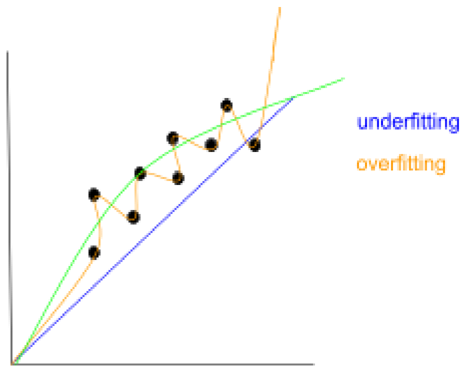
# Machine Learning

"Field of study that gives computers the ability to learn with out being explicitly programmed" - Arthur Samuel

## **<u>Example</u> "fitting" over a linear regression**



- underfitting $y = \theta_0 + \theta_1 x_1$
- overfitting $y = \sum_{i=0}^{d} \theta_j x_1^j$
- best size $y = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$

## 1. Locally weighted regression

| Model | Loss function | Output |
|---|---|---|
| Linear | $\sum_i \left(y^{(i)} - \theta^T x^{(i)}\right)^2$ | $\theta^T x$ |
| Locally weighted (LWR) | $\sum_i w^{(i)} \left(y^{(i)} - \theta^T x^{(i)}\right)$ | $\theta^T x$ |

Where

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

And $x^{(i)}$ being a *train example*, $x$ a *test example*, $\tau$ the *bandwidth parameter*.

Note that $0 \leq w^{(i)} \leq 1$ and for examples very far from each other it tends to $0$, for examples very close to each other it tends to $1$

In LWR, the output $\theta^T x$ gets re-estimated for every test example. You calculate a new weight for the specific test example (each one), so the bigger the data set, the bigger the value of the model.

## 2. Logistic regression (classification)

When we talked about the linear regression, we meant a model such that

$$\mathbb{R}^n \to \mathbb{R}$$

While in linear classification, we are considering a *restricted* set of $y^{(i)}$s

$$\mathbb{R}^n \to \{0,\ 1\}$$

The previous example refers explicitly to a binary classification, but it could be a different set.

| Linear regression | Logistic regression |
|---|---|
| $y \in \mathbb{R}$ | $y \in \{0,1\}$ |
| $h_\theta(x) = \theta^T x$ | $h_\theta(x) = g(\theta^T x) = \dfrac{1}{1+\exp(-\theta^T x)}$ |

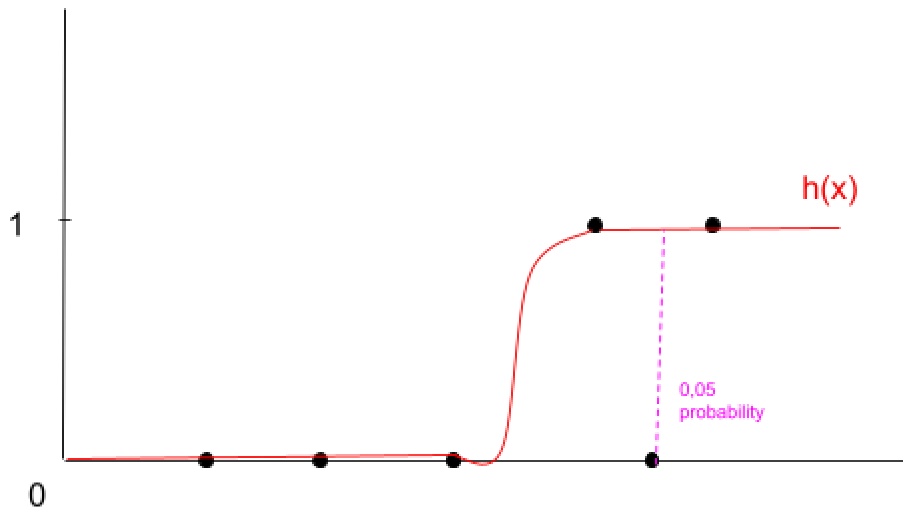
The classification's hypothesis function is called *logistic function*

$$h_\theta^{logistic} = \frac{1}{1+\ \exp(-\theta^T x)}$$

So the actual score $\theta^T x$ determines if the output of the function and

$$0 \le g(\theta^T x) \le 1$$

Ideally, this mean that the function *shrinks* all the possible outcomes in a $\{0,1\}$ range.



Now, let's talk about probabilities

$$\begin{cases} P(y=1||x_1\theta) = h_\theta(x) \\ P(y=0||x_1\theta) = 1 - h_\theta(x) \end{cases}$$

is a *Bernoulli distribution*.

$x$ could be any transformation $(x^2, x^n, \dots)$

Let's rearrange the terms and we obtain

$$P(y||x_1\theta) = h_\theta(x)^y(1-h_\theta(x))^{1-y}$$

the probability distribution function.

Let's now define $L(\theta)$ the likelihood function

$$L(\theta) \; = \; P(y||x_1\theta) = \prod_{i=1}^{n} P\left(y^{()i)}||x_1^{(i)}\theta\right)$$

$$L(\theta) = \prod_{i=1}^{n} h_\theta\left(x^{(i)}\right)^{y^{(i)}} \left[1 - h_\theta\left(x^{(i)}\right)\right]^{1-y^{(i)}}$$

$$\log L(\theta) = \sum_{i=1}^{n} \left\{ y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left[1 - h_\theta\left(x^{(i)}\right)\right] \right\}$$

Now, let's apply *gradient ascent* to $L(\theta)$ in order to maximize the likelihood

$$\begin{cases} \theta = 0 \\ \theta_k = \theta_{k-1} + \alpha \nabla_\theta \log L(\theta) \end{cases}$$ initialization

$$\nabla_\theta \log L(\theta) = \sum_{i=1}^{n} y^{(i)} \nabla_\theta \log \frac{1}{1+\exp\left(-\theta^T x\right)} + \left(1 - y^{(i)}\right) \nabla_\theta \log \left(1 - \frac{1}{1+\exp\left(-\theta^T x\right)}\right)$$

The gradient of the function is

$$\nabla_\theta L(\theta) = \sum_{i=1}^{n} \left(y^{(i)} - h_\theta\left(x^{(i)}\right)\right) x^{(i)}$$

Now, repeat the algorithm until convergence and make predictions by calculating $\theta^T x$ (we want it to be >> or <<, so the prediction is either $1$ or $0$.

## 3. Newton's method

NsM is an optimization algorithm, very efficient to find zeros of a function. In respect to gradient descent, Newton's converges faster if the considered starting point is close to the solution. In this case, we say Newton's is more efficient since the 0 of a gradient might or not be a *minimum.*

In general, the Newton's algorithm looks like this

$$\theta_k = \theta_{k-1} - \frac{f(\theta)}{f'(\theta)}$$

And iterate until convergence $f(\theta) = 0$

Now let's apply the gradient, set $f = L'$ and we obtain

$$\theta_k = \theta_{k-1} - \frac{L'(\theta)}{L''(\theta)}$$

Now, let's consider an $\mathbb{R}^n$ vector space and we finally obtain the expression to iterate

$$\theta_k = \theta_{k-1} - \alpha H^{-1} \nabla_\theta L(\theta)$$

where $H^{-1}$ is the *hessian inverse*.

$$H_{ij} = \frac{\partial^2 L(\theta)}{\partial\theta_i \partial\theta_j}$$

Newton's always puts you into the closest *max* or *min* in the vector space. This method if really effective for small values of $n$, while it is not for higher values.

## 4. Stochastic gradient descent

In *linear regression* we have defined the *train loss function* $TrainLoss(\theta)$ as follows

$$TrainLoss = \sum_{i=0}^{n} J\left(\theta,\, x^{(i)},\, y^{(i)}\right)$$

where

$$J\left(\theta, x^{(i)}, y^{(i)}\right) = \left(\theta^T x^{(i)} - y^{(i)}\right)$$

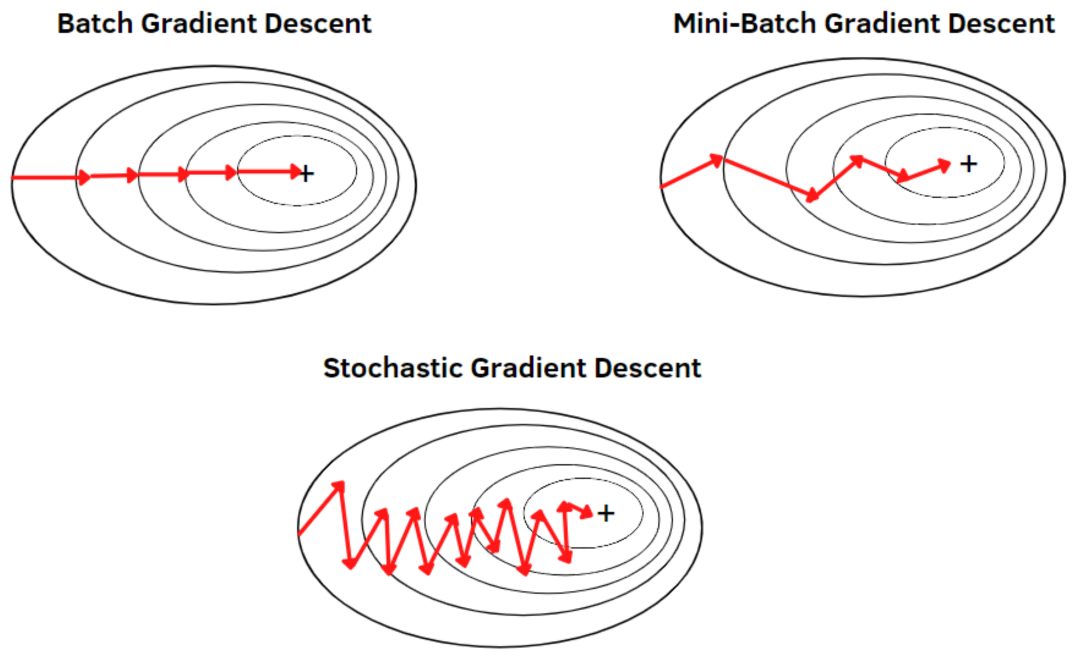
So the gradient descent's update rule would look like this

$$\theta_k = \theta_{k-1} - \alpha \sum_{i=0}^{n} \nabla_\theta J\left(\theta,\, x^{(i)}, y^{(i)}\right)$$

Which can be computationally expensive and inefficient if the data set is big.
The *SGD* update rule, instead of taking the sum of the gradients of all the train examples with respect to $\theta$ and iterating this process until convergence, uses the following function

$$\tilde{L}(\theta) = J\left(\theta,\, x^{(r)},\, y^{(r)}\right), \;\; r \sim [0,1]$$

that takes one *random* example at a time.
This is much more efficient for big data sets. It is also important to mention that the idea of *convergence* is also different in SGD. Here the convergence means that the gradients reached a point that is *within a circle around the converge point* and the radius of the circle equals to *learning grade* $\alpha$ (or step size).


Batch Gradient Descent
Mini-Batch Gradient Descent
Stochastic Gradient Descent