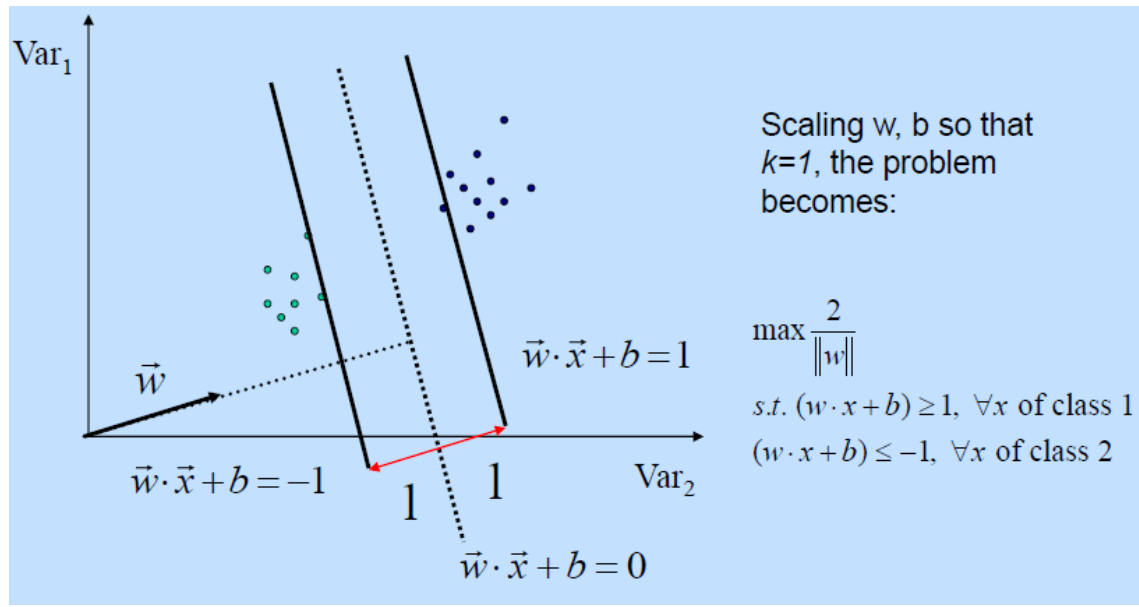
Maximize the margin



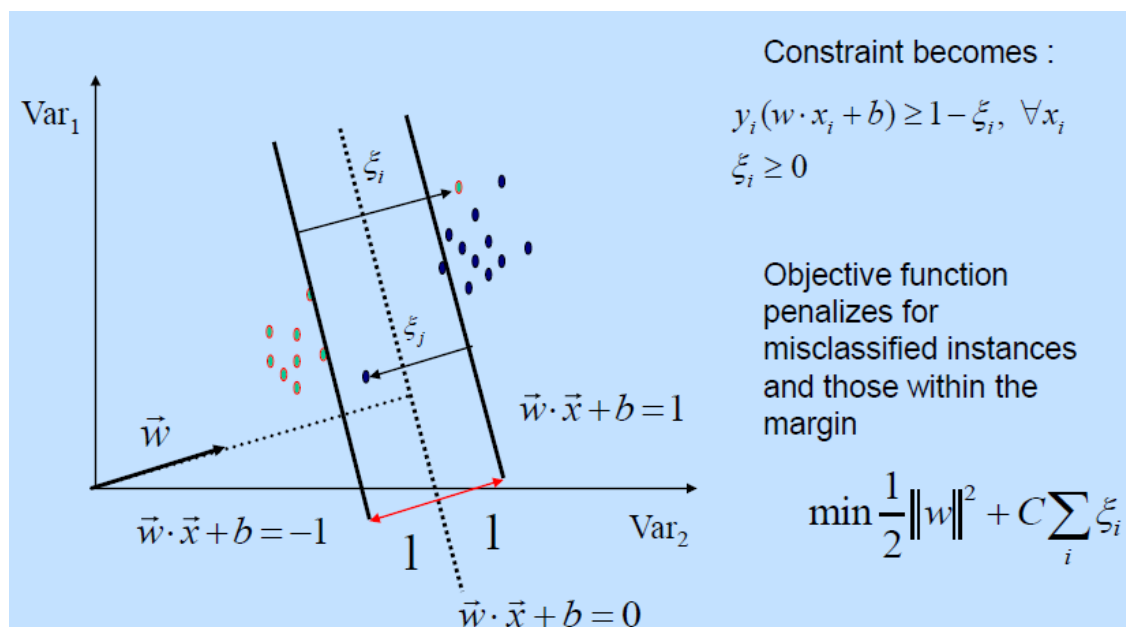
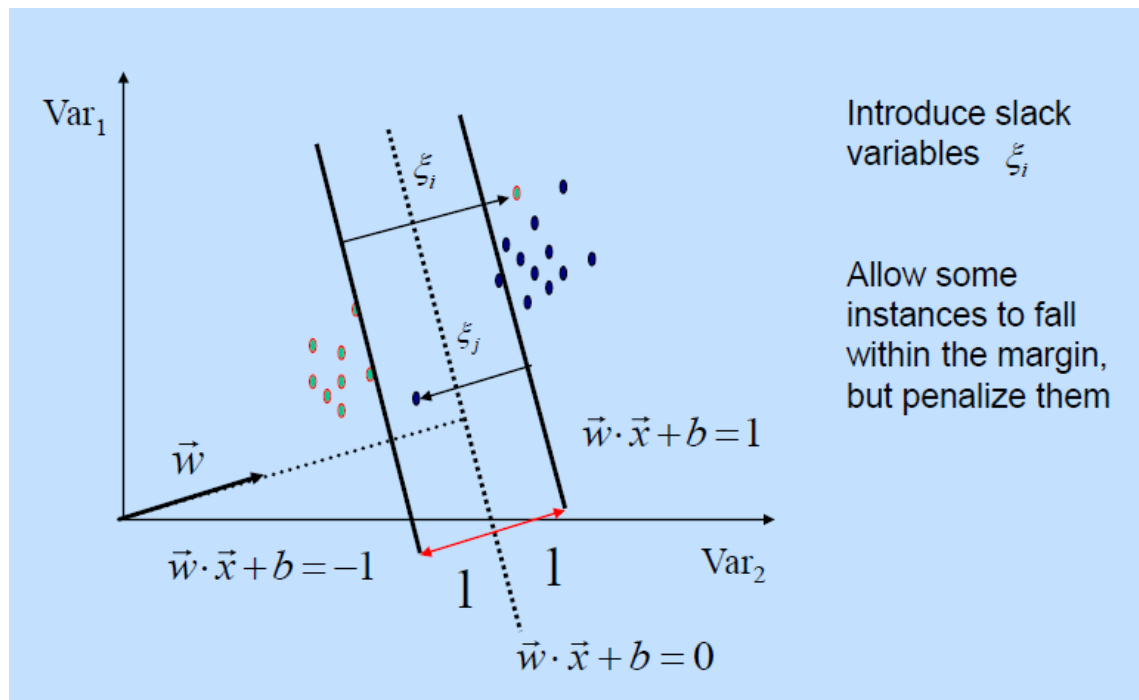


The optimization problem of SVM is ,





For data which is not linearly separable, we need to allow misclassifications or introduce slack variables.



C is a trade-off between margin-width and misclassification.

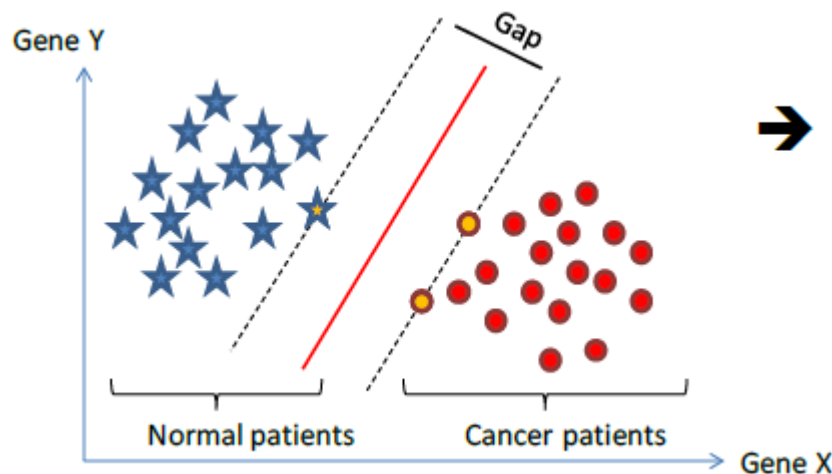
Hence,

- SVMs formulate learning as a mathematical program taking advantage of the rich theory in optimization

- SVM uses kernels to map indirectly to extremely high dimensional spaces
- SVMs are extremely successful, robust, efficient, and versatile, and have a good theoretical basis

In other words, SVM is all about,
if we know

- How to represent patients (as “vectors”)
- How to define a linear decision surface (“hyperplane”)
- We can find
- How to efficiently compute the hyperplane that separates two classes with the largest “gap”? Using the SVM



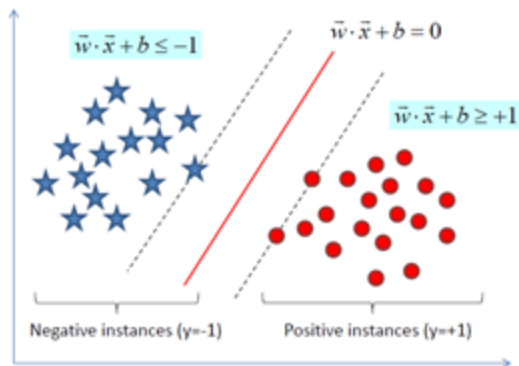
- Want to find a classifier (hyperplane) to separate negative instances from the positive ones.
- An infinite number of such hyperplanes exist
- SVMs finds the hyperplane that maximizes the gap between data points on the boundaries (so-called “support vectors”).
- If the points on the boundaries are not informative (e.g., due to noise), SVMs will not do well.

SVM-Linearly separable

- The representation of the correct classifications are :

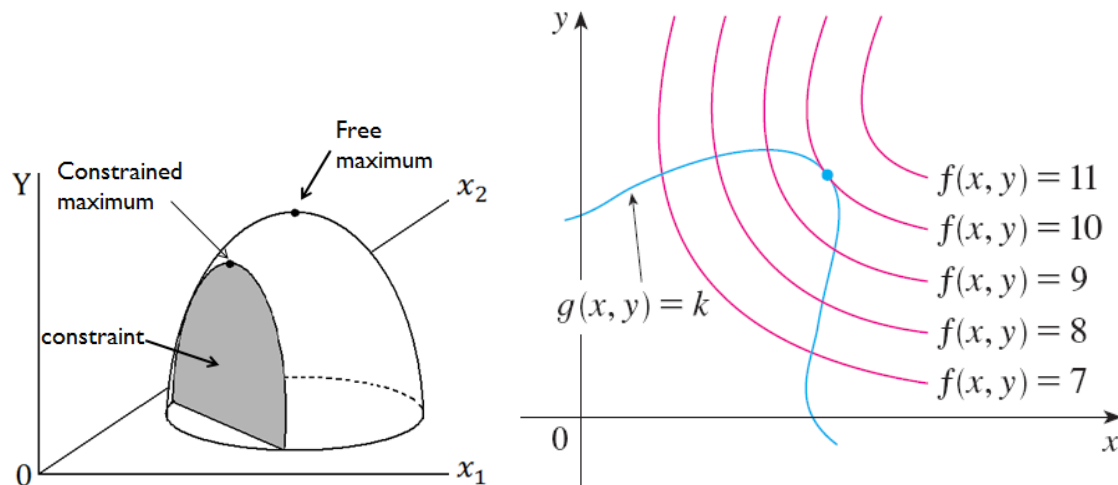
$$\bar{w} \cdot \bar{x} + b \leq -1 \text{ if } y_1 = -1 \text{ and}$$

$$\bar{w} \cdot \bar{x} + b \geq 1 \text{ if } y_1 = 1$$



- Want to minimize $\frac{1}{2} \|\bar{w}\|^2$ subject to $y_i(\bar{w} \cdot \bar{x} + b) \geq 1$ for $i = 1, \dots, N$
- Then given a new instance x , the classifier is $f(\bar{x}) = \text{sign}(\bar{w} \cdot \bar{x} + b)$

- We'll employ Lagrange Multiplier method to minimize.



- In other words, we seek the extreme values of $f(x, y)$ when the point (x, y) is restricted to lie on the level curve $g(x, y) = k$.

- Procedure:
- Step1: Form the Lagrangian (augmented function) $L = f(x, y) + \lambda g(x, y)$
- Step2: Obtain the partial derivatives $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial \lambda}$
- Step3: Equate the derivatives to zero, solve them simultaneously to get the solution.

Lagrange Multiplier method

- Example: Opt $y = 5x_1x_2$ subject to $2x_1 + x_2 = 100$
- Solution: Lagrangian $y_\lambda = 5x_1x_2 + \lambda(100 - 2x_1 - x_2)$

$$\left. \begin{aligned} \frac{\partial y_\lambda}{\partial x_1} &= 5x_2 - 2\lambda = 0 \\ \frac{\partial y_\lambda}{\partial x_2} &= 5x_1 - \lambda = 0 \\ \frac{\partial y_\lambda}{\partial \lambda} &= 100 - 2x_1 - x_2 = 0 \end{aligned} \right\} \begin{array}{l} 3 \text{ unknowns:} \\ x_1, x_2, \lambda \\ 3 \text{ equations} \end{array}$$

$$\therefore x_2 = \frac{2\lambda}{5}$$

$$x_1 = \frac{\lambda}{5}$$

$$\therefore 100 - 2\left(\frac{\lambda}{5}\right) - \left(\frac{2\lambda}{5}\right) = 0$$

$$\therefore \lambda = 125$$

$$x_1 = \frac{\lambda}{5} = \frac{125}{5} = 25$$

$$x_2 = \frac{2\lambda}{5} = \frac{2(125)}{5} = 50$$

- The Lagrangian for SVM is formed $L = \frac{1}{2}\|w\|^2 - \sum \alpha_i (y_i(\bar{w} \bullet \bar{x} + b) - 1)$
- The partial derivatives are equated to zero to get the solution as
- $L = \sum \alpha_i - \frac{1}{2} \sum \sum \alpha_i \alpha_j y_i y_j \bar{x}_i \bullet \bar{x}_j$

SVM-Not linearly separable:

SVM

- When data is linearly separable



- We can use support vectors on each class, build a classifier using the margin.
- When the data is not linearly separable



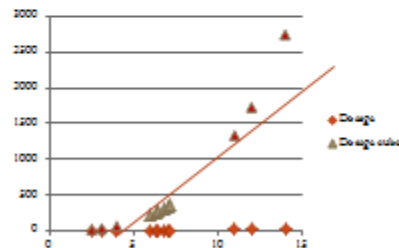
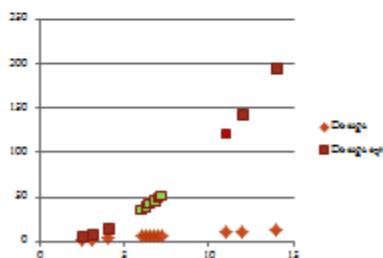
- Then, the idea is transform to a higher dimension and search for a hyperplane –support vector classifier

SVM

- When the data is not linearly separable



- We can use polynomial kernel like $f(x)=x^4$, $f(x)=x^3$.



SVM-Polynomial Kernel

- Polynomial kernel is defined by $(a*b+r)^d$
- Where r is the coefficient of the polynomial and d is the degree of the polynomial and these two are the parameters.
- If $r=1/2$ and $d=2$

$$(a*b+1/2)^2 = (a*b+1/2)(a*b+1/2) = (a, a^2, 1/2) \bullet (a, a^2, 1/2)$$

- If $r=1$ and $d=2$

$$(a*b+1)^2 = (a*b+1)(a*b+1) = (\sqrt{2}a, a^2, 1) \bullet (\sqrt{2}a, a^2, 1)$$

SVM-Polynomial Kernel

- Polynomial kernel is defined by $(a*b+r)^d$
- So, Kernel=dot product.
- If $a=9$, $b=14$, $r=1/2$, $d=2$, distance between a and b in 2 dimension is

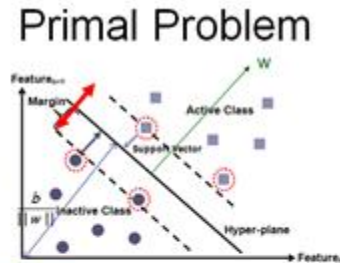
$$(9*14+1/2)^2 = 16.002$$
- So, without transforming them onto a higher (2) dimension, we are getting the distance between them in that dimension. This is called the Kernel Trick.

- RBF kernel is defined by
- Where γ is the scale.
- If $\gamma=1$, $a=2.5$, $b=4$, $e^{-\gamma(a-b)^2} = e^{-(2.5-4)^2} = 0.11$
- If $\gamma=1$, $a=2.5$, $b=16$, $e^{-\gamma(a-b)^2} = e^{-(2.5-16)^2} \approx 0$
- So, distance is more implies the amount of influence is small.

- RBF also transforms data to a higher dimension but this is to infinite dimension.

SVM-problems

- What is the dual involved in the SVM.
- Original SVM optimization problem is



Minimize $\frac{1}{2} \sum_{i=1}^n w_i^2$ subject to $y_i(\bar{w} \cdot \bar{x}_i + b) - 1 \geq 0$ for $i = 1, \dots, N$

Objective function Constraints

- Its Dual is

Maximize $\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \bar{x}_i \cdot \bar{x}_j$ subject to $\alpha_i \geq 0$ and $\sum_{i=1}^N \alpha_i y_i = 0$.

Objective function Constraints

SVM problems

- What is a soft Margin SVM?
- SVM that allows misclassifications. Indicated by C.

Primal formulation:

Minimize $\frac{1}{2} \sum_{i=1}^n w_i^2 + C \sum_{i=1}^N \xi_i$ subject to $y_i(\bar{w} \cdot \bar{x}_i + b) \geq 1 - \xi_i$ for $i = 1, \dots, N$

Objective function Constraints

Dual formulation:

Minimize $\sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \bar{x}_i \cdot \bar{x}_j$ subject to $0 \leq \alpha_i \leq C$ and $\sum_{i=1}^N \alpha_i y_i = 0$

Objective function Constraints

for $i = 1, \dots, N$.

- For large C, this is equivalent to Hard Margin SVM.

$$K(\vec{x}_i, \vec{x}_j) = \Phi(\vec{x}_i) \cdot \Phi(\vec{x}_j)$$

Examples:

$$K(\vec{x}_i, \vec{x}_j) = \vec{x}_i \cdot \vec{x}_j$$

Linear kernel

$$K(\vec{x}_i, \vec{x}_j) = \exp(-\gamma \|\vec{x}_i - \vec{x}_j\|^2)$$

Gaussian kernel

$$K(\vec{x}_i, \vec{x}_j) = \exp(-\gamma \|\vec{x}_i - \vec{x}_j\|)$$

Exponential kernel

$$K(\vec{x}_i, \vec{x}_j) = (p + \vec{x}_i \cdot \vec{x}_j)^q$$

Polynomial kernel

$$K(\vec{x}_i, \vec{x}_j) = (p + \vec{x}_i \cdot \vec{x}_j)^q \exp(-\gamma \|\vec{x}_i - \vec{x}_j\|^2)$$

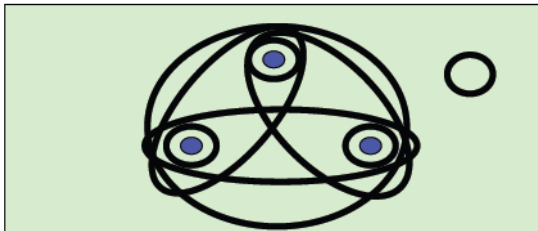
Hybrid kernel

$$K(\vec{x}_i, \vec{x}_j) = \tanh(k\vec{x}_i \cdot \vec{x}_j - \delta)$$

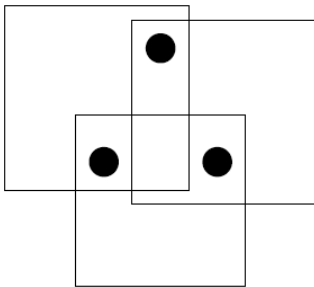
Sigmoidal

VC dim:

- What is VC dim?
- Def.1: (**set shattering**): a subset S of instances of a set
- X is shattered by a collection of function F if $\forall S' \subseteq S$
- there is a function $f \in F$ such data:
- $f(x) = 1 \quad x \in S'$
- $0 \quad x \in S - S'$.
- Definition: the VC-dimension of a function set F ($\text{VCdim}(F)$) is the cardinality of the largest dataset that can be shattered by F .

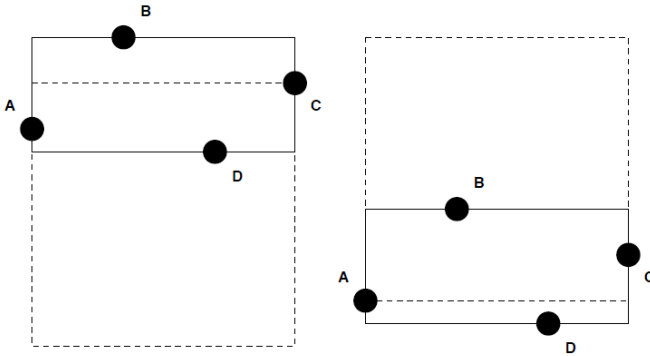


- VC dim of squares is 3.
- There exist 3 points that can be shattered.
- 1 point and 3 points are trivial. The figure below shows how we can capture 2 points.
- So, yes, there exists an arrangement of 3 points that can be shattered.



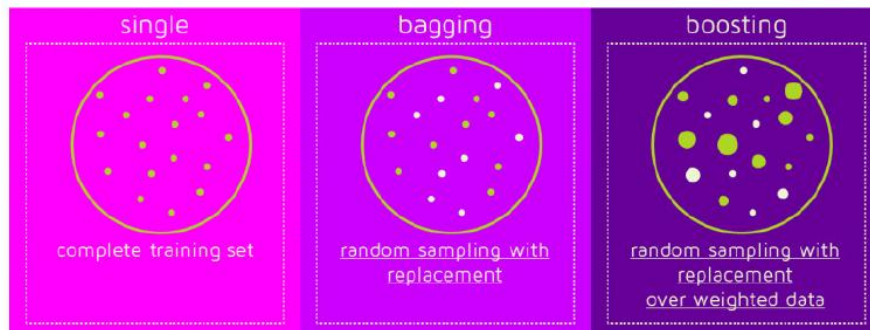
- VC dim of squares is 3.

- No set of 4 points can be shattered.
- Suppose we have four points arranged such that they define a rectangle. Now, suppose we want to select two points (A&C, in this case).
- The minimum enclosing square for A&C must contain either B or D – so we can't capture just two points with a square.



Ensemble Methods:

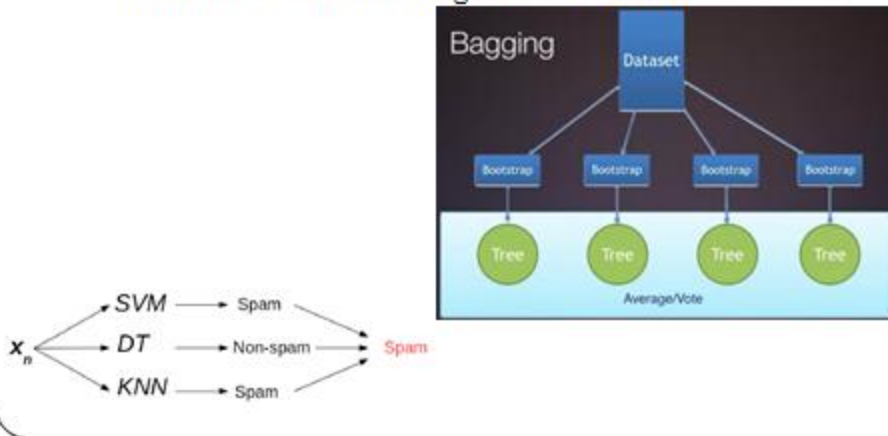
- Combines works of many Experts.



Bagging:

Bagging

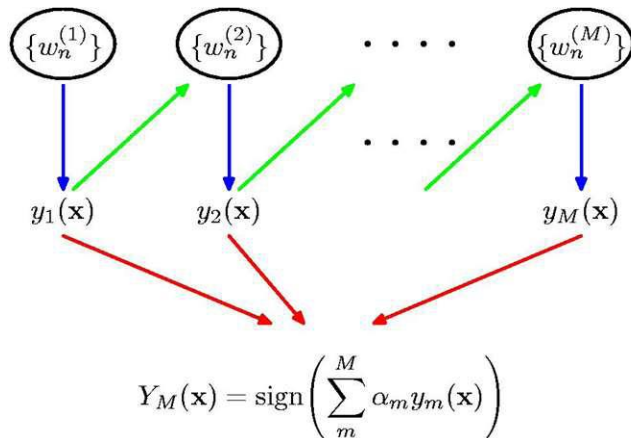
- Bootstrap Aggregation
- Uses many / different models with different samples(Bootstrap) and uses the method of Voting to make decisions.



- The error rate for a particular learning algorithm is called its **bias** for the learning problem and measures how well the learning method matches the problem.
- A second source of error in a learned model, in a practical situation, stems from the particular training set used, which is inevitably finite and therefore not fully representative of the actual population of instances. The expected value of this component of the error, over all possible training sets of the given size and all possible test sets, is called the **variance** of the learning method for that problem
- Combining multiple classifiers decreases the expected error by reducing the variance component.
- Takes original data set D with N training examples
- Creates M copies $[D_m]$, $m=1,2,...M$
- Each D_m is generated from D by sampling with replacement
- Each data set D_m has the same number of examples as in data set D
- These data sets are reasonably different from each other (since only about 63% of the original examples appear in any of these data sets)
- Train models $h_1, h_2, ..., h_M$ using $D_1, D_2, ..., D_M$, respectively
- Use an averaged model/Voting to make the final decision.
- Useful for models with high variance and noisy data

Boosting:

- The boosting method seeks models that complement one another just like Bagging but it takes the route of minimizing the errors by focusing on the errors themselves more along with the models.
- It is iterative.
- Each model depends on the previous unlike Bagging where models are independently built.



- Boosting weighs the instances according to their error factor.
- By weighting instances, the learning algorithm can be forced to concentrate on a particular set of instances, namely, those with high weight.
- How much should the weights be altered after each iteration? The answer depends on the current classifier's overall error. More specifically, if e denotes the classifier's error on the weighted data (a fraction between 0 and 1), then weights are updated by
 - new weight of instance = weight * $e / (1 - e)$
- for correctly classified instances.
- Whenever the error on the weighted training data exceeds or equals 0.5, the boosting procedure deletes the current classifier and does not perform any more iterations. The same thing happens when the error is 0, because then all instance weights become 0.
- To form a prediction, their output is combined using a weighted vote. To determine the weights, note that a classifier that performs well on the weighted training data from which it was built (e close to 0) should receive a high weight, and a classifier that performs badly (e close to 0.5) should receive a low one. More specifically,
 - Weight of classifier = $-\log[e/(1-e)]$
- which is a positive number between 0 and infinity.
- This formula explains why classifiers that perform perfectly on the training data must be deleted, because when e is 0 the weight is undefined. To make a prediction, the weights

of all classifiers that vote for a particular class are summed, and the class with the greatest total is chosen.

AdaBoost:

- Start with a supervised training data $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ where $y_n \in [-1, 1]$, as we are in Binary classification .
- Initialize weight of each example
- $(x_n, y_n): D_1(n) = 1/N$ for all n

AdaBoost

• For $t=1:T$

- Learn a weak $h_t(x) \rightarrow \{-1, +1\}$ using training data **weighted as per D_t**
- Compute the **weighted** fraction of errors of h_t on this training data

$$\epsilon_t = \sum_{n=1}^N D_t(n) \mathbb{I}[h_t(x_n) \neq y_n]$$

- Set "importance" of h_t : $\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ (gets larger as ϵ_t gets smaller)
- **Update the weight** of each example

$$\begin{aligned} D_{t+1}(n) &\propto \begin{cases} D_t(n) \times \exp(-\alpha_t) & \text{if } h_t(x_n) = y_n \quad (\text{correct prediction: decrease weight}) \\ D_t(n) \times \exp(\alpha_t) & \text{if } h_t(x_n) \neq y_n \quad (\text{incorrect prediction: increase weight}) \end{cases} \\ &= D_t(n) \exp(-\alpha_t y_n h_t(x_n)) \end{aligned}$$

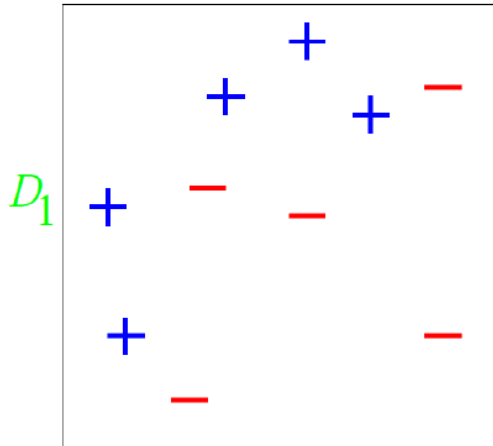
- Normalize D_{t+1} so that it sums to 1: $D_{t+1}(n) = \frac{D_t(n) \exp(-\alpha_t y_n h_t(x_n))}{\sum_{n=1}^N D_t(n) \exp(-\alpha_t y_n h_t(x_n))}$

$$H(x) = \text{sign}\left(\sum_{t=1}^T \alpha_t h_t(x)\right)$$

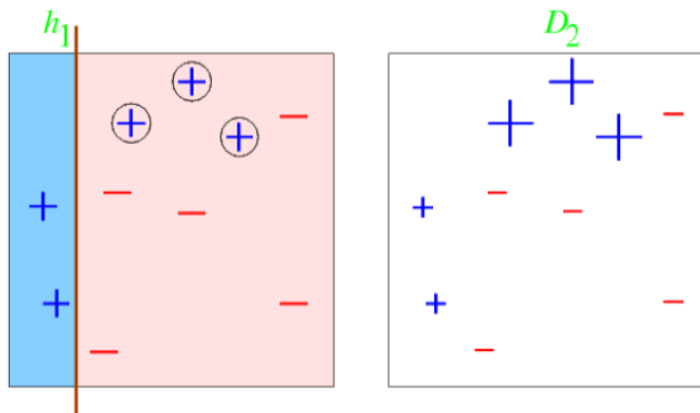
Example:

- Consider binary classification with 10 training examples.
- Initial weight distribution D_1 is uniform (each point has equal weight = $1/10$)
- In this case, all classifiers

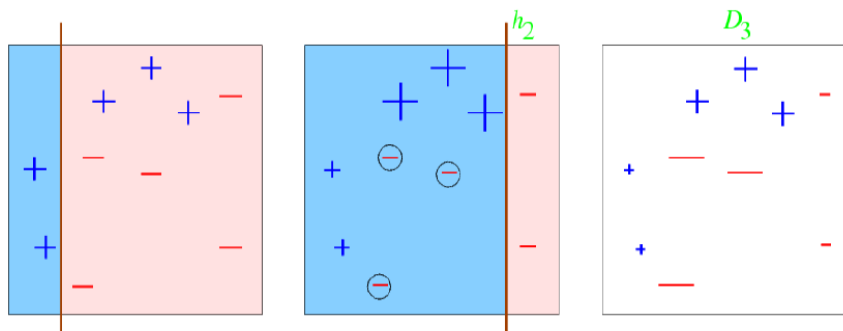
which are parallel to the axes can be the weak classifiers to start with.



-
- After iteration 1,

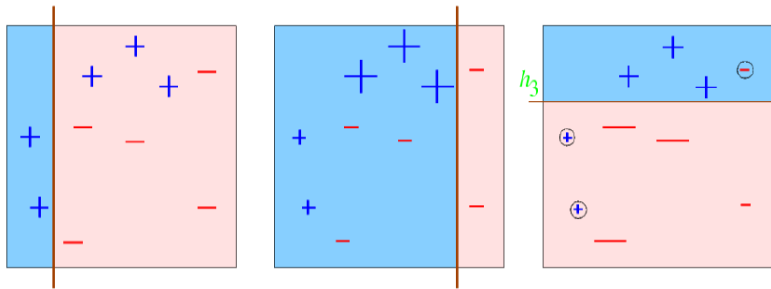


-
- Error rate of h_1 : $\epsilon_1 = [0.1+0.1+0.1]/1=0.3$; weight of
- h_1 : $\alpha_1=1/2 \ln[(1- \epsilon_1)/ \epsilon_1] = 0.42$
- Each misclassified point's weight is increased (weight multiplied by $\exp(\alpha_1)$)
- Each correctly classified point's weight is decreased (weight multiplied by $\exp(-\alpha_1)$)
- After iteration 2,



-
- Error rate of h_2 : $\epsilon_2 = 0.21$;
- weight of h_2 : $\alpha_2=1/2 \ln[(1- \epsilon_2)/ \epsilon_2] = 0.65$
- Each misclassified point's weight is increased (weight multiplied by $\exp(\alpha_2)$)
- Each correctly classified point's weight is decreased (weight multiplied by $\exp(-\alpha_2)$)

- After iteration 3,



-
- Error rate of h_3 : $\epsilon_3 = 0.14$;
- weight of h_3 : $\alpha_3 = 1/2 \ln[(1 - \epsilon_3)/\epsilon_3] = 0.92$
- Suppose we decide to stop after round 3
- Our ensemble now consists of 3 classifiers: h_1, h_2, h_3
- Final Classifier is a Weighted Linear Combination of
- All three, h_i with weight α_i ,

$$H_{\text{final}} = \text{sign} \left(0.42 \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \\ \hline \end{array} + 0.65 \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \\ \hline \end{array} + 0.92 \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \\ \hline \end{array} \right)$$

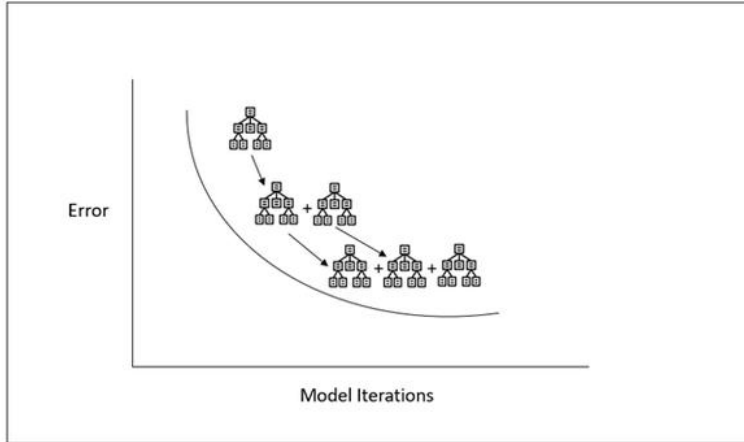
=

-
- Multiple weak, linear classifiers combined to give a strong, nonlinear classifier

Gradient Boosting:

- Recall, adaptive boosting **changes the sample distribution** by modifying the weights attached to each of the instances. It increases the weights of the wrongly predicted instances and decreases the ones of the correctly predicted instances.
- On the other hand, gradient boosting doesn't modify the sample distribution. Instead of training on a newly sample distribution, the weak learner **trains on the remaining errors** (so-called pseudo-residuals) of the strong learner, just like in the case of Gradient descent.
- GB has three major ideas.
- Loss function: Is different for different problems, measuring fit of the model, but we always try and minimize it. For regression we may use a squared error and classification may use logarithmic loss.

- Weak Learner: In GB these are generally Decision stumps(DT with single split). Also called Shallow trees.
- Adding classifiers: Generally the Decision stumps are added sequentially, one at a time in such a way that it reduces the loss. This is where we use the Gradient descent idea.



- Given a training data $\{\mathbf{x}_i, y_i\}$, $i=1,2,\dots,N$, we need to find a model $F(\mathbf{x})$ that map \mathbf{x} to y . We will measure the mapping effectiveness by choosing a loss function $L(y, F(\mathbf{x}))$ and then minimize it: $F^* = \arg \min_F E_{y,x} L(y, F(x))$

- The model $F(\mathbf{x})$ is constructed in the form: $F(x) = F_0(x) + \sum_{m=1}^M \rho_m h_m(x)$

- This is called boosting where:
- $F_0(\mathbf{x})$ is a initial guess, the first starting point of the model.
- $h_m(\mathbf{x})$ is called “weak learner” or “base learner”.

- We start the process by choosing $F_0(\mathbf{x})$, then measure the error between target y and prediction $F_0(\mathbf{x})$: $y - F_0(\mathbf{x})$. Next, we try to find a new function $h_0(\mathbf{x})$ so that:
- $F_0(\mathbf{x}) + \rho_0 h_0(\mathbf{x}) = y$
- The role of h_0 is to reduce the error of predictive model $F(\mathbf{x})$. This step is proceeded recursively, we can add up another h_1, \dots, h_m until the best F^* is found. Each new h_m will try to correct errors made by previous h_{m-1} so that:
- $F_m(\mathbf{x}) + \rho_m h_m(\mathbf{x}) = y$.
- We use gradient descent to guide h

- If we are using squared loss function, then $\frac{\partial (y_i - F(x)_i)^2}{\partial F(x)_i} = -2 * y_i - F(x)_i = -2 * r_i$

- We want to fit $h(x)$ (shallow trees) such that it reaches close to the actual model:
- $F(x_1) + h(x_1) = y_1$

- $F(x^2) + h(x^2) = y^2 \dots\dots$
- We find the γ which minimizes the loss

Bagging Vs Boosting:

- Bagging is computationally more efficient than boosting (note that bagging can train the M models in parallel, boosting can't).
- Both reduce variance (and over-fitting) by combining different models
 - The resulting model has higher stability as compared to the individual ones
- Bagging usually can't reduce the bias, boosting can (note that in boosting, the training error steadily decreases).
- Bagging usually performs better than boosting if we don't have a high bias and only want to reduce variance (i.e., if we are over-fitting)

Unsupervised Learning:

k-Means Clustering:

- *Distance measure* will determine how the *similarity* of two elements is calculated and it will influence the shape of the clusters.

They include:

The [Euclidean distance](#) (also called 2-norm distance) is given by: $d(x, y) = \sqrt{\sum_{i=1}^P (x_i - y_i)^2}$

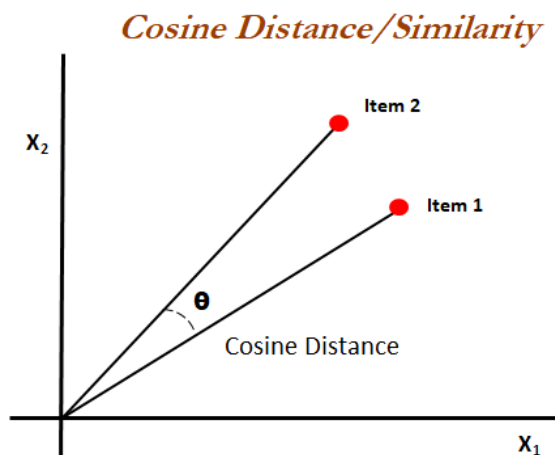
The [Manhattan distance](#) (also called taxicab norm or 1-norm) is given by:

$$d(x, y) = \sqrt{\sum_{i=1}^P |x_i - y_i|}$$

3. The [maximum norm](#) is given by: $d(x, y) = \max_{1 \leq i \leq p} |x_i - y_i|$

Cosine similarity is a measure of **similarity** between two non-zero vectors of an inner product space that measures the **cosine** of the angle between them. The **cosine** of 0° is 1, and it is less than 1 for any angle in the interval $(0, \pi]$ radians

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}},$$



Any Distance measure used should have the following properties:

- Symmetric
 - $D(A,B)=D(B,A)$
 - Otherwise, we can say A looks like B but B does not look like A
- Positivity, and self-similarity
 - $D(A,B) \geq 0$, and $D(A,B)=0$ iff $A=B$
 - Otherwise there will different objects that we cannot tell apart
- Triangle inequality
 - $D(A,B)+D(B,C) \geq D(A,C)$
 - Otherwise one can say “A is like B, B is like C, but A is not like C at all”

K-Means Clustering

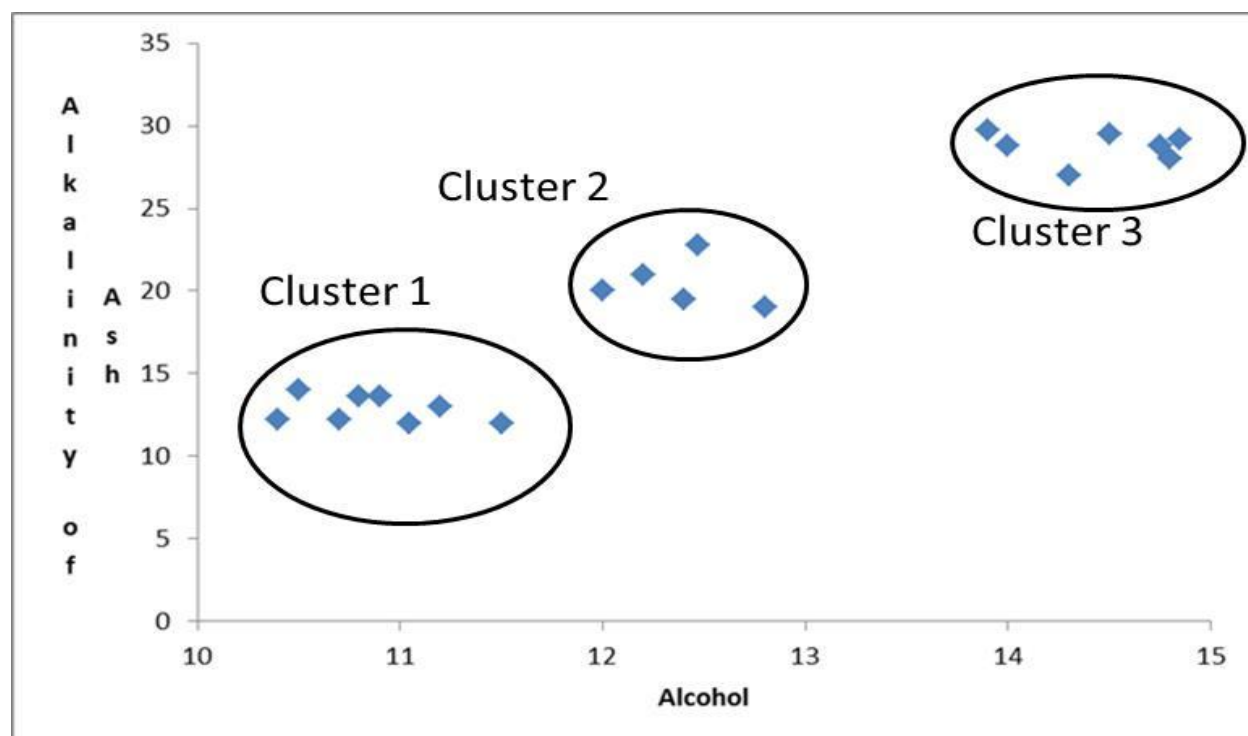
- Given k , the *k-means* algorithm consists of four steps:
 - Select initial centroids at random.
 - Assign each object to the cluster with the nearest centroid.
 - Compute each centroid as the mean of the objects assigned to it.
 - Repeat previous 2 steps until no change
- Strengths
 - *Relatively efficient: $O(tkn)$* , where n is # objects, k is # clusters, and t is # iterations. Normally, $k, t \ll n$.
 - Often terminates at a *local optimum*. The *global optimum* may be found using techniques such as *simulated annealing* and *genetic algorithms*
- Weaknesses
 - Applicable only when *mean* is defined (what about categorical data?)
 - Need to specify k , the *number* of clusters, in advance
 - Trouble with noisy data and *outliers*
 - Not suitable to discover clusters with *non-convex shapes*

The **k-means algorithm** is an algorithm to cluster n objects based on attributes into k partitions, where $k < n$.

- It is similar to the expectation-maximization algorithm for mixtures of Gaussians in that they both attempt to find the centers of natural clusters in the data.
- It assumes that the object attributes form a vector space.

Example: The below table has information about 20 wines sold in the market along with their alcohol and alkalinity of ash content

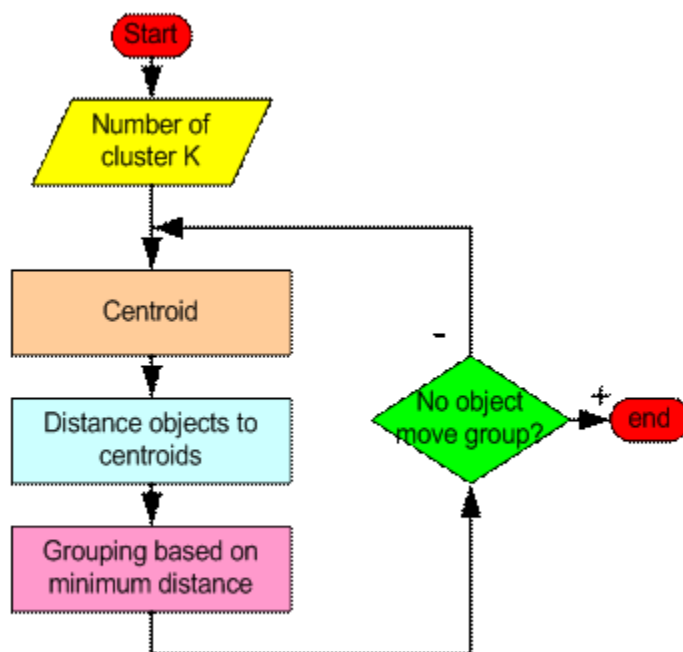
Wine	Alcohol	Alkalinity of Ash	Wine	Alcohol	Alkalinity of Ash
1	14.8	28	11	10.7	12.2
2	11.05	12	12	14.3	27
3	12.2	21	13	12.4	19.5
4	12	20	14	14.85	29.2
5	14.5	29.5	15	10.9	13.6
6	11.2	13	16	13.9	29.7
7	11.5	12	17	10.4	12.2
8	12.8	19	18	10.8	13.6
9	14.75	28.8	19	14	28.8
10	10.5	14	20	12.47	22.8



k-Means Algorithm

- An algorithm for partitioning (or clustering) N data points into K disjoint subsets S_j containing data points so as to minimize the sum-of-squares criterion
- where x_n is a vector representing the n^{th} data point and u_j is the [geometric centroid](#) of the data points in S_j .
- The grouping is done by minimizing the sum of squares of distances between data and the corresponding cluster centroid.

$$J = \sum_{j=1}^K \sum_{n \in S_j} |x_n - \mu_j|^2,$$



- **Step 1:** Begin with a decision on the value of $k =$ number of clusters .
- **Step 2:** Put any initial partition that classifies the data into k clusters. You may assign the training samples randomly, or systematically as the following:
 1. Take the first k training sample as single-element clusters

2. Assign each of the remaining (N-k) training sample to the cluster with the nearest centroid. After each assignment, recompute the centroid of the gaining cluster.

- **Step 3:** Take each sample in sequence and compute its [distance](#) from the centroid of each of the clusters. If a sample is not currently in the cluster with the closest centroid, switch this sample to that cluster and update the centroid of the cluster gaining the new sample and the cluster losing the sample.
- **Step 4 .** Repeat step 3 until convergence is achieved, that is until a pass through the training sample causes no new assignments.

Example:

Individual	Variable 1	Variable 2
1	1.0	1.0
2	1.5	2.0
3	3.0	4.0
4	5.0	7.0
5	3.5	5.0
6	4.5	5.0
7	3.5	4.5

Step 1:

Initialization: Randomly we choose following two centroids (k=2) for two clusters.

In this case the 2 centroid are: $m1=(1.0,1.0)$ and $m2=(5.0,7.0)$.

Individual	Variable 1	Variable 2
1	1.0	1.0
2	1.5	2.0
3	3.0	4.0
4	5.0	7.0
5	3.5	5.0
6	4.5	5.0
7	3.5	4.5

	Individual	Mean Vector
Group 1	1	(1.0, 1.0)
Group 2	4	(5.0, 7.0)

Step 2:

- Thus, we obtain two clusters containing:

{1,2,3} and {4,5,6,7}.

- Their new centroids are:

$$m_1 = \left(\frac{1}{3}(1.0 + 1.5 + 3.0), \frac{1}{3}(1.0 + 2.0 + 4.0) \right) = (1.83, 2.33)$$

$$m_2 = \left(\frac{1}{4}(5.0 + 3.5 + 4.5 + 3.5), \frac{1}{4}(7.0 + 5.0 + 5.0 + 4.5) \right) = (4.12, 5.38)$$

Individual	Centroid 1	Centroid 2
1	0	7.21
2 (1.5, 2.0)	1.12	6.10
3	3.61	3.61
4	7.21	0
5	4.72	2.5
6	5.31	2.06
7	4.30	2.92

$$d(m_1, 2) = \sqrt{|1.0 - 1.5|^2 + |1.0 - 2.0|^2} = 1.12$$

$$d(m_2, 2) = \sqrt{|5.0 - 1.5|^2 + |7.0 - 2.0|^2} = 6.10$$

Step 3:

- Now using these centroids we compute the Euclidean distance of each object, as shown in table.
- Therefore, the new clusters are:
 $\{1, 2\}$ and $\{3, 4, 5, 6, 7\}$
- Next centroids are: $m_1 = (1.25, 1.5)$ and $m_2 = (3.9, 5.1)$

Individual	Centroid 1	Centroid 2
1	1.57	5.38
2	0.47	4.28
3	2.04	1.78
4	5.64	1.84
5	3.15	0.73
6	3.78	0.54
7	2.74	1.08

- Step 4:

The clusters obtained are:

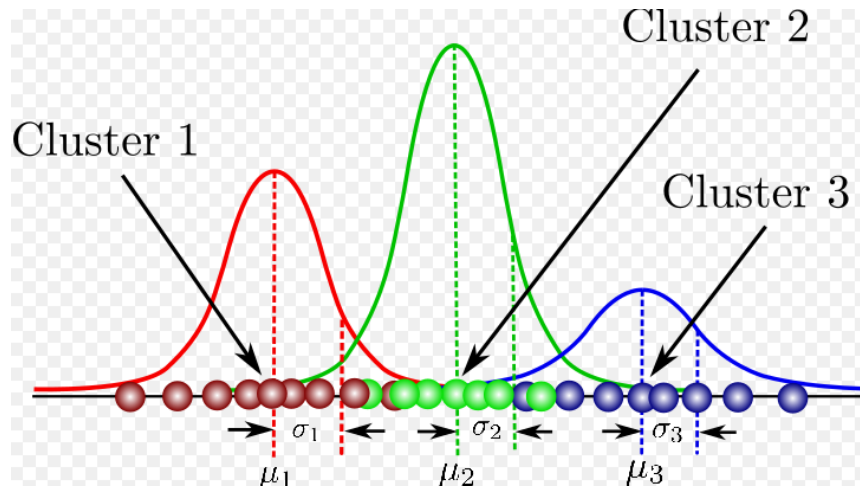
{1,2} and {3,4,5,6,7}

- Therefore, there is no change in the cluster.
- Thus, the algorithm comes to a halt here and final result consist of 2 clusters {1,2} and {3,4,5,6,7}.

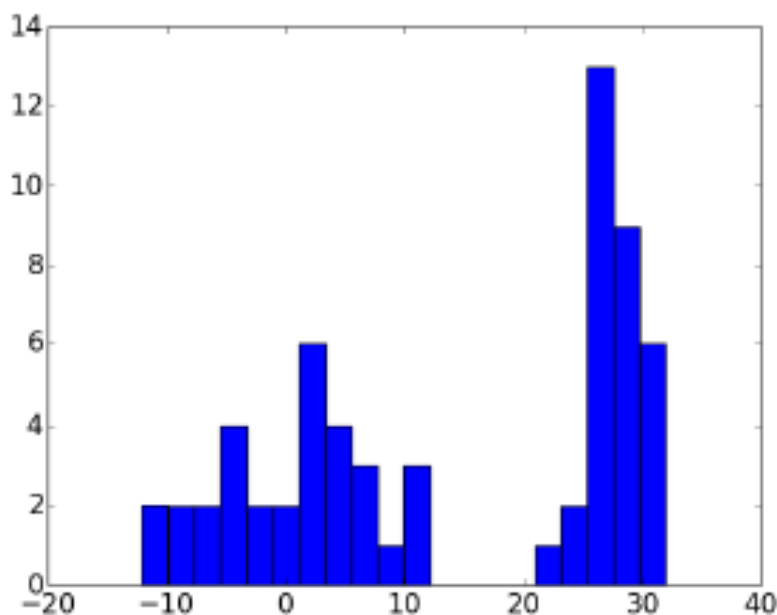
Individual	Centroid 1	Centroid 2
1	0.56	5.02
2	0.56	3.92
3	3.05	1.42
4	6.66	2.20
5	4.16	0.41
6	4.78	0.61
7	3.75	0.72

Mixture models:

- MM is a probabilistic/Statistic model which assumes that the data in analysis comes from more than one population.
- Prevalent MM is a Gaussian MM or GMM



Here we model the data in terms of a mixture of several components, where each component has a simple parametric form (such as a Gaussian). In other words, we assume each data point belongs to one of the components, and we try to infer the distribution for each component separately. For example consider the example of , Temperatures of two cities. We happen to know that the two mixture components should correspond to the two cities. The model itself doesn't know" anything about cities, though: this is just something we would read into it at the very end, when we analyze the results. In general, we won't always know precisely what meaning should be attached to the latent variables.



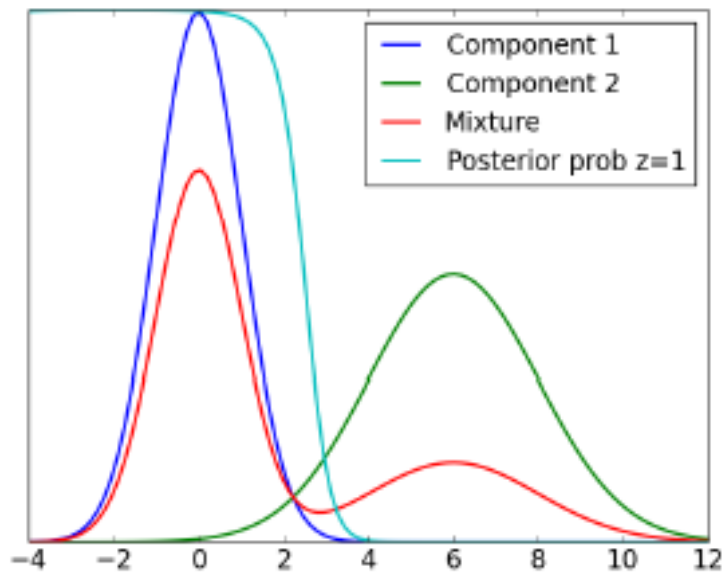
In order to represent this mathematically, we formulate the model in terms of latent variables, usually denoted z . These are variables which are never observed, and where we don't know the correct values in advance. They are roughly analogous to hidden units, in that the learning algorithm needs to find out what they should represent, without a human specifying it by hand. Variables which are always observed, or even sometimes observed, are referred to as observables. In the above example, the city is the latent variable and the temperature is the observable.

In mixture models, the latent variable corresponds to the mixture component. It takes values in a discrete set, which we'll denote $\{1, \dots, K\}$. (For now, assume K is fixed; we'll talk later about how to choose it.) In general, a mixture model assumes the data are generated by the following process: first we sample z , and then we sample the observables x from a distribution which depends on z , i.e.

$$p(z; x) = p(z) p(x | z):$$

In mixture models, $p(z)$ is always a multinomial distribution. $p(x | z)$ can take a variety of parametric forms, but for this lecture we'll assume it's a Gaussian distribution. We refer to such a model as a mixture of Gaussians.

For example, consider



The above picture shows an example of a mixture of Gaussians model with 2 components. It has the following generative process:

- _ With probability 0.7, choose component 1, otherwise choose component 2
- _ If we chose component 1, then sample x from a Gaussian with mean 0 and standard deviation 1
- _ If we chose component 2, then sample x from a Gaussian with mean 6 and standard deviation 2

This can be written in a more compact mathematical notation:

$$z \approx \text{Multinomial}(0.7, 0.3) \quad (1)$$

$$x | z = 1 \approx \text{Gaussian}(0, 1) \quad (2)$$

$$x | z = 2 \approx \text{Gaussian}(6, 2) \quad (3)$$

For the general case,

$$z \approx \text{Multinomial}(\pi) \quad (4)$$

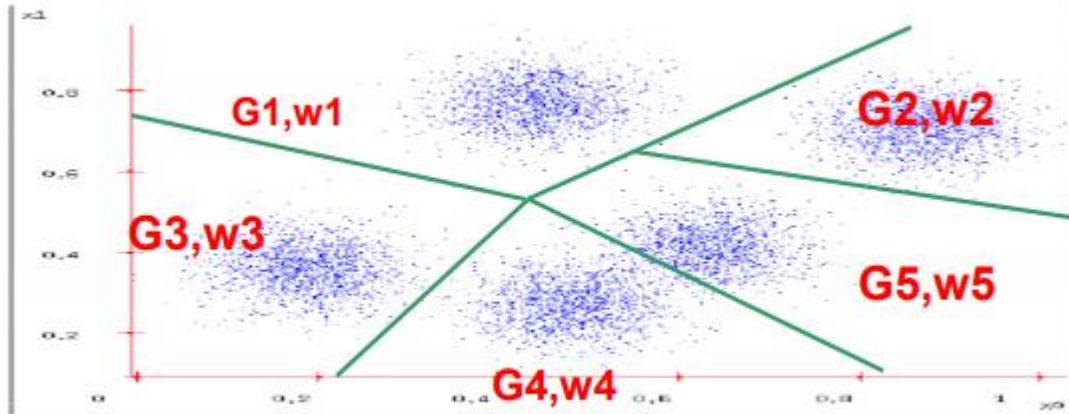
$$x | z = k \approx \text{Gaussian}(\mu_k; \sigma_k) \quad (5)$$

Here, π is a vector of probabilities (i.e. nonnegative values which sum to 1) known as the mixing proportions.

Mathematical Description of GMM

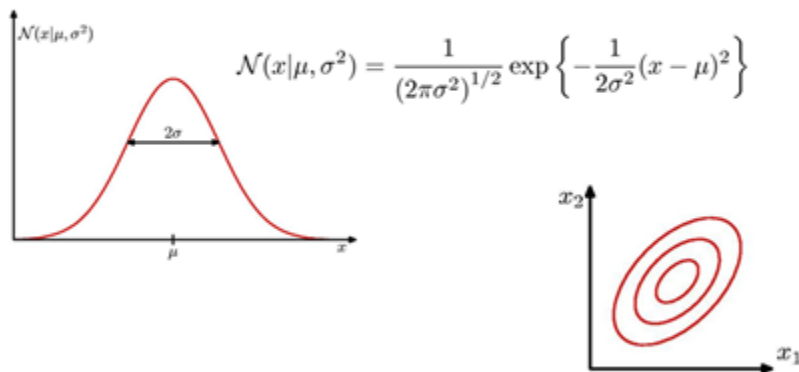
- $p(x) = w_1 p_1(x) + w_2 p_2(x) + w_3 p_3(x) + \dots + w_n p_n(x)$ where
- $p(x)$ = mixture component
- w_1, w_2, \dots, w_n = mixture weight or mixture coefficient
- $p_i(x)$ = Density functions

- The most common mixture distribution is the Gaussian (Normal) density function, in which each of the mixture components are Gaussian distributions, each with their own mean and variance parameters.”
- $p(x) = w_1 N(x | \mu_1 \Sigma_1) + w_2 N(x | \mu_2 \Sigma_2) \dots + w_n N(x | \mu_n \Sigma_n)$
- μ_i 's are means and Σ_i 's are covariance-matrix of individual components(probability density function)



GMM

- Gaussian models pdf



$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\Sigma|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right\}$$

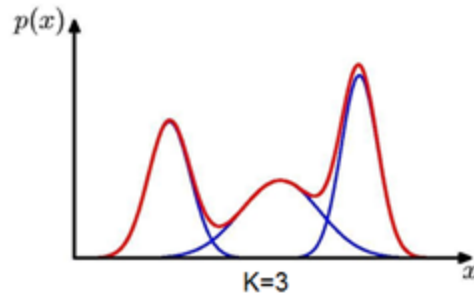
GMM

- Combine simple models into a complex model.

$$p(\mathbf{x}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

Component
Mixing coefficient

$$\forall k : \pi_k \geq 0 \quad \sum_{k=1}^K \pi_k = 1$$



- Find parameters through EM (Expectation Maximization) algorithm

EM algorithm:

- Clustering-Hard clusters-No overlap of clusters
- Mixture models- Soft clusters-Clusters overlap-Data points have prob/association with every cluster
- There is an underlying variable (latent-like z of normal)
- Given data we wish to identify the parameters of each component(Gaussian) and also the proportion of the mixture(π).
- EM finds all the parameters.

Expectation Maximization (EM) Algorithm

Example:

- _ Have two coins: Coin 1 and Coin 2
- _ Each has it's own probability of seeing "H" on any one ip. Let
 $p_1 = P(\text{H on Coin 1})$
 $p_2 = P(\text{H on Coin 2})$
- _ Select a coin at random and ip that one coin m times.
- _ Repeat this process n times.

_ Now the data is

11	X12	__	X1m	Y1
X21	X22	__	X2m	Y2
...				-
...				..
...				..
Xn1	Xn2	__	Xnm	Yn

Here, the X_{ij} are Bernoulli random variables taking values in $\{0, 1\}$ where

$$X_{ij} = \begin{cases} 1 & \text{if } j\text{th flip of the } i\text{th chosen coin is } H \\ 0 & \text{if } j\text{th flip of the } i\text{th chosen coin is } T \end{cases}$$

and the Y_i s in $\{1, 2\}$ and indicate which coin was used on the i th trial. Note that all the X 's are independent and, in particular

$$X_{i1}, X_{i2}, \dots, X_{im} | Y_i = j \stackrel{iid}{\sim} \text{Bernoulli}(p_j)$$

We can write out the joint pdf of all $nm + n$ random variables and formally come up with MLEs for p_1 and p_2 . Call these MLEs \hat{p}_1 and \hat{p}_2 . They will turn out as expected:

$$\hat{p}_1 = \frac{\text{Number of time coin 1 falls } H}{\text{Number of time coin 1 falls } T}; \quad \hat{p}_2 = \frac{\text{Number of time coin 2 falls } H}{\text{Number of time coin 2 falls } T}$$

Now suppose that the Y_i are not observed but we still want MLEs for p_1 and p_2 . The data set now consists of only the X 's and is "incomplete".

The goal of the EM Algorithm is to find MLEs for p_1 and p_2 in this case.

Expectation-maximization (EM) algorithm

An iterative algorithm for maximizing likelihood when the model contains unobserved latent variables.

The algorithm iterate between **E-step** (expectation) and **M-step** (maximization).

E-step: create a function for the expectation of the log-likelihood, evaluated using the current estimate for the parameters.

M-step: compute parameters maximizing the expected log-likelihood found on the E step.

Note: Generally the EM algorithm converges slowly.

Note: An expectation-maximization (EM) algorithm is used in statistics for finding maximum likelihood estimates of parameters in probabilistic models, where the model depends on unobserved hidden variables.

Note: EM alternates between performing an expectation (E) step, which computes an expectation of the likelihood by including the latent variables as if they were observed, and a maximization (M) step, which computes the maximum likelihood estimates of the parameters by maximizing the expected likelihood found on the E step. The parameters found on the M step are then used to begin another E step, and the process is repeated.

EM algorithm

Responsibility perspective

- Let us define a variable r_{nk} , as $r_{nk} \in \{0,1\}$ $\sum_k r_{nk} = 1$ for every data point x_n and cluster k , called responsibilities, the probability that x_n belongs to cluster k .
- If we have some number clusters, each with known mean, covariance and size(area) μ_c, Σ_c, π_c
- E step: Calculate responsibilities

$$r_{nk} = \frac{\pi_k N(x_n; \mu_k, \Sigma_k)}{\sum_k \pi_k N(x_n; \mu_k, \Sigma_k)}$$

EM algorithm

Responsibility perspective

- Let us define a variable r_{nk} , as $r_{nk} \in \{0,1\}$ $\sum_k r_{nk} = 1$ for every data point x_n and cluster k , called responsibilities.
- This is for association of the data point with the cluster.
- If data is 5 points and 3 clusters, we might have

		k means			EM		
		C1	C2	C3	C1	C2	C3
nk	x1	1	0	0	0.3	0.3	0.4
	x2	0	0	1	0.2	0.1	0.7
	x3	0	1	0	0.35	0.35	0.3
	x4	0	0	1	0	0.5	0.5
	x5	1	0	0	0.9	0.1	0

EM algorithm

Responsibility perspective

- M step: Maximization-Update the parameters of the clusters using the r_{nk} values.
- $m_k = \sum_n r_{nk}$ Total responsibility of cluster k
- $\pi_k = \frac{m_k}{m}$ Fraction of total assigned to cluster k
- $\mu_k = \frac{1}{m_k} \sum_n r_{nk} x_n$ Weighted mean of assigned data
- $\Sigma_k = \frac{1}{m_k} \sum_n r_{nk} (x_n - \mu_k)^T (x_n - \mu_k)$ Weighted covariance of assigned data

EM algorithm

Responsibility/Cost function perspective

- We can prove that the above procedure increase the log likelihood of the model
- To generalize from k-means to EM, we define a cost function as

$$J = \sum_{n=1}^N \sum_{k=1}^K r_{nk} \|x_n - \mu_k\|^2$$

- Where r_{nk} is the measure, x_n is the data and μ_k is the centroid/mean of the cluster.
- Then the EM algorithm could be viewed as
- E step: Minimize J w.r.t r_{nk} .
- M step: Minimize J w.r.t μ_k .