Machine Learning Definition:

- Arthur Samuel (1959): field of study that gives computers the ability to learn without being explicitly programmed.

- Tom Mitchell (1998): A computer program is said to learn from experience E with respect to some task T and some perfomance P, if its performance on T, as measured by P, improves with expierence E.

Supervised Learning

- most widely used ML tool.
- a given dataset with x and y labels.
  Goal: find mapping $f: x \rightarrow y$
- Regression problem: using a straight line to map the relationships
- Classification problem: k-different outputs to assign

Unsupervised Learning

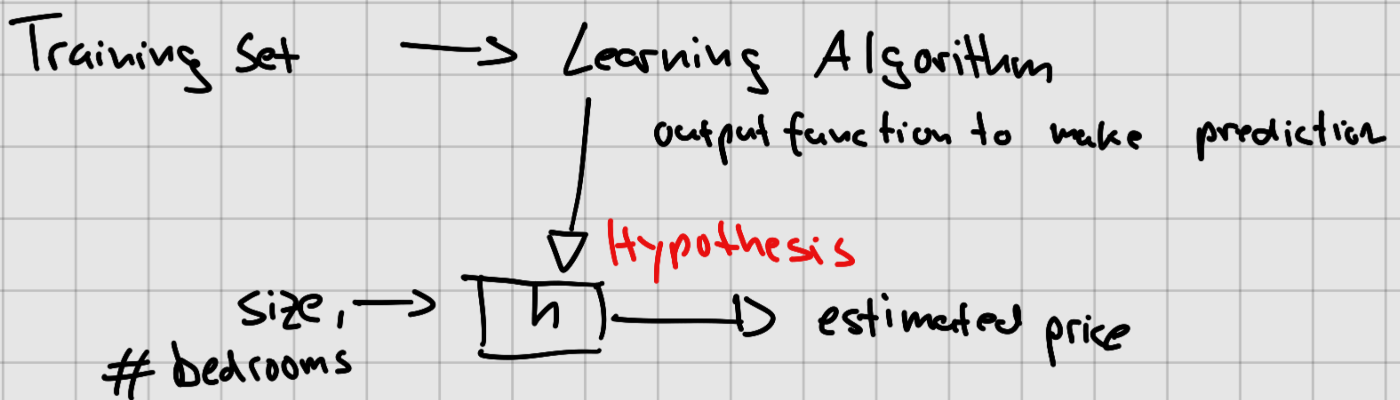
- No labels are given, find interesting structure in the data
- finding what groups in the data belong together
- Social network analysis, market segmentation, cocktail party problem

Linear Regression and Gradient Descent

Predict prices of houses:

Size	# bedrooms	price
2104	3	400
1416	3	232
1536	3	315
852	2	178

(Size, # bedrooms) = X; price = Y



How do we represent h?

x_1 = size
x_2 = # bedrooms

$$h(x) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2$$

(*)

$$h(x) = \sum_{j=0}^{2} \theta_j \cdot x_j \,, \text{ where } x_0 = 1$$

(2 → # features)

$$h_\theta(x) = \sum_{i=1}^{n} \theta_i \cdot x_i = \theta^T \cdot x$$

parameters/weights $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$, features (input) $x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$

n = # features
m = # training examples (rows)
(x, y) = training example
y = output

$(x^{(i)}, y^{(i)})$ = i-th training-example

$x_1^1 = 2104$, $x_2^1 = 3$

$x_1^2 = 1416$

Now, given a dataset, how do we learn/pick the parameters θ ?

Choose θ so that $h(x) \approx y$ for training example.

Choose values of θ that minimize $\frac{1}{2} \sum_{i=0}^{m} (h(x^{(i)}) - y)^2 = J(\theta)$ — cost function

$\frac{1}{2}$: makes math easier

$(h(x^{(i)}) - y)^2$: squared loss function

Why do we use the square loss ?

The squared error forces h(x) and y to match. It is minimized at $u = v$, if possible and is always ≥ 0 because it is a square of a reel number $u - v$.

Why is the squared loss function better than others?

If your data does not fit all points exactly, i.e. $h(x) - y$ is never 0 for some point no matter what θ you choose (as it will always happen in practise), that might be due to noise. In any complex system there will be many small independent causes for the difference between h(x) and y. (measurement error, enviromental factors).

According to the Central Limit Theorem, the total noise would be distributed normally => We want to pick the best fit θ taking this noise distribution into account.

Assume $R = h(x) - y$, the part of y that your model can not explain, follows Normal distribution $N(\mu, \sigma)$

The Normal distribution has two parameters

$$\mu = E[R] = \frac{1}{m} \sum (h_\theta(x_i) - y_i)$$ (systematic error of our measurement)

↳ to correct: $\mu' = E[R'] = 0$

$$\sigma^2 = E[R^2] = \frac{1}{m} \sum (h_\theta(x^i) - y^i)^2$$ (random error = noise)

The best predictor is the one with the tightest distribution (smallest variance) around the predicted value.

$\frac{1}{2}(\ldots)^2$ => cancel each other out

Gradient descent (Least mean squares algorithm)

Start with some value of θ (say $\theta = \vec{0}$)
Keep changing θ, until hopefully, we converge to a value of θ that minimizes $J(\theta)$.

α = Learning rate

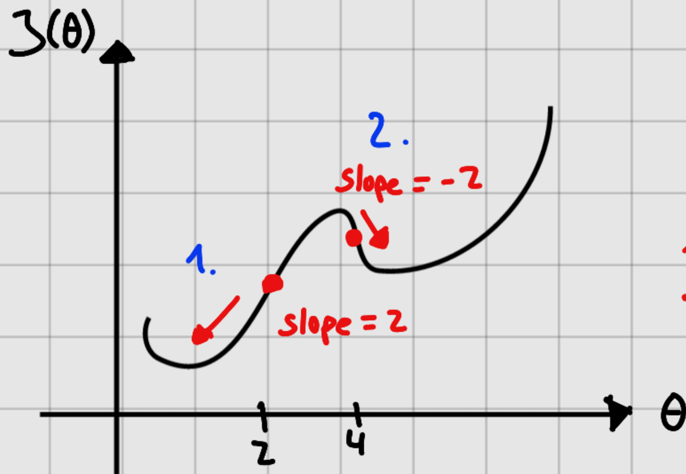
$$\theta_j = \theta_j - \alpha \cdot \frac{d}{d\theta_j} J(\theta)$$

(This update is simultaneously performed for all values of $j = 0, \ldots, n$). Takes repeatedly a step in the direction of steepest decrease of J.

$$\theta_j = \theta_j - \frac{d}{d\theta_j} J(\theta)$$

why is this minus ?

Because $\frac{d}{d\theta_j} J(\theta)$ is the direction of change (steepness).



If you set $\theta = \theta - \alpha \cdot$ slope at the points, $J(\theta)$ will decrease: $\alpha = 0.05$

1. $2 - \alpha \cdot (2) = 1.9$
2. $4 - \alpha \cdot (-2) = 4.1$

If we set $\theta = \theta + \alpha \cdot$ slope, then $J(\theta)$ will increase.

In order to implement the algorithm, we have to work out what is the partial derivative term on the right side:

Lets first work it out for one training example, so we can neglect the sum in the definition of J:

$$\frac{d}{d\theta_j} J(\theta) = \frac{d}{d\theta_j} \frac{1}{2} (h_\theta(x) - y)^2$$

$$= 2 \cdot \frac{1}{2} \cdot (h(x) - y) \cdot \frac{d}{d\theta_j} (h_\theta(x) - y)$$

$$= (h(x) - y) \cdot \frac{d}{d\theta_j} (\theta_0 \cdot x_0 + \theta_1 \cdot x_1 + \ldots + \theta_n \cdot x_n - y) \quad (*)$$

$$= (h(x) - y) \cdot x_j$$

(*) All of these terms do not depend on θ_j = const. => 0
Except for '$\theta_j \cdot x_j$' => x_j

For a single training example, this gives us the update rule:

$$\theta_j := \theta_j + \alpha (y^{(i)} - h_\theta(x^{(i)})) \cdot x_j^{(i)}$$

This rule is called the LMS update rule, and is also known as the Widrow-Hoff learning rule.

There are two ways to modify this method for a training set of more than one example.

Repeat until convergence { (Batch gradient descent)

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}, \text{(for every } j)$$

}

1 batch = 1 training example

Disadvantage: we have to calculate the sum over m in order to do one step.
If m is very large (100.000.000) => very slow

There is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

Loop { (Stochastic gradient descent)

for i = 1 .... m {

$$\theta_j := \theta_j - \alpha (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

(for every j)

}

}

In this algorithm we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that singe training example only.

Whereas batch gradient descent has to scan through the entire training set before taking a single step - a costly operation if n is large - stochastic gradient descent can start the making progress right away, and continues to make progress with each example it looks at. Often, stochastic gradient descent gets θ "close" to the minimum faster than batch gradient descent. Monitor $J(\theta) \Rightarrow$ No drastic change $\Rightarrow$ stop algorithm

Normal equations

Gradient descent gives us one way to minimize J. Let's discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In this method, we will minimize J by explicitly taking its derivatives with respect to the θ_j's, and setting them to zero. To enable us to do this without having to write realms of algebra and pages full of matrices of derivatives, let's introduce some notation for doing calculus with matrices.

Matrix derivatives

For a function $f: \mathbb{R}^{m \times n} \to \mathbb{R}$, we define the derivative of f with respect to A to be:

$$\nabla_A f(A) = \begin{bmatrix} \frac{df}{dA_{11}} & \cdots & \frac{df}{dA_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{df}{dA_{m1}} & \cdots & \frac{df}{dA_{mn}} \end{bmatrix}$$

Thus, the gradient $\nabla_A f(A)$ is itself an $m \times n$ matrix, whose (i,j)-element is $\partial f / \partial A_{ij}$.

For example, suppose $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ is a 2 by 2 matrix and the function $f: \mathbb{R}^2 \to \mathbb{R}$ is given by

$$f(A) = \frac{3}{2} A_{11} + 5 A_{12}^2 + A_{21} \cdot A_{22}$$

We then have:

$$\nabla_A f(A) = \begin{bmatrix} \frac{3}{2} & 10 A_{12} \\ A_{22} & A_{21} \end{bmatrix}$$

Armed with the tools of matrix derivatives, let us now proceed to find in closed-form the value of θ that minimizes $J(\theta)$. We begin rewriting J in matrix-vectorial notation.

Given a training set, define the design matrix X to be the $m \times n$ matrix that contains the training examples' input variables in its rows

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$$

Also, let $\vec{y}$ be the m-dimensional vector containing all the target values from the training set:

$$\vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Now, since $h_\theta(x^{(i)}) = (x^{(i)})^T \cdot \theta$, we can easily verify that

$$X \cdot \theta - \vec{y} = \begin{bmatrix} (x^{(1)})^T \cdot \theta \\ \vdots \\ (x^{(n)})^T \cdot \theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

$$= \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(n)}) - y^{(n)} \end{bmatrix}$$

Thus using the fact that for a vector z, we have that $z^T \cdot z = \sum_i z_i^2$

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

$$= \frac{1}{2} \cdot (h_\theta(x^{(i)}) - y^{(i)})^T \cdot (h_\theta(x^{(i)}) - y^{(i)})$$

Finally, to minimize J, let's find its derivative with respect to θ. Hence,

$$\nabla_\theta J(\theta) = \nabla_\theta \frac{1}{2}(X\cdot\theta - y)^T \cdot (X\cdot\theta - \vec{y})$$

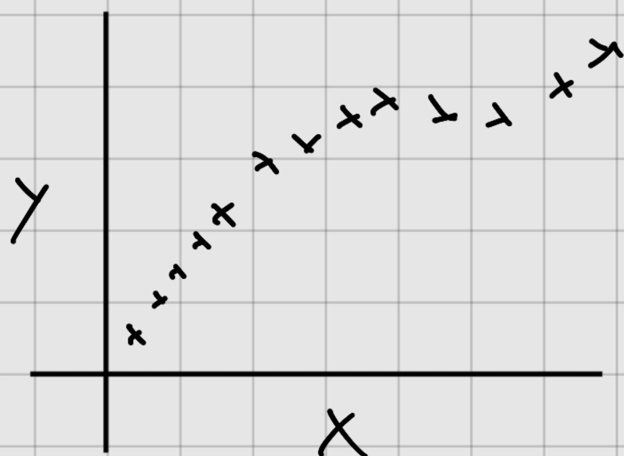
$$= \frac{1}{2} \cdot \nabla_\theta\left((X\cdot\theta)^T X\theta - (X\theta)^T\vec{y} - \vec{y}^T(X\theta) + y^T\cdot y\right)$$

$$= \dots = X^T\cdot X\theta - X^T\cdot\vec{y} = \dots\ \dots$$

Thus, the value of θ that minimizes $J(\theta)$ is given in closed form by the equation

$$\theta = (X^T\cdot X)^{-1}\cdot X^T\cdot\vec{y}$$

Locally weighted & Logistic regression



=> straight line would not provide a good fit

Instead we add an extra feature x^2 and fit $y = \theta_0 + \theta_1 \cdot x + \theta_2 \cdot x^2$, then we obtain a slightly better fit.

Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features. Even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for all diverent living areas (x).

Straight line = underfitting, in which the data clearly shows structure not captured by the model.

x^5 - features = overfitting

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 $\theta^T \cdot x$

In contrast, the locally weighted linear regression algorithm does the following:

1. Fit θ to mimize $\sum_i w^{(i)} \cdot (y^{(i)} - \theta^T \cdot x^{(i)})^2$.

2. Output $\theta^T \cdot x$

Where $w^{(i)}$ is a weight function. If $|x^{(i)} - x|$ (training example $x^{(i)}$, prediction location x) is small, then $w^{(i)}$ is close to 1. If $|x^{(i)} - x|$ is big, then $w^{(i)}$ is close to 0.

locally weighted regression

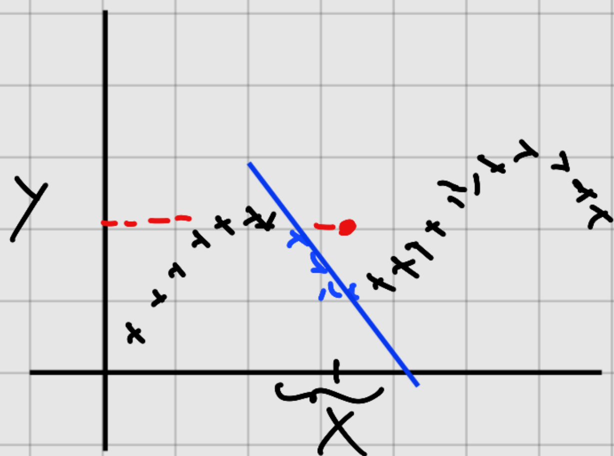
"Parametric" learning algorithm: (linear regression)

- Fit a fixed set of parameters (θ_i) to data.
- No matter how big your training set is: you fit the parameters. Then you could erase the whole training set and make predictions just using the parameters.

Here you need to keep more

"Non-parametric" learning algorithm: (locally weighted).

- Amount of data/parameters you need to keep grows linearly with size of data.
- Not great for big data sets



$0 < w^{(i)} < 1.$

To evaluate h at a certain x:

LR: fit θ to minimize cost function. Return $\theta^T x$

locally weighted:

fit θ to minimize the modified cost function $\sum_{i=1}^{m} w^{(i)} \cdot (y^{(i)} - \theta^T x^{(i)})^2$

where $w^{(i)}$ is a weight function.

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

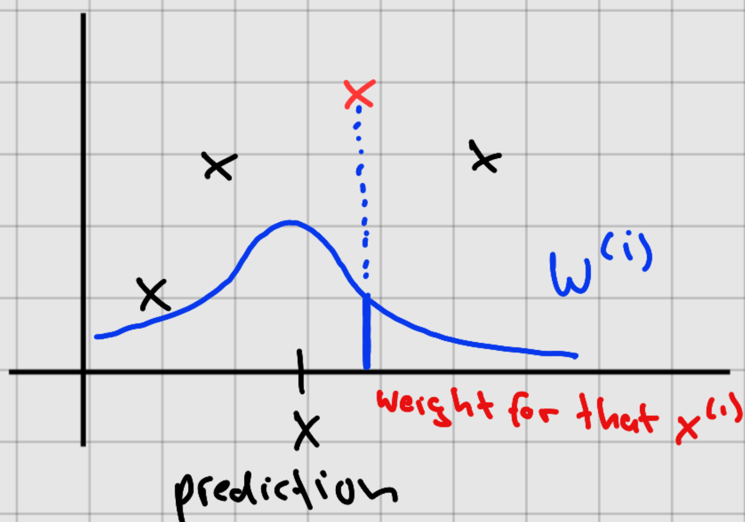
If $|x^{(i)} - x|$ is small, $w^{(i)} \approx 1$. x is the location where we want to make a prediction. $x^{(i)}$ is the input x for the i-th training example. $w^{(i)}$ tells us how much we should pay attention to the values of $(x^{(i)}, y^{(i)})$ when fitting the line.

What does locally weighted regression do?

If an example $x^{(i)}$ is far from x $\Rightarrow w^{(i)} \approx 0$

If an example $x^{(i)}$ is close to x $\Rightarrow w^{(i)} \approx 1$

The sum is then essentialy over the term, that are close to x. Terms that were multiplied by 0 dissappear.



How do we choose the width?

τ (tau) = band width, controls how quickly the weight of a training example falls off

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

Depending on the choice of τ, we can choose a fatter or slimmer bell shaped curve. Which causes you to to look in a bigger or narrower window in order to decide how many nearby examples to use in order to fit the straight line.

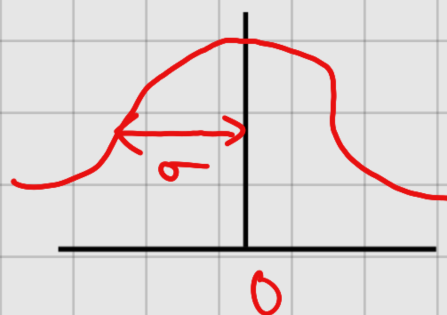
Why do we use the squared error in the cost function?

Assume there is a true price of a house $y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}$, where $\varepsilon^{(i)}$ is an error term that includes unmodelled effects + random noise

Assume that $\varepsilon^{(i)}$ is distributed $\mathcal{N}(0, \sigma^2)$ normal distribution

The probability density:

$$P(\varepsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\varepsilon^{(i)})^2}{2\sigma^2}\right)$$



This implies:

Given $x^{(i)}$ and Theta, what is the probability of a particular house's price?

$$P(y^{(i)} \mid x^{(i)}; \theta) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(y^{(i)} - \theta^T \cdot x^{(i)})^2}{2\sigma^2}\right)$$

mean

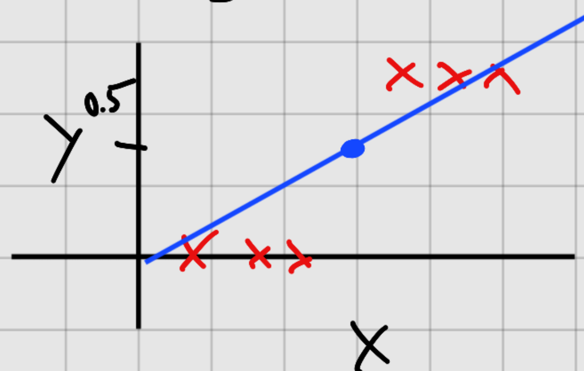
variance

$$\Rightarrow y^{(i)} \mid x^{(i)}; \theta \sim \mathcal{N}(\theta^T \cdot x^{(i)}, \sigma^2)$$

Classification

- binary classification $\Rightarrow y \in \{0,1\}$ (two classes)

Applying linear regression is not a good idea.

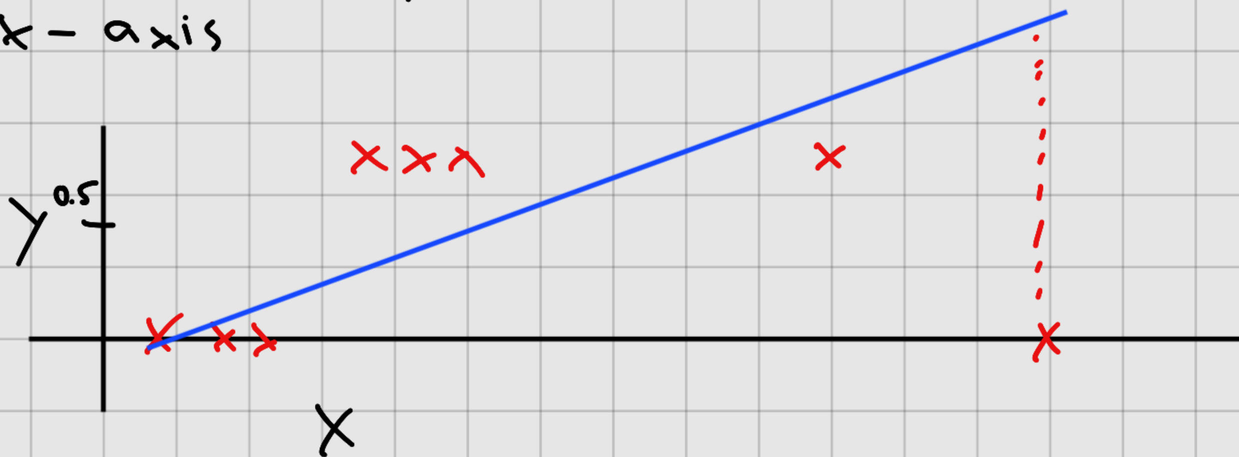


We could, for example, fit a straight line like this and threshhold it at 0.5. Everything above 0.5 will be rounded off to 1 and everything below to 0.

Let's now consider the same dataset but include an outlier:

Due to the outlier, our line shifts and the x value with the assigned threshhold changes. This results in our model not making accurate assumptions.

Another example, where LR fails: Outliers on the x-axis



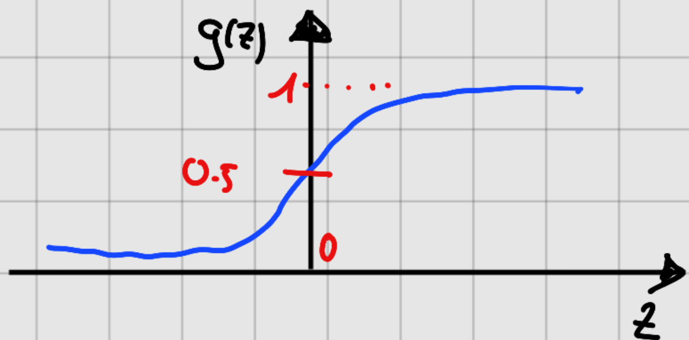
We can easily see that x value would get an y-value which is greater than 1.

Logistic regression

$$g(z) = \frac{1}{1+e^{-z}}$$

Sigmoid function

As we could see it doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0,1\}$



We want $h_\theta(x) \in [0,1]$

$$h_\theta(x) = g(\theta^T \cdot x) = \frac{1}{1+e^{-\theta^T \cdot x}}$$

$$P(y=1 \mid x) = h_\theta(x)$$

$$P(y=0 \mid x) = 1 - h_\theta(x)$$

express these in one, since we know that $y \in \{0,1\}$

$$P(y \mid x;\theta) = h_\theta(x)^y \cdot (1 - h_\theta(x))^{1-y}$$

Assuming that we are having n training examples, we can then write down the likelihood of the parameters as

$$L(\theta) = p(\vec{y} \mid X;\theta)$$

$$= \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)};\theta)$$

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

It is easier to maximize the log likelihood:

$$\Rightarrow \sum_{i=1}^{n} y^{(i)} \cdot \log(h_\theta(x^{(i)})) + (1-y^{(i)}) \cdot \log(1-h_\theta(x^{(i)}))$$

Now we need to choose a θ to maximize $\ell(\theta)$:

↳ Algorithm Batch gradient ascent

$$\theta_j := \theta_j + \frac{d}{d\theta_j} \ell(\theta) \quad \left(\begin{array}{l} \text{maximize} = + \\ \text{minimize} = - \end{array} \right)$$

Let's start by working with just one training example (x, y) and take derivatives to derive the stochastic gradient ascent rule:

$$\frac{d}{d\theta_j} \ell(\theta) = (y - h_\theta(x)) \cdot x_j$$