Clustering & Classification

Clustering vs classification

Clustering is partitioning an unlabelled dataset into groups of similar objects

Classification sorts data into specific categories using a labelled dataset

Clustering

From Wikipedia

> **Cluster analysis** *or* **clustering** *is the task of grouping a set of objects in such a way that objects in the same group (called a* **cluster***) are more similar (in some sense) to each other than to those in other groups (clusters).*

There are a myriad of ways to do clustering, this is an extremely active field of research and application. See the Wikipedia page for leads

We have done clustering already

We have seen some clustering:

- when we sought strongly connected components
- when we sought cliques
- to some extent, with PCA

Classification

From Wikipedia

> *In statistics,* **classification** *is the problem of identifying which of a set of categories (sub-populations) an observation (or observations) belongs to. Examples are assigning a given email to the "spam" or "non-spam" class, and assigning a diagnosis to a given patient based on observed characteristics of the patient (sex, blood pressure, presence or absence of certain symptoms, etc.).*

Support vector machines (SVM)

We are given a training dataset of n points of the form

$$(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$$

where $\mathbf{x}_i \in \mathbb{R}^p$ and $y_i = \{-1, 1\}$. The value of y_i indicates the class to which the point $\mathbf{x}_i$ belongs

We want to find a **surface** $\mathcal{S}$ in $\mathbb{R}^p$ that divides the group of points into two subgroups

Once we have this surface $\mathcal{S}$, any additional point that is added to the set can then be *classified* as belonging to either one of the sets depending on where it is with respect to the surface $\mathcal{S}$

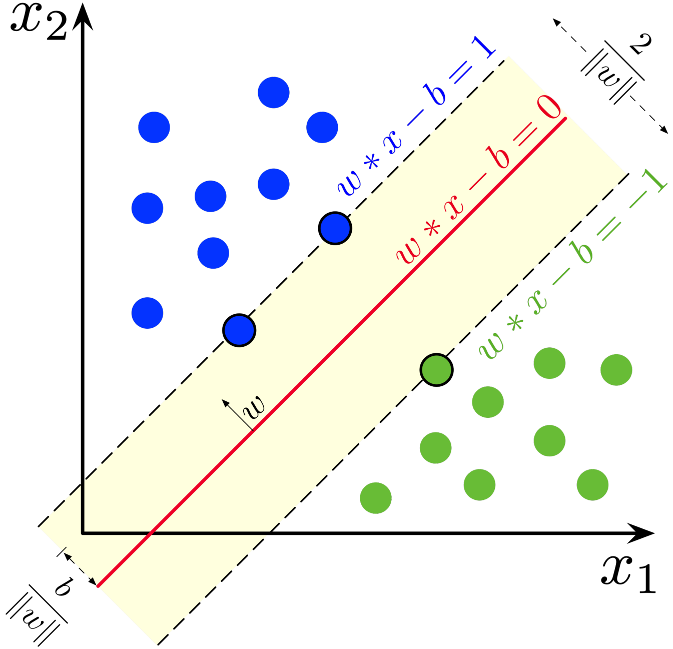
Linear SVM

We are given a training dataset of n points of the form

$$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$$

where $\mathbf{x}_i \in \mathbb{R}^p$ and $y_i = \{-1, 1\}$. The value of y_i indicates the class to which the point $\mathbf{x}_i$ belongs

Linear SVM – *Find the "maximum-margin hyperplane" that divides the group of points* $\mathbf{x}_i$ *for which* $y_i = 1$ *from the group of points for which* $y_i = -1$*, which is such that the distance between the hyperplane and the nearest point* $\mathbf{x}_i$ *from either group is maximized.*



Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are the **support vectors**

Any **hyperplane** can be written as the set of points $\mathbf{x}$ satisfying

$$\mathbf{w}^{\mathsf{T}}\mathbf{x} - b = 0$$

where $\mathbf{w}$ is the (not necessarily normalized) **normal vector** to the hyperplane (if the hyperplane has equation $a_1 z_1 + \cdots + a_p z_p = c$, then $(a_1, \ldots, a_n)$ is normal to the hyperplane)

The parameter $b/\|\mathbf{w}\|$ determines the offset of the hyperplane from the origin along the normal vector $\mathbf{w}$

Remark: a hyperplane defined thusly is not a subspace of $\mathbb{R}^p$ unless $b = 0$. We can of course transform the data so that it is...

Linearly separable points

Let X_1 and X_2 be two sets of points in $\mathbb{R}^p$

Then X_1 and X_2 are **linearly separable** if there exist $w_1, w_2, .., w_p, k \in \mathbb{R}$ such that

- every point $x \in X_1$ satisfies $\sum_{i=1}^{p} w_i x_i > k$
- every point $x \in X_2$ satisfies $\sum_{i=1}^{p} w_i x_i < k$

where x_i is the ith component of x

Hard-margin SVM

If the training data is **linearly separable**, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible

The region bounded by these two hyperplanes is called the “margin”, and the maximum-margin hyperplane is the hyperplane that lies halfway between them

With a normalized or standardized dataset, these hyperplanes can be described by the equations

- $\mathbf{w}^\mathsf{T}\mathbf{x} - b = 1$ (anything on or above this boundary is of one class, with label 1)
- $\mathbf{w}^\mathsf{T}\mathbf{x} - b = -1$ (anything on or below this boundary is of the other class, with label -1)

Distance between these two hyperplanes is $2/\|\mathbf{w}\|$

$\Rightarrow$ to maximize the distance between the planes we want to minimize $\|\mathbf{w}\|$

The distance is computed using the distance from a point to a plane equation

We must also prevent data points from falling into the margin, so we add the following constraint: for each *i* either

$$\mathbf{w}^{\mathsf{T}}\mathbf{x}_i - b \geq 1\,, \text{ if } y_i = 1$$

or

$$\mathbf{w}^{\mathsf{T}}\mathbf{x}_i - b \leq -1\,, \text{ if } y_i = -1$$

(Each data point must lie on the correct side of the margin)

This can be rewritten as

$$y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b) \geq 1, \quad \text{for all } 1 \leq i \leq n$$

or

$$y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b) - 1 \geq 0, \quad \text{for all } 1 \leq i \leq n$$

We get the optimization problem:

Minimize $\|\mathbf{w}\|$ *subject to* $y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b) - 1 \geq 0$ *for* $i = 1, \ldots, n$

The **w** and b that solve this problem determine the classifier, $\mathbf{x} \mapsto \mathrm{sgn}(\mathbf{w}^\mathsf{T}\mathbf{x} - b)$ where $\mathrm{sgn}(\cdot)$ is the **sign function**.

The maximum-margin hyperplane is completely determined by those $\mathbf{x}_i$ that lie nearest to it.

These $\mathbf{x}_i$ are the **support vectors**

Writing the goal in terms of Lagrange multipliers

Recall that our goal is to

$$\textit{minimize } \|\mathbf{w}\| \textit{ subject to } y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b) - 1 \geq 0 \textit{ for } i = 1, \ldots, n$$

Using Lagrange multipliers $\lambda_1, \ldots, \lambda_n$, we have the function

$$L_P := F(\mathbf{w}, b\lambda_1, \ldots, \lambda_n) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \lambda_i y_i(\mathbf{x}_i\mathbf{w} + b) + \sum_{i=1}^{n} \lambda_i$$

Note that we have as many Lagrange multipliers as there are data points. Indeed, there are that many inequalities that must be satisfied

The aim is to minimise L_p with respect to $\mathbf{w}$ and b while the derivatives of L_p w.r.t. λ_i vanish and the $\lambda_i \geq 0$, $i = 1, \ldots, n$

Lagrange multipliers

We have already seen Lagrange multipliers, when we were studying PCA

Maximisation using Lagrange multipliers (V1.0)

We want the max of $f(x_1, \ldots, x_n)$ under the constraint $g(x_1, \ldots, x_n) = k$

1. Solve

$$\begin{aligned} \nabla f(x_1, \ldots, x_n) &= \lambda \nabla g(x_1, \ldots, x_n) \\ g(x_1, \ldots, x_n) &= k \end{aligned}$$

   where $\nabla = (\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n})$ is the **gradient operator**

2. Plug all solutions into $f(x_1, \ldots, x_n)$ and find maximum values (provided values exist and $\nabla g \neq \mathbf{0}$ there)

λ is the **Lagrange multiplier**

The gradient

$f : \mathbb{R}^n \to \mathbb{R}$ function of several variables, $\nabla = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n}\right)$ the gradient operator

Then

$$\nabla f = \left(\frac{\partial}{\partial x_1} f, \ldots, \frac{\partial}{\partial x_n} f\right)$$

So ∇f is a *vector-valued* function, $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$; also written as

$$\nabla f = f_{x_1}(x_1, \ldots, x_n)\mathbf{e}_1 + \cdots f_{x_n}(x_1, \ldots, x_n)\mathbf{e}_n$$

where f_{x_i} is the partial derivative of f with respect to x_i and $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ is the standard basis of $\mathbb{R}^n$

Lagrange multipliers (V2.0)

However, the problem we were considering then involved a single multiplier λ

Here we want $\lambda_1, \ldots, \lambda_n$

Lagrange multiplier theorem

Theorem 1

Let $f : \mathbb{R}^n \to \mathbb{R}$ be the objective function, $g : \mathbb{R}^n \to \mathbb{R}^c$ be the constraints function, both being C^1. Consider the optimisation problem

$$\begin{aligned} \textit{maximize } & f(x) \\ \textit{subject to } & g(x) = 0 \end{aligned}$$

Let x^* be an optimal solution to the optimization problem, such that $\text{rank}(Dg(x^*)) = c < n$, *where $Dg(x^*)$ denotes the matrix of partial derivatives*

$$[\partial g_j / \partial x_k]$$

Then there exists a unique Lagrange multiplier $\lambda^* \in \mathbb{R}^c$ such that

$$Df(x^*) = \lambda^{*T} Dg(x^*)$$

Lagrange multipliers (V3.0)

Here we want $\lambda_1, \ldots, \lambda_n$

But we also are looking for $\lambda_i \geq 0$

So we need to consider the so-called Karush-Kuhn-Tucker (KKT) conditions

Karush–Kuhn–Tucker (KKT) conditions

Consider the optimisation problem

$$\begin{aligned} &\text{maximize } f(x) \\ &\text{subject to} \quad g_i(x) \leq 0 \\ &\qquad\qquad\quad h_i(x) = 0 \end{aligned}$$

Form the Lagrangian

$$L(\mathbf{x}, \mu, \lambda) = f(\mathbf{x}) + \mu^T \mathbf{g}(\mathbf{x}) + \lambda^T \mathbf{h}(\mathbf{x})$$

Theorem 2

If $(\mathbf{x}^*, \mu^*)$ is a [[saddle point]] of $L(\mathbf{x}, \mu)$ in $\mathbf{x} \in \mathbf{X}$, $\mu \geq \mathbf{0}$, then $\mathbf{x}^*$ is an optimal vector for the above optimization problem. Suppose that $f(\mathbf{x})$ and $g_i(\mathbf{x})$, $i = 1, \ldots, m$, are [[Convex function—convex]] in $\mathbf{x}$ and that there exists $\mathbf{x}_0 \in \mathbf{X}$ such that $\mathbf{g}(\mathbf{x}_0) < 0$. Then with an optimal vector $\mathbf{x}^*$ for the above optimization problem there is associated a non-negative vector μ^* such that $L(\mathbf{x}^*, \mu^*)$ is a saddle point of $L(\mathbf{x}, \mu)$

KKT conditions

$$\frac{\partial}{\partial w_\nu} L_P = w_\nu - \sum_i^n \lambda_i y_i x_{i\nu} = 0 \qquad \nu = 1, \dots, p$$

$$\frac{\partial}{\partial b} L_P = -\sum_{i=1}^n \lambda_i y_i = 0$$

$$y_i(\mathbf{x}_i^T \mathbf{w} + b) - 1 \geq 0 \qquad i = 1, \dots, n$$

$$\lambda_i \geq 0 \qquad i = 1, \dots, n$$

$$\lambda_i(y_i(\mathbf{x}_i^T \mathbf{w} + b) - 1) = 0 \qquad i = 1, \dots, n$$

Soft-margin SVM

To extend SVM to cases in which the data are not linearly separable, the **hinge loss** function is helpful

$$\max\left(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\right)$$

y_i is the *i*th target (i.e., in this case, 1 or -1), and $\mathbf{w}^\mathsf{T}\mathbf{x}_i - b$ is the *i*-th output

This function is zero if the constraint is satisfied, in other words, if $\mathbf{x}_i$ lies on the correct side of the margin

For data on the wrong side of the margin, the function's value is proportional to the distance from the margin

The goal of the optimization then is to minimize

$$\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\right)\right]$$

where the parameter $\lambda > 0$ determines the trade-off between increasing the margin size and ensuring that the $\mathbf{x}_i$ lie on the correct side of the margin. Thus, for sufficiently small values of λ, it will behave similar to the hard-margin SVM, if the input data are linearly classifiable, but will still learn if a classification rule is viable or not

Artificial neural network (ANN)

From Wikipedia

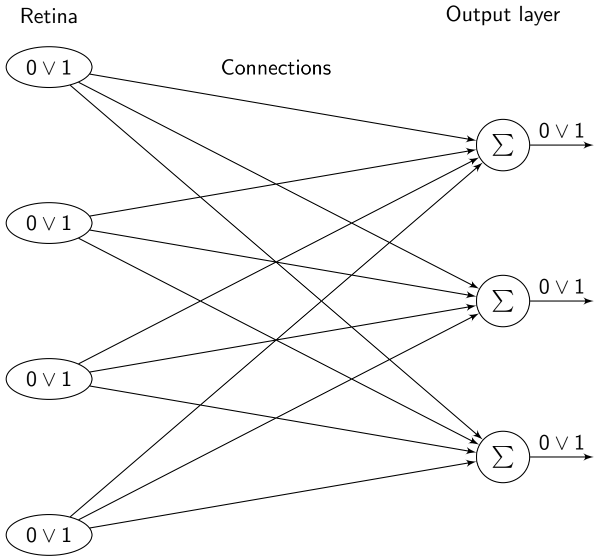
> **Artificial neural networks** *(ANNs) are computing systems inspired by the biological neural networks that constitute animal brains.*
> *An ANN is based on a collection of connected units or nodes called* **artificial neurons**, *which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.*

The perceptron

One of the first neural networks (invented 1943, implemented 1957), made for simple classification tasks, for example recognising letters or numbers

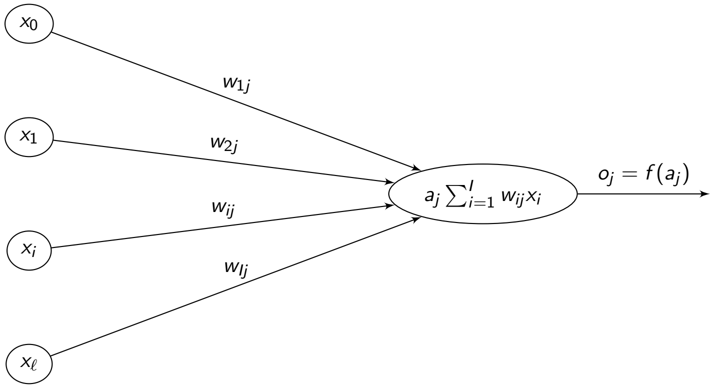
Two layers: the *input* layer (the *retina*) and the *output* layer

Inputs are 0 or 1, and so are outputs



The connections into the output layer are called *synapses*, they are modifiable

The activation function



Here,

$$f(a_j) = \begin{cases} 0 & \text{if } a_j \leq 0 \\ 1 & \text{if } a_j > 0 \end{cases}$$

The activation function

So, we are given I input neurons that take values 0 or 1, O output neurons that take values 0 or 1, weights $W = [w_{ij}] \in \mathcal{M}_{IO}$ and a threshold function f

More generally, use a threshold θ_j for each output neuron,

$$o_j = \begin{cases} 0 & \text{if } a_j \leq \theta_j \\ 1 & \text{if } a_j > \theta_j \end{cases}$$

The thresholds (or *response bias*) and the weights are modifiable by learning. To do that easily for the threshold, consider an input neuron that is always on, say neuron 0, and setting the weights $w_{0j} = -\theta_j$, making the weights matrix an $(I + 1) \times O$-matrix

Another way to write the activation is

$$o_j = \begin{cases} 0 & \text{if } a_j + w_{0j} \leq 0 \\ 1 & \text{if } a_j + w_{0j} > 0 \end{cases}$$

where $w_{0j} = -\theta_j$

Learning something simple

The aim is to adjust the synaptic weights so that the proper response is provided to a given stimulus

Let us first do a simple example: the OR truth table

$$\begin{array}{cccc} 0 & 0 & \mapsto & 0 \\ 1 & 0 & \mapsto & 1 \\ 0 & 1 & \mapsto & 1 \\ 1 & 1 & \mapsto & 1 \end{array}$$

So we will have two neurons in the retina and a single output neuron

Supervised learning

(From R. Rojas)

Supervised learning: *method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured*

The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm

Also called learning with a teacher, since a control process knows the correct answer for the set of selected input vectors

Further distinctions in supervised learning

Supervised learning is further divided into methods which use **reinforcement** or **error correction**

- Reinforcement learning is used when after each presentation of an input-output example we only know whether the network produces the desired result or not. The weights are updated based on this information (that is, the Boolean values true or false) so that only the input vector can be used for weight correction
- In learning with error correction, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights, and in many cases we try to eliminate the error in a single correction step

A first learning algorithm

Suppose the training set consists of two sets of points P and N

- **start:** The weight vector w0 is generated randomly, set $t := 0$
- **test:** A vector $x \in P \cup N$ is selected randomly
  - if $x \in P$ and $w_t \bullet x > 0$ go to **test**
  - if $x \in P$ and $w_t \bullet x \leq 0$ go to **add**
  - if $x \in N$ and $w_t \bullet x < 0$ go to **test**
  - if $x \in N$ and $w_t \bullet x \geq 0$ go to **subtract**
- **add:** set $w_{t+1} = w_t + x$ and $t := t + 1$, goto **test**
- **subtract:** set $w_{t+1} = w_t - x$ and $t := t + 1$, goto **test**

Widrow-Hoff learning rule

Need to provide the correct answer, i.e., this is a supervised learning rule

An output cell only learns if it is mistaken

Present random inputs and apply the rule if the output does not match the known output

Widrow-Hoff learning rule

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \eta(t_j - o_j)x_j = w_{ij}^{(t)} + \Delta w_{ij} \tag{1}$$

with

- Δw_{ij} correction to add to the weight w_{ij}
- x_i: value (0 or 1) of the ith retinal cell
- o_j: response of the jth output cell
- t_j target response (correct desired response)
- $w_{ij}^{(t)}$: weight of the synapse between the ith retinal cell and jth output cell at time t. Typically initiated at small random values
- η: small positive constant, the *learning constant*

Learning OR

$$\begin{array}{cccc} 0 & 0 & \mapsto & 0 \\ 1 & 0 & \mapsto & 1 \\ 0 & 1 & \mapsto & 1 \\ 1 & 1 & \mapsto & 1 \end{array}$$

Three cells in the retina (two inputs and the "dummy" cell used for the threshold) and one output cell. So inputs and outputs must be

$$\begin{array}{ccccc} 1 & 0 & 0 & \mapsto & 0 \\ 1 & 1 & 0 & \mapsto & 1 \\ 1 & 0 & 1 & \mapsto & 1 \\ 1 & 1 & 1 & \mapsto & 1 \end{array}$$

Initialise the 3×1 weight matrix W to zero:

$$W = \begin{pmatrix} w_0 = -\theta \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Procedure

We choose one random association in

$$\begin{array}{cccccc} 1 & 0 & 0 & \mapsto & 0 \\ 1 & 1 & 0 & \mapsto & 1 \\ 1 & 0 & 1 & \mapsto & 1 \\ 1 & 1 & 1 & \mapsto & 1 \end{array}$$

say, the fourth one. So we present $[1, 1, 1]$ and expect an output of 1. We have

$$a = \sum_i w_i x_i = (1 \times 0) + (1 \times 0) + (1 \times 0) = 0$$

This being ≤ 0 means that $o = 0$, giving an error of 1

Applying the rule

Suppose the learning constant $\eta = 0.1$. Then applying (**??**),

$$\begin{aligned}\Delta w_0 &= \eta(t-o)x_0 = 0.1 \times (1-0) \times 1 = 0.1\\ \Delta w_1 &= \eta(t-o)x_0 = 0.1 \times (1-0) \times 1 = 0.1\\ \Delta w_2 &= \eta(t-o)x_0 = 0.1 \times (1-0) \times 1 = 0.1\end{aligned}$$

Applying the correction, W becomes

$$W = \begin{pmatrix} 0.1 \\ 0.1 \\ 0.1 \end{pmatrix}$$

Trying another input

Suppose we now present the first input, i.e., $[1, 0, 0]$, which should produce a result of 0. Then

$$a = \sum_i w_i x_i = (1 \times 0.1) + (0 \times 0.1) + (0 \times 0.1) = 0.1$$

which is > 0, so $o = 1$. We compute the correction

$$\begin{aligned}
\Delta w_0 &= \eta(t - o)x_0 = 0.1 \times (0 - 1) \times 1 = -0.1 \\
\Delta w_1 &= \eta(t - o)x_1 = 0.1 \times (0 - 1) \times 0 = 0 \\
\Delta w_1 &= \eta(t - o)x_2 = 0.1 \times (0 - 1) \times 0 = 0
\end{aligned}$$

and adjust the weights, giving

$$W = \begin{pmatrix} 0 \\ 0.1 \\ 0.1 \end{pmatrix}$$

And we are done!

With the weights

$$W = \begin{pmatrix} 0 \\ 0.1 \\ 0.1 \end{pmatrix}$$

we are done. Indeed

Input 0	Input 1	Input 2	a	o	Should be
1	0	0	0	0	0
1	1	0	0+0.1+0	1	1
1	0	1	0+0+0.1	1	1
1	1	1	0+0.1+0.1	1	1

Learning XOR

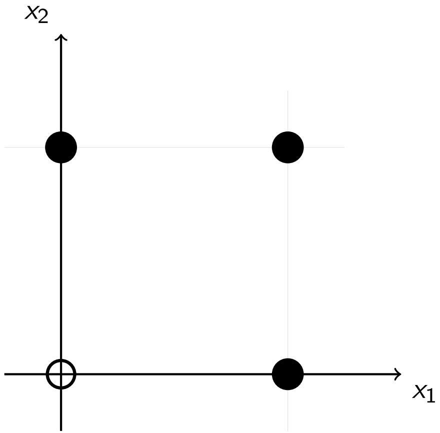
Let us now look at the XOR truth table

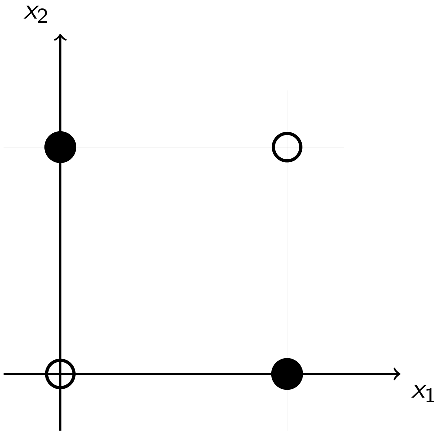
$$\begin{array}{cccc} 0 & 0 & \mapsto & 0 \\ 1 & 0 & \mapsto & 1 \\ 0 & 1 & \mapsto & 1 \\ 1 & 1 & \mapsto & 0 \end{array}$$

This problem is not solvable with a simple perceptron of the type we just used, as truth table is not *linearly separable*

Indeed, we would get weights $w_1 > 0$, $w_2 > 0$ to activate when presenting $[1, 0]$ and $[0, 1]$, but would require that the sum of the weights when applied to the input $[1, 1]$, give a negative value.

Linear separability and OR and XOR

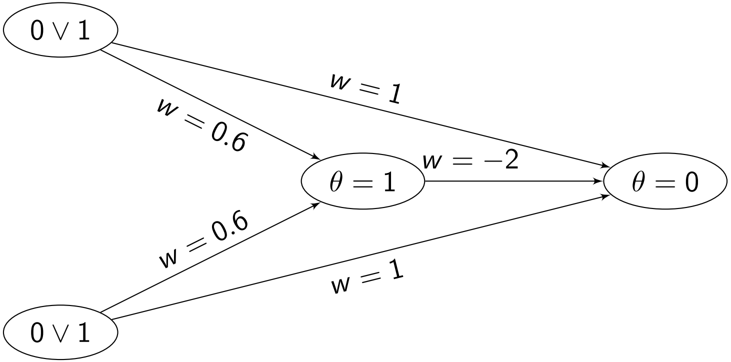




A single-layer perceptron can only learn linearly separable problems

Adding a hidden layer

It is possible to do XOR, but we need to add a **hidden layer**


$0 \vee 1$
$0 \vee 1$
$w = 0.6$
$w = 0.6$
$w = 1$
$w = 1$
$\theta = 1$
$w = -2$
$\theta = 0$