Machine Learning and Bayesian Inference

Dr Sean Holden

Computer Laboratory, Room FC06

Telephone extension 63725

Email: `sbh11@cl.cam.ac.uk`

`www.cl.cam.ac.uk/~sbh11/`

Artificial Intelligence: what have we seen so far?

What did we learn in Artificial Intelligence I?

1. *We used logic for knowledge representation and reasoning*. However we saw that logic can have drawbacks:
   (a) *Laziness:* it is not feasible to assemble a set of rules that is sufficiently exhaustive. If we could, it would not be feasible to apply them.
   (b) *Theoretical ignorance:* insufficient knowledge *exists* to allow us to write the rules.
   (c) *Practical ignorance:* even if the rules have been obtained there may be insufficient information to apply them.

Instead of considering *truth* or *falsity*, deal with *degrees of belief*.

Probability theory is the perfect tool for application here.

Probability theory allows us to *summarise* the uncertainty due to laziness and ignorance.

Artificial Intelligence: what have we seen so far?

What did we learn in Artificial Intelligence I?

2. We looked at how to choose a *sequence of actions* to *achieve a goal* using *search*, *adversarial search (game-playing)*, *logical inference (situation calculus)*, and *planning*.
    - All these approaches suffer in the same way as inference.
    - So *all benefit* from considering uncertainty.
    - All implicitly deal with *time*. How is this possible under uncertainty?
    - All tend to be trying to reach *goals*, but these may also be uncertain.

Utility theory is used to assign preferences.

Decision theory combines probability theory and utility theory.

A *rational* agent should act in order to *maximise expected utility* as time passes.

Artificial Intelligence: what have we seen so far?

What did we learn in Artificial Intelligence I?

3. We saw some basic ways of *learning from examples*.
    - Again, there was no real mention of *uncertainty*.
    - Learning from *labelled examples* is only one kind of learning.
    - We did not consider how learning might be applied to the *other tasks in AI*, such as planning.

We need to look at *other ways of learning*.

We need to introduce *uncertainty* into learning.

We need to consider *wider applications* of learning.

Artificial Intelligence: what are we going to learn now?

What are we going to learn now?

In moving from logic to probability:

- We replace the *knowledge base* by a *probability distribution* that represents our beliefs about the world.
- We replace the task of *logical inference* with the task of *computing conditional probabilities*.

Both of these changes turn out to be *considerably more complex than they sound*.

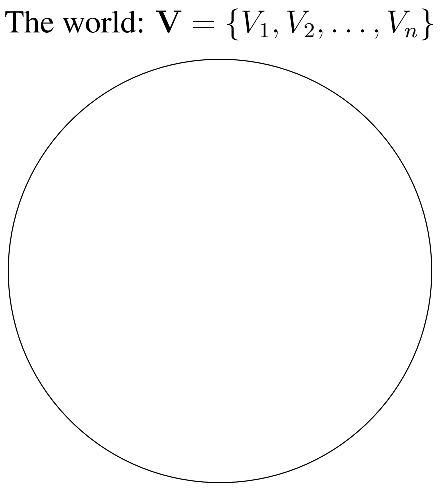
Bayesian networks and *Markov random fields* allow us to represent *probability distributions*.

Various algorithms can be used to perform *efficient inference*.

General knowledge representation and inference: the BIG PICTURE

The current approach to *uncertainty* in AI can be summed up in a few sentences:

Everything of interest in the world is a *random variable*. The *probabilities* associated with RVs summarize our *uncertainty*.



If the n RVs $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$ represent everything of interest, then our *knowledge base* is the *joint distribution*

$$\boxed{\Pr(\mathbf{V}) = \Pr(V_1, V_2, \ldots, V_n)}$$

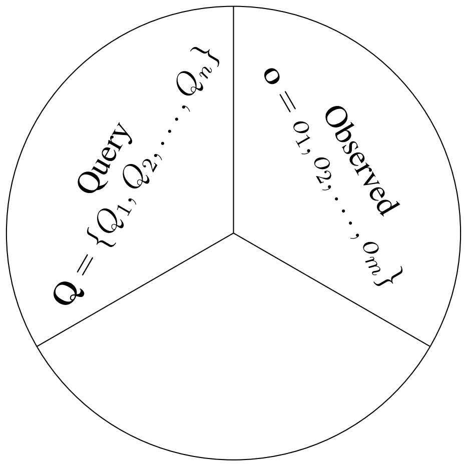
General knowledge representation and inference: the BIG PICTURE

Say we have *observed* the values of a subset $\mathbf{O} = \{O_1, O_2, \ldots, O_m\}$ of m RVs.

In other words, we know that $(O_1 = o_1, O_2 = o_2, \ldots, O_m = o_m)$.

Also, say we are interested in some subset $\mathbf{Q}$ of *query variables*.

The world: $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$

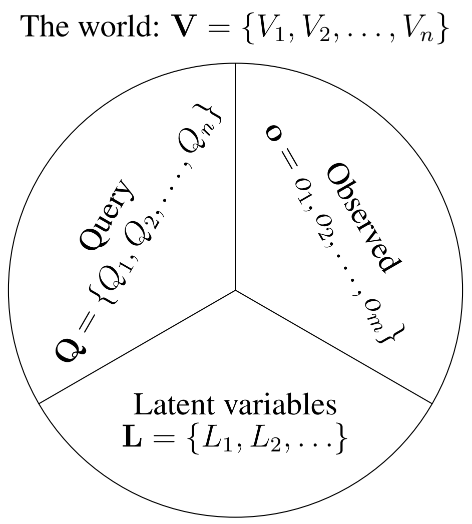

Query
$\mathbf{Q} = \{Q_1, Q_2, \ldots, Q_n\}$
Observed
$\mathbf{o} = o_1, o_2, \ldots, o_m\}$

Then *inference* corresponds to computing a *conditional distribution*

$$\boxed{\Pr(\mathbf{Q} | o_1, o_2, \ldots, o_m)}$$

General knowledge representation and inference: the BIG PICTURE

The *latent variables* $\mathbf{L}$ are *all the RVs not in the sets* $\mathbf{Q}$ *or* $\mathbf{O}$.



The world: $\mathbf{V} = \{V_1, V_2, \dots, V_n\}$
Query
$\mathbf{Q} = \{Q_1, Q_2, \dots, Q_n\}$
Observed
$\mathbf{o} = o_1, o_2, \dots, o_m\}$
Latent variables
$\mathbf{L} = \{L_1, L_2, \dots\}$

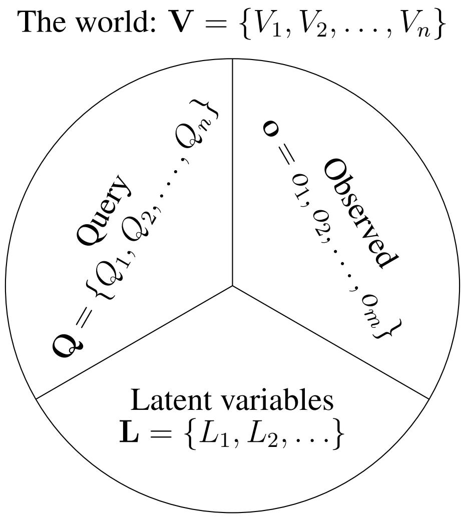
To compute a conditional distribution from a knowledge base $\Pr(\mathbf{V})$ we have to *sum over the latent variables*

$$\Pr(\mathbf{Q}|o_1, o_2, \dots, o_m) = \sum_{\mathbf{L}} \Pr(\mathbf{Q}, \mathbf{L}|o_1, o_2, \dots, o_m)$$

$$= \boxed{\frac{1}{Z} \sum_{\mathbf{L}} \underbrace{\Pr(\mathbf{Q}, \mathbf{L}, o_1, o_2, \dots, o_m)}_{\text{Knowledge base}}}$$

General knowledge representation and inference: the BIG PICTURE

Bayes' theorem tells us how to update an inference when *new information* is available.


The world: $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$
Query
$\mathbf{Q} = \{Q_1, Q_2, \ldots, Q_n\}$
Observed
$\mathbf{o} = o_1, o_2, \ldots, o_m\}$
Latent variables
$\mathbf{L} = \{L_1, L_2, \ldots\}$

For example, if we now receive a new observation $O' = o'$ then

$$\boxed{\underbrace{\Pr\left(\mathbf{Q}|o', o_1, o_2, \ldots, o_m\right)}_{\text{After } O' \text{ observed}} = \frac{1}{Z}\Pr\left(o'|\mathbf{Q}, o_1, o_2, \ldots, o_m\right)\underbrace{\Pr\left(\mathbf{Q}|o_1, o_2, \ldots, o_m\right)}_{\text{Before } O' \text{ observed}}}$$

General knowledge representation and inference: the BIG PICTURE

Simple eh?

HAH!!! No chance...

Even if all your RVs are just Boolean:

- For n RVs knowing the knowledge base $\Pr(\mathbf{V})$ means storing 2^n numbers.
- So it looks as though storage is $O(2^n)$.
- You need to establish 2^n numbers to work with.
- Look at the summations. If there are n latent variables then it appears that time complexity is also $O(2^n)$.
- In reality we might well have $n > 1000$, and of course it's *even worse* if *variables are non-Boolean*.

And it *really is this hard*. The problem in general is *#P-complete*.

Even getting an *approximate solution* is provably intractible.

General knowledge representation and inference: the BIG PICTURE

How can we get around this?

1. You can be clever about representing $\Pr(\mathbf{V})$ to avoid storing all $O(2^n)$ numbers.
2. You can take that a step further and *exploit the structure of* $\Pr(\mathbf{V})$ in specific scenarios to get good time-complexity.
3. You can do *approximate inference*.

We'll be looking at all three…

Artificial Intelligence: what are we going to learn now?

What are we going to learn now?

By addressing AI using *Bayesian Inference* in this way, in addition to general methods for making inferences:

- We get rigorous methods for *supervised learning*.
- We get one of the most *unreasonably effective* ideas in computer science: the *hidden Markov model*.
- We get methods for *unsupervised learning*.

Bayesian supervised learning provides a (potentially) *optimal* method for supervised learning.

Hidden Markov models allow us to infer (probabilistically) the *state* of the world as *time passes*.

Mixture models form the basis of probabilistic methods for *unsupervised learning*.

Artificial Intelligence: what are we going to learn now?

Putting it all together…

Ideally we want an agent to be able to:

- *Explore* the world to see how it works.
- Use the resulting knowledge to form a *plan* of how to act in the future.
- Achieve both, even when the world is *uncertain*.

In essence *reinforcement learning* algorithms allow us to do this.

In practice they often employ *supervised learners* as a subsystem.

Books

Books recommended for the course:

I suggest you make use of the recommended text for Artificial Intelligence I:

Artificial Intelligence: A Modern Approach. Stuart Russell and Peter Norvig, 3rd Edition, Pearson, 2010.

and supplement it with one of the following:

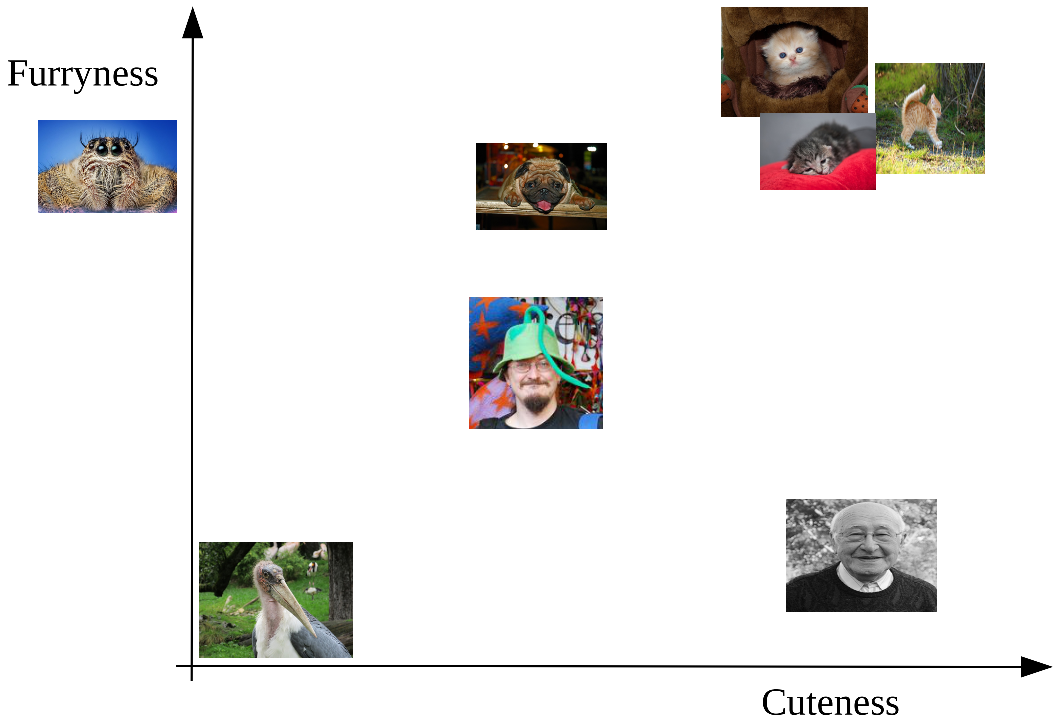
1. *Pattern Recognition and Machine Learning*. Christopher M. Bishop, Springer, 2006.
2. *Machine Learning: A Probabilistic Perspective*. Kevin P. Murphy, The MIT Press, 2012.

The latter is more comprehensive and goes beyond this course.

Further recommended books, covering specific areas in greater detail, can be found on the course web site.

What have we done so far?

We're going to begin with a review of the material on *supervised learning* from *Artificial Intelligence I*.

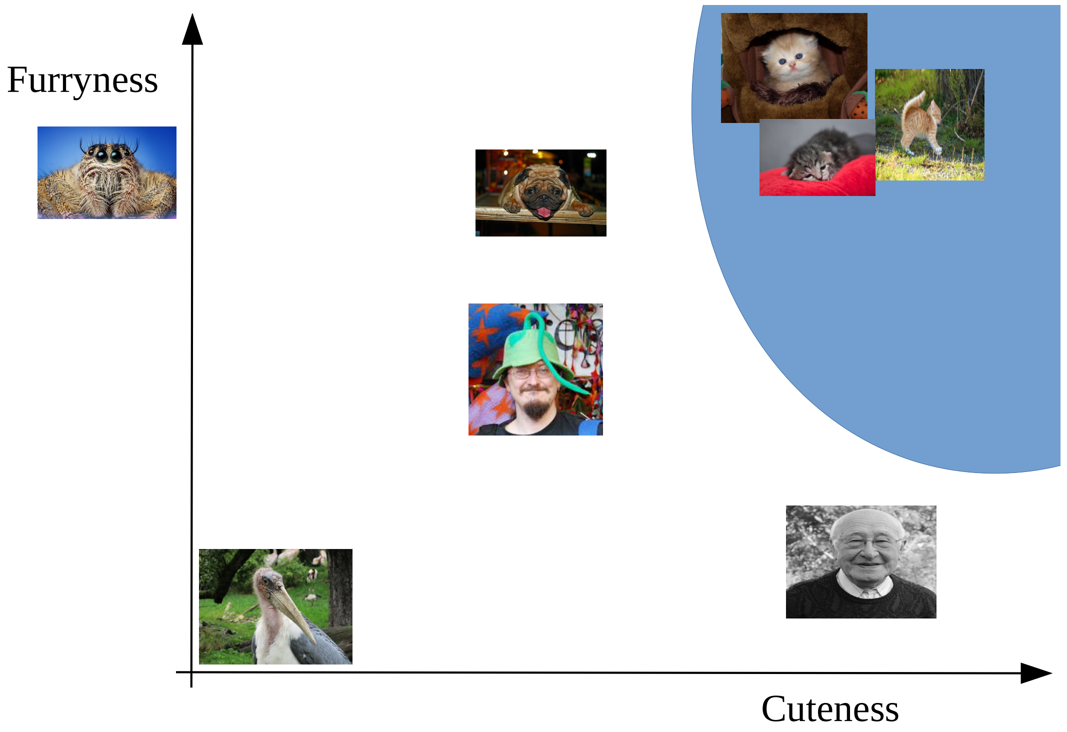


Evil Robot hates kittens, and consequently wants to build a *kitten detector*.

He thinks he can do this by measuring *cuteness* and *furryness*.

What have we done so far?

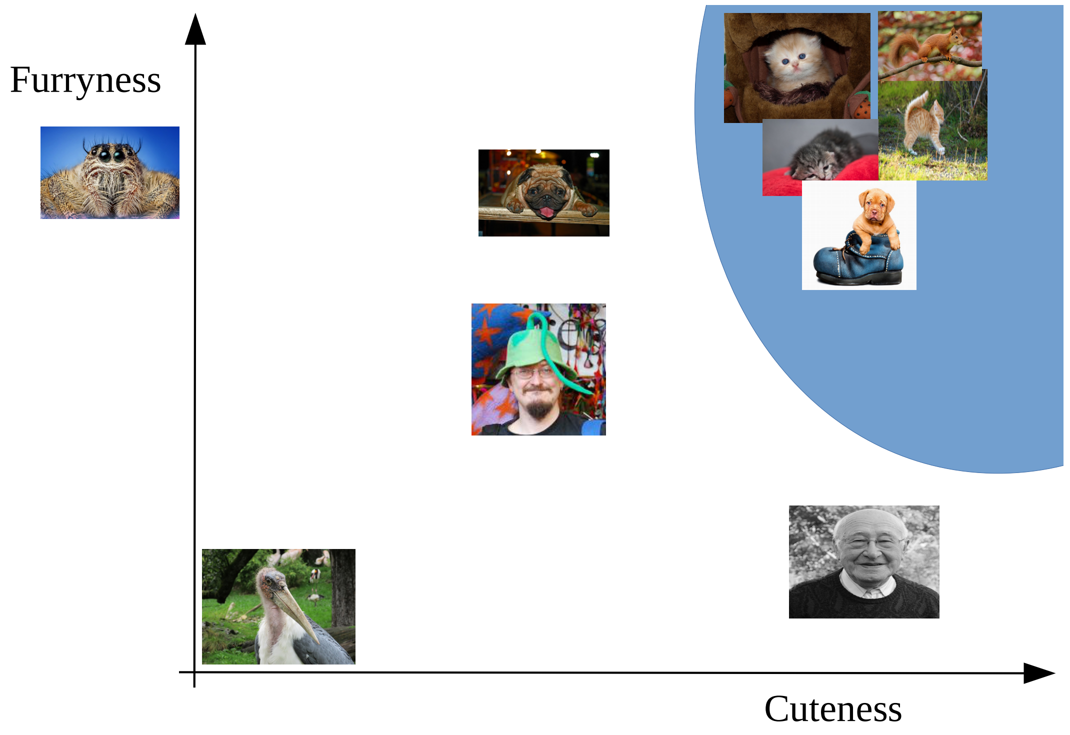
Provided he has some examples labelled as *kitten* or *not kitten*. . .



. . . this seems sufficient to find a region that identifies kittens.

What have we done so far?

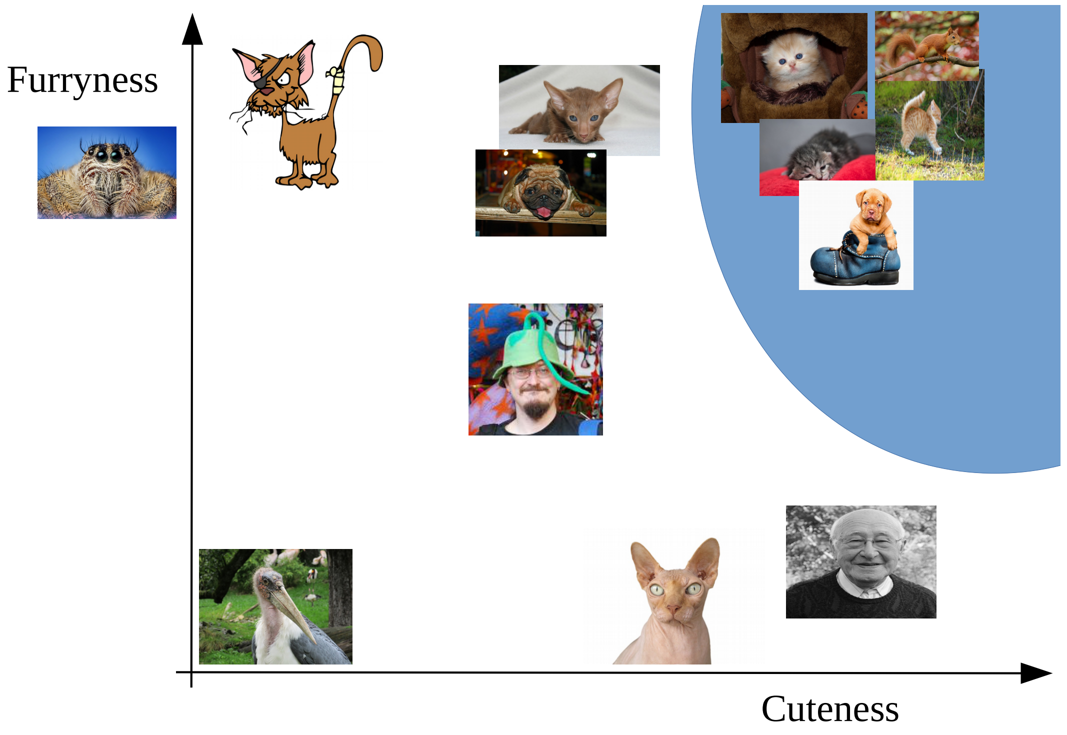
Of course, when put into practice…



…some non-kittens will be labelled as kittens.

What have we done so far?

And conversely…



…some kittens will be labelled as non-kittens.

Kinds of learning: supervised learning

Supervised learning:

We have m vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ each in $\mathbb{R}^n$.

We have corresponding *labels* $\{y_1, y_2, \ldots, y_m\}$ each in a set Y.

We wish to find a *hypothesis* $h : \mathbb{R}^n \rightarrow Y$ that can be used to predict y from $\mathbf{x}$.

This may itself be defined by a vector $\mathbf{w}$ of *weights*.

To make the latter point clear the hypothesis will be written $h_{\mathbf{w}}(\mathbf{x})$.

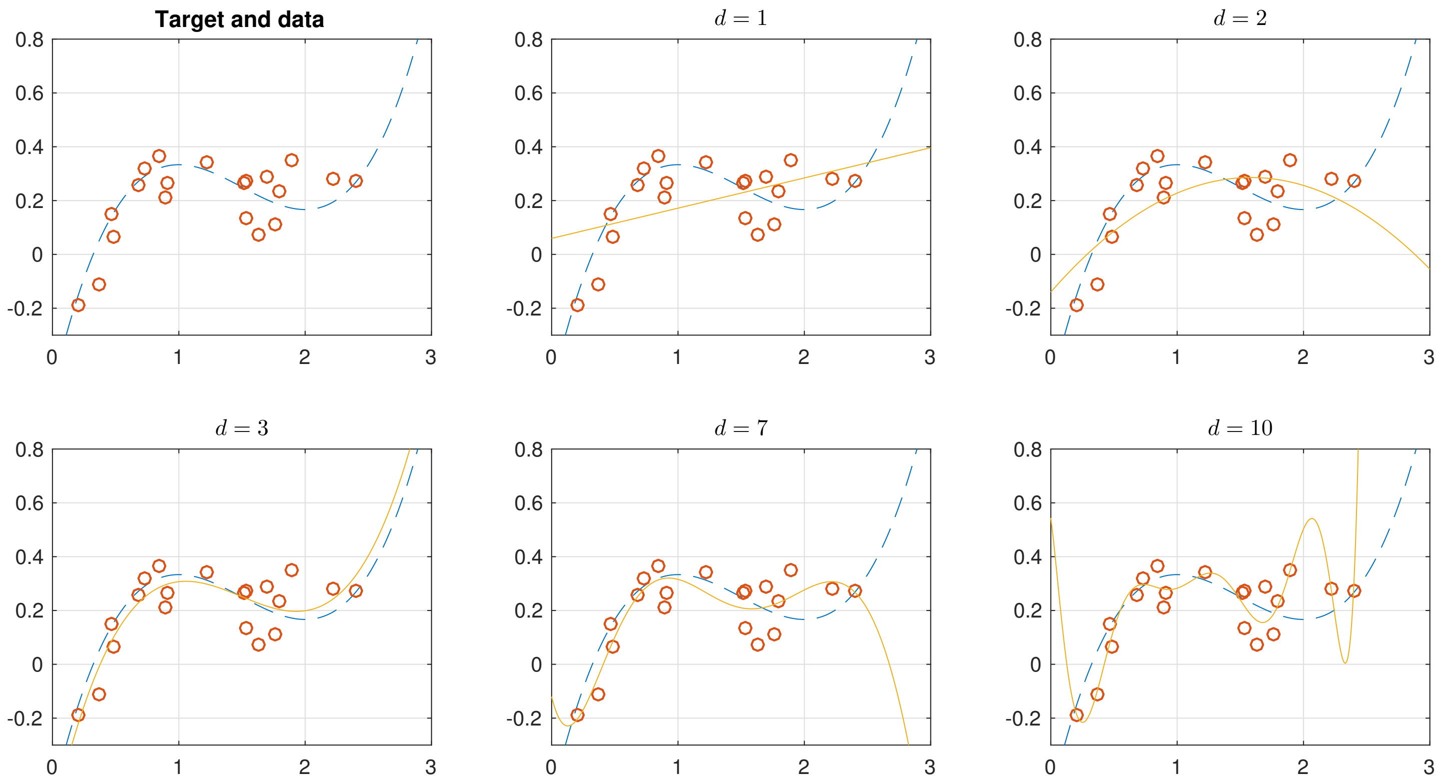
If it can do this well it *generalizes*.

- If $Y = \mathbb{R}$ or some other set such that the output can be regarded as *continuous* then we're doing *regression*.
- If Y has a finite number K of categories, so $Y = \{c_1, c_2, \ldots, c_K\}$ then we are doing *classification*.
- In the case of classification, we might alternatively treat Y as a random variable (RV), and find a *hypothesis* $h_{\mathbf{w}} : \mathbb{R}^n \rightarrow [0, 1]$ of the form

$$h_{\mathbf{w}}(\mathbf{x}) = \Pr\left(Y = c_i | \mathbf{x}\right).$$

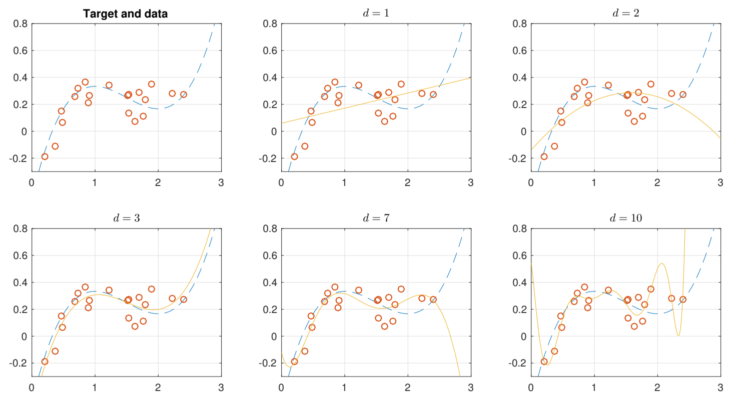
What have we done so far?

Supervised learning is essentially curve fitting:



The *key issue* is to choose the correct degree of *complexity*.

What have we done so far?



The *training data* is $\mathbf{s} = \left[(x_1, y_1) \;\; (x_2, y_2) \;\; \cdots \;\; (x_m, y_m)\right].$

Fit a polynomial

$$h_{\mathbf{w}}(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_d x^d$$

by choosing the ***weights*** w_i to minimize

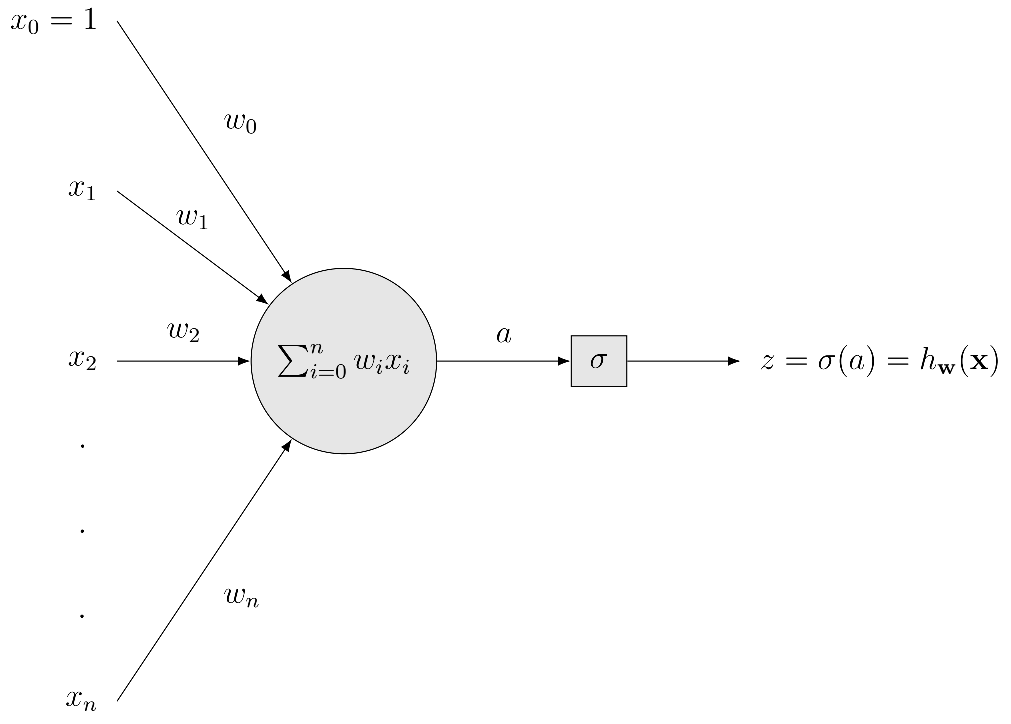
$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(x_i))^2.$$

The degree d sets how *complex* the fitted function can be.

What have we done so far?

Real problems tend to have more than 1 input.

We can solve problems like this using a *perceptron*:



The trick is the same: select the *weights* w_i to minimize some measure of *error* $E(\mathbf{w})$ on some *training examples*.

What have we done so far?

If we use a very simple function $\sigma(x) = x$ then we're back to polynomials with $d = 1$ and now

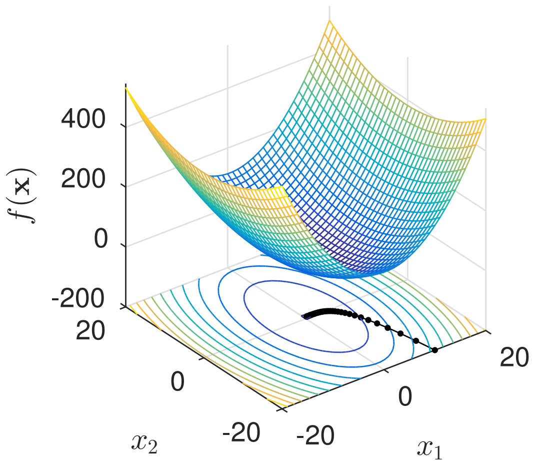
$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

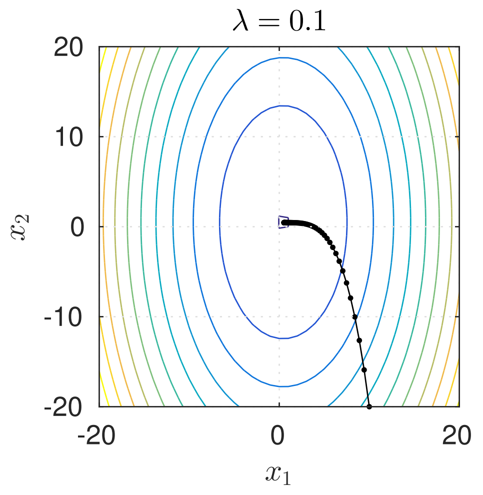
If we can find the *gradient* $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ of $E(\mathbf{w})$ then we can minimize the error using *gradient descent*

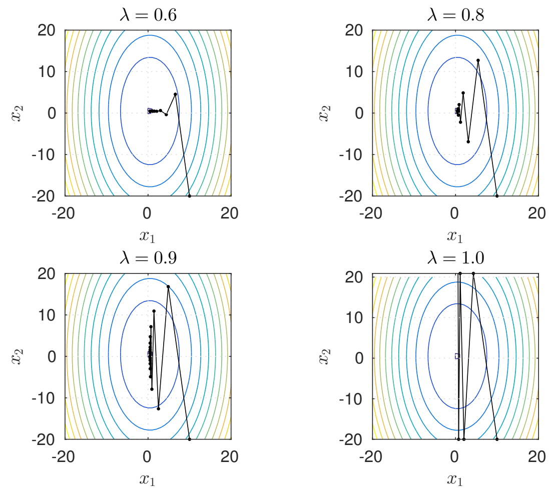
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}_t}$$

What have we done so far?

Gradient descent: the *simplest possible method* for minimizing such functions:







Take *small steps downhill* until you reach the minimum.

But remember: *there might be many minima*.

Some minima might be *local* and some *global*.

The *step size* matters.

What have we done so far?

For a perceptron with $\sigma(x) = (x)$ this is easy:

$$
\begin{aligned}
\frac{\partial E(\mathbf{w})}{\partial w_j} &= \frac{1}{2}\frac{\partial}{\partial w_j}\left(\sum_{i=1}^{m}(y_i - \mathbf{w}^T\mathbf{x}_i)^2\right) \\
&= \sum_{i=1}^{m}\left((y_i - \mathbf{w}^T\mathbf{x}_i)\frac{\partial}{\partial w_j}(-\mathbf{w}^T\mathbf{x}_i)\right) \\
&= -\sum_{i=1}^{m}(y_i - \mathbf{w}^T\mathbf{x}_i)\,\mathbf{x}_i^{(j)}
\end{aligned}
$$

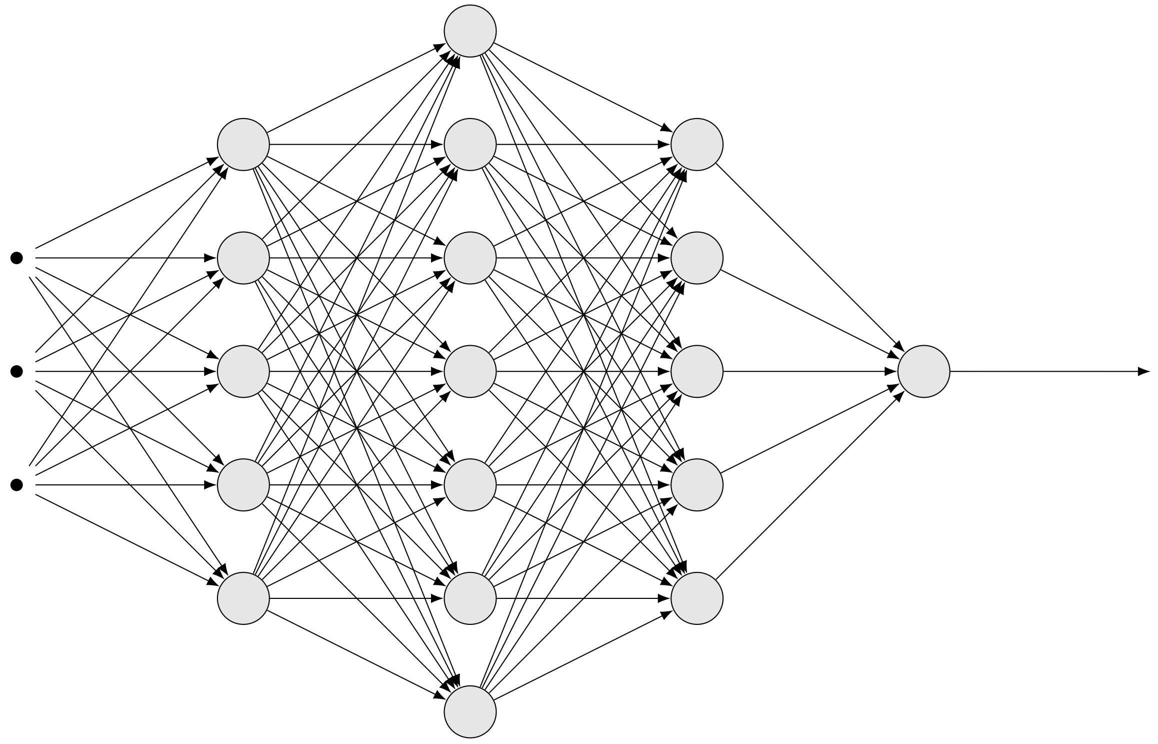
where $\mathbf{x}_i^{(j)}$ is the jth element of $\mathbf{x}_i$. So:

$$
\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = -\sum_{i=1}^{m}(y_i - \mathbf{w}^T\mathbf{x}_i)\mathbf{x}_i
$$

The multilayer perceptron

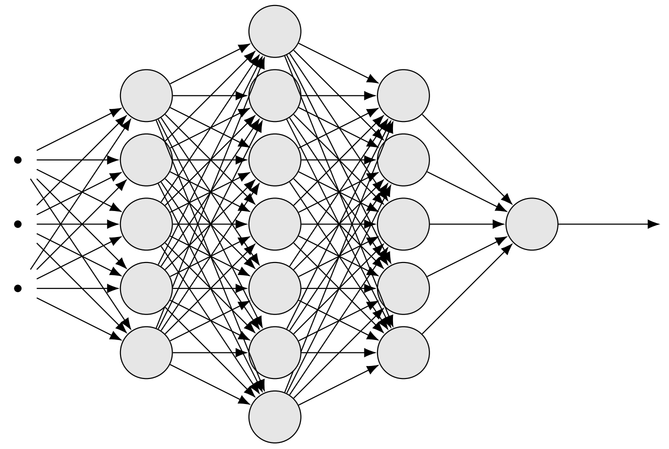
Real problems tend also to be *nonlinear*.

We can combine perceptrons to make a *multilayer perceptron*:



Here, each *node* is a perceptron and each *edge* has a weight attached.

The multilayer perceptron



- The network computes a function $h_{\mathbf{w}}(\mathbf{x})$.
- The trick remains the same: minimize an error $E(\mathbf{w})$.
- We do that by *gradient descent*

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}_t}$$

- This can be achieved using *backpropagation*.
- Backpropagation is *just* a method for computing $\partial E(\mathbf{w})/\partial \mathbf{w}$.

Backpropagation

I want to emphasize the last three statements:

Backpropagation is *just* a method for computing $\partial E(\mathbf{w})/\partial \mathbf{w}$.

It's needed because we're doing *gradient descent*

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}_t}$$

In supervised learning, you can get quite a long way using a multilayer perceptron.

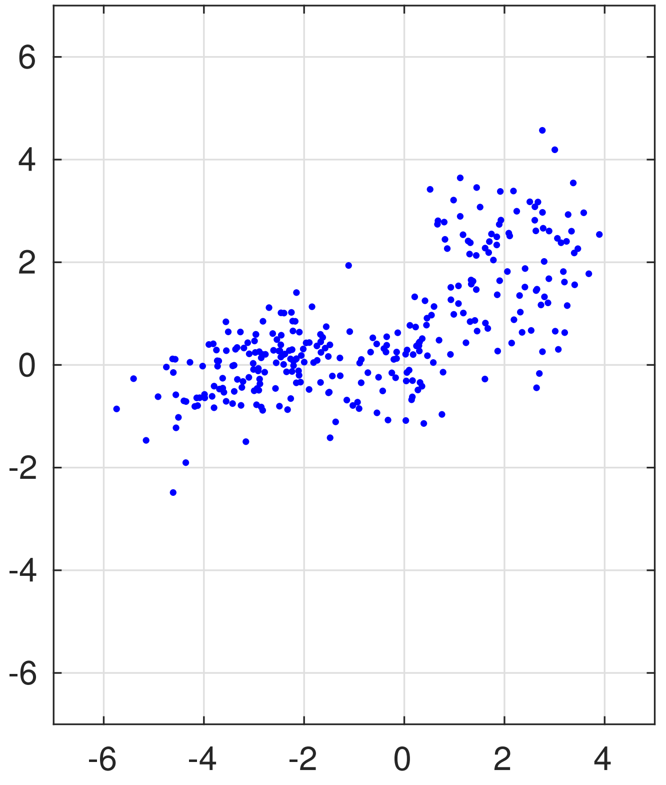
If you understand backpropagation, you already know the key idea needed for *stuff involving the word 'deep'*.

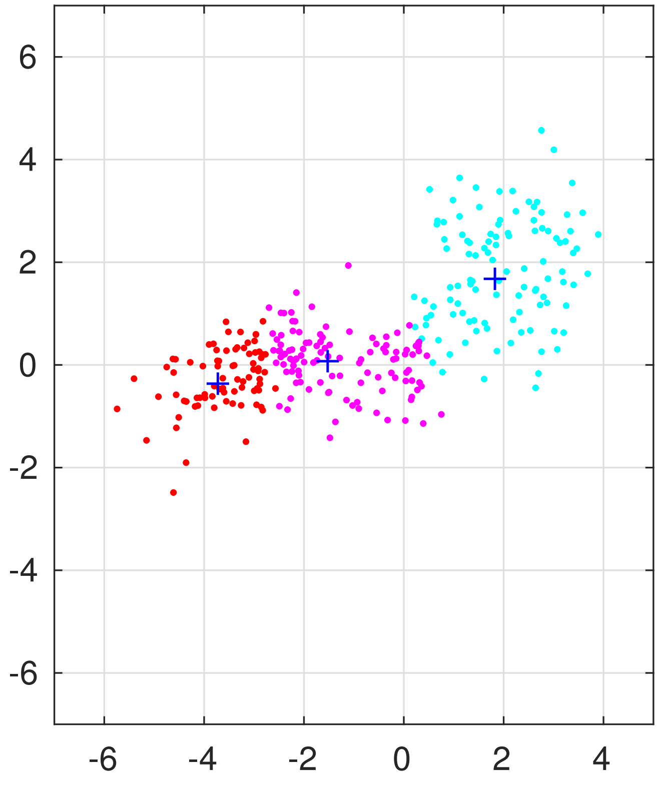
But this is a long way from being the *full story*.

Kinds of learning: unsupervised learning

What if we have *no labels*?

Unsupervised learning: we have m vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ each in $\mathbb{R}^n$ …



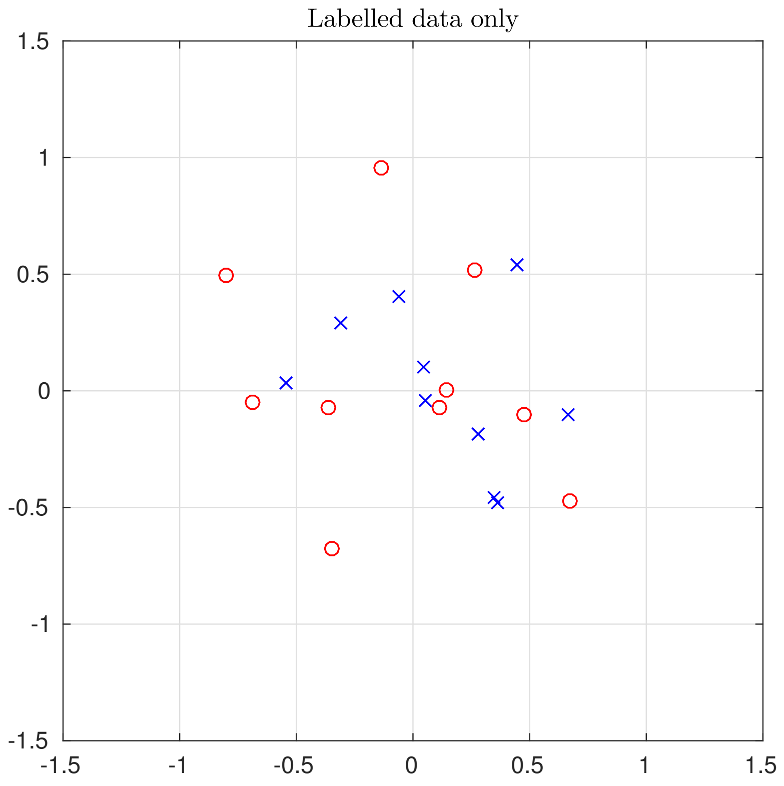


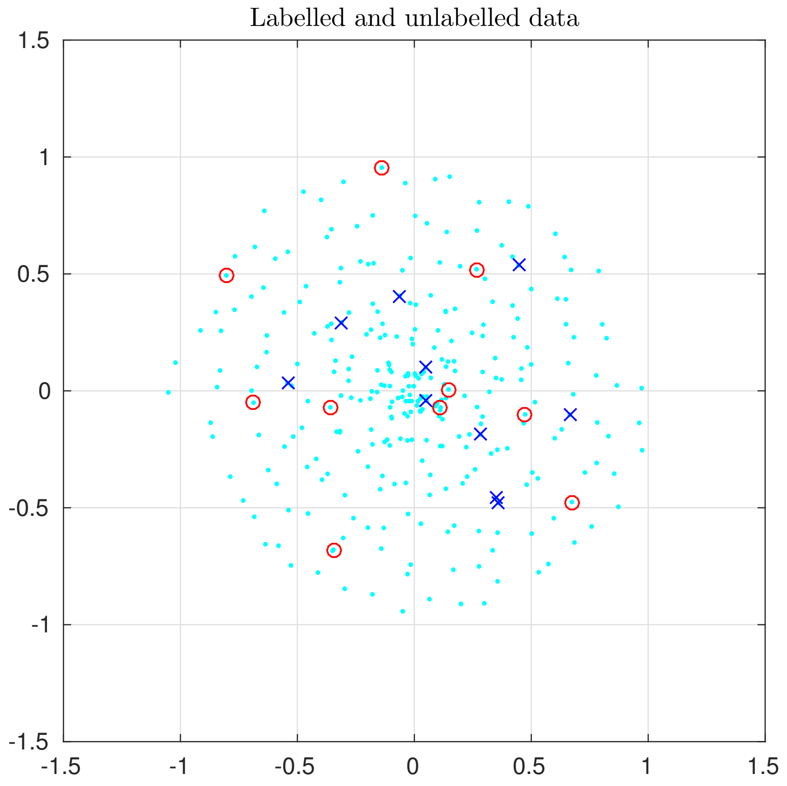
…and we want to find some *regularity*.

Kinds of learning: semi-supervised learning

Semi-supervised learning: we have the same labelled data as for supervised learning, but…

…*in addition* a further m' input vectors $\mathbf{x}'_1, \ldots, \mathbf{x}'_{m'}$.





We want to use the extra information to *improve the hypothesis obtained*.

Kinds of learning: reinforcement learning

What if we want to learn from *rewards* rather than *labels*?

Reinforcement learning works as follows.

1. We are in a *state* and can perform an *action*.
2. When an *action* is performed we move to a *new state* and receive a *reward*. (Possibly *zero* or *negative*.)
3. New states and rewards can be *uncertain*.
4. We have *no knowledge in advance* of how actions affect either the new state or the reward.
5. We want to learn a *policy*. This tells us what action to perform in any state.
6. We want to learn a policy that in some sense *maximizes reward obtained over time*.

Note that this can be regarded as a form of *planning*.

Matrix notation

We denote by $\mathbb{R}^n$ the set of n-dimensional *vectors of reals*, and by the set $\mathbb{R}^{m\times n}$ the set of m (rows) by n (columns) *matrices of reals*.

Vectors are denoted using *lower-case bold* and matrices in *upper-case bold*.

It is conventional to assume that vectors are *column vectors* and to denote the *transpose* using superscripted T. So for $\mathbf{x} \in \mathbb{R}^n$ we write

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

and for $\mathbf{X} \in \mathbb{R}^{m\times n}$ we write

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

Denote by $\mathbf{X}_{i\star}$ and $\mathbf{X}_{\star j}$ the ith *row* and jth *column* of $\mathbf{X}$ respectively.

Matrix notation

If we have m vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ then the jth element of the ith vector is $\mathbf{x}_i^{(j)}$.
We may also form the matrix

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_m^T \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^{(1)} & \mathbf{x}_1^{(2)} & \cdots & \mathbf{x}_1^{(n)} \\ \mathbf{x}_2^{(1)} & \mathbf{x}_2^{(2)} & \cdots & \mathbf{x}_2^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_m^{(1)} & \mathbf{x}_m^{(2)} & \cdots & \mathbf{x}_m^{(n)} \end{bmatrix}$$

Similarly we can write

$$\mathbf{X}^T = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_m \end{bmatrix}$$

The *identity matrix* is as usual

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

The *inverse* of $\mathbf{X}$ is $\mathbf{X}^{-1}$ and its *determinant* is $|\mathbf{X}|$.

General notation

An RV can take on one of a *set* of values. For example, X is an RV with values $\{x_1, x_2, \ldots, x_n\}$.

By convention *random variables (RVs)* are denoted using *upper-case* and their *values* using *lower-case*.

The probability that X takes a *specific* value $x \in \{x_1, x_2, \ldots, x_n\}$ is $\Pr(X = x)$. This will generally be abbreviated to just $\Pr(x)$

Sometimes we need to sum over all possible values. We write this using the usual notation. So for example the *expected value* of X is

$$\mathbb{E}[X] = \sum_{x \in X} x \Pr(x) = \sum_{X} X \Pr(X).$$

We extend this to *vector-valued* RVs in the obvious way.

So for example we might define an RV $\mathbf{X}$ taking values in $\mathbb{R}^n$ and refer to a specific value $\mathbf{x} \in \mathbb{R}^n$.

(But remember: asking about something like $\Pr(\mathbf{X} = \mathbf{x})$ now makes little sense if $\mathbf{x} \in \mathbb{R}^n$.)

General notation for supervised learning

- *Inputs* are in n dimensions and are denoted by

$$\mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

  Each element x_i is a *feature*.

- A training sequence has m elements. The m inputs are $\mathbf{x}_1, \ldots, \mathbf{x}_m$ and can be collected into the matrix

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_m^T \end{bmatrix}$$

- The *labels* in the training sequence are denoted by

$$\mathbf{y}^T = \begin{bmatrix} y_1 & y_2 & \cdots y_m \end{bmatrix}$$

  with each y_i in a set Y depending on the type of problem.

General notation for supervised learning

- For *regression problems* we have $Y = \mathbb{R}$.
- For *classification problems* with *two classes* we have $Y = \mathbb{B}$.
- For two classes it is sometimes convenient to use labels $\{+1, -1\}$ and sometimes $\{0, 1\}$. We shall therefore denote these sets by $\mathbb{B}$ and rely on the context.
- For *classification problems* with $K > 2$ *classes* we have $Y = \{c_1, \ldots, c_K\}$.

Inputs and labels are collected together and written

$$\mathbf{s}^T = \left[(\mathbf{x}_1, y_1) \;\; (\mathbf{x}_2, y_2) \;\; \ldots \;\; (\mathbf{x}_m, y_m) \right].$$

This is the *training sequence*.

Machine Learning and Bayesian Inference

Major subject number one:

Making learning *probabilistic*.

It will turn out that in order to talk about *optimal* methods for machine learning we'll have to put it into a probabilistic context.

As a bonus, this leads to a much better understanding of what happens when we *choose weights by minimizing an error function*.

And it turns out that choosing weights in this way is *suboptimal*. . .

. . . although, intriguingly, that's not a reason not to do it.

Probabilistic models for generating data

I'm going to start with a *very simple*, but *very informative* approach.

Typically, we can think of individual examples as being generated according to some distribution $p(\mathbf{X}, Y)$.

We generally make the simplifying assumption that examples are *independent and identically distributed (iid)*. Thus the training data

$$\mathbf{s}^T = \left[(\mathbf{x}_1, y_1) \;\; (\mathbf{x}_2, y_2) \;\; \cdots \;\; (\mathbf{x}_m, y_m)\right]$$

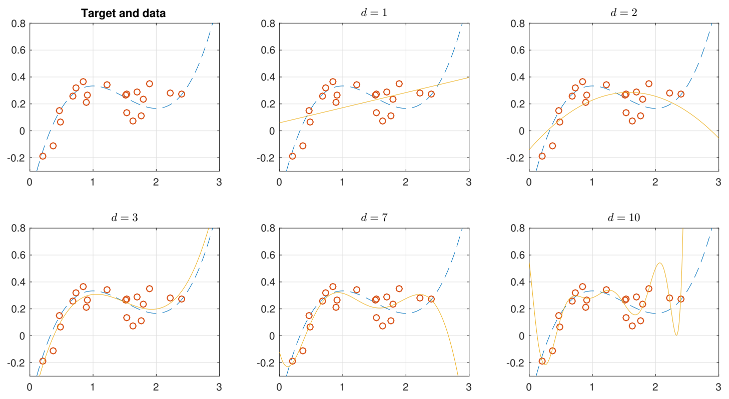
represents m iid samples from the relevant distribution.

As the examples are iid we can write

$$p(\mathbf{s}) = \prod_{i=1}^{m} p(\mathbf{x}_i, y_i).$$

Example: simple regression

Here's how I generated the regression data for the initial examples:



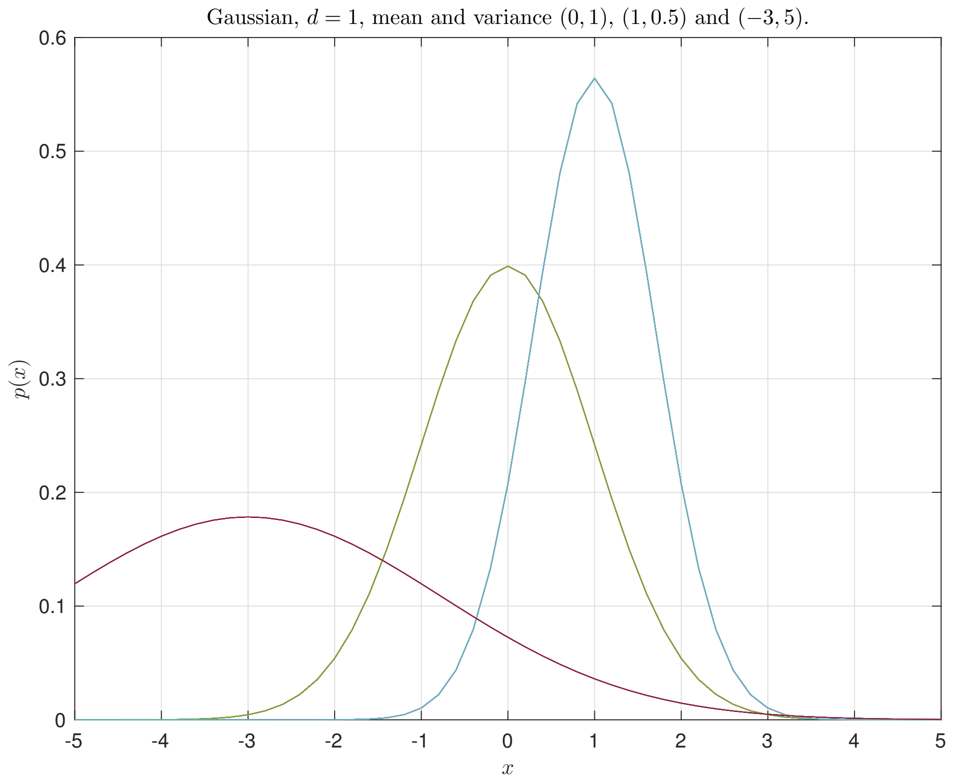
We have spoken of an *unknown underlying function* f used to generate the data. In fact, this is the *hypothesis* $h_{\mathbf{w}}$ that we *want to identify* by choosing $\mathbf{w}$.

I chose $h_{\mathbf{w}}$ to be a polynomial with parameters $\mathbf{w}$ — this is the dashed blue line.

So in fact the unknown function is $h_{\mathbf{w}}(\mathbf{x})$, emphasizing that $\mathbf{w}$ *determines a specific function* f.

Remember: you don't know what $\mathbf{w}$ is: *you need to identify it by analysing* $\mathbf{s}$.

The Normal Distribution



In 1 dimension $\mathcal{N}(\mu, \sigma^2)$ is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

with mean μ and variance σ^2.

Example: simple regression

To make $\mathbf{s}$:

For the ith example:

1. I sampled $\mathbf{x}_i$ according to the *uniform density* on $[0, 3]$. So there is a distribution $p(\mathbf{x})$.
2. I computed the value $h_{\mathbf{w}}(\mathbf{x}_i)$.
3. I sampled $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ with $\sigma^2 = 0.1$ and formed $y_i = h_{\mathbf{w}}(\mathbf{x}_i) + \epsilon_i$.

Combining steps 2 and 3 gives you $p(y_i|\mathbf{x}_i, \mathbf{w})$.

$$\begin{aligned} p(y_i|\mathbf{x}_i, \mathbf{w}) &= \mathcal{N}(h_{\mathbf{w}}(\mathbf{x}_i), \sigma^2) \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2\right). \end{aligned}$$

The likelihood function

The quantity $p(y_i|\mathbf{x}_i, \mathbf{w})$ is important: it is known as the *likelihood*.

You will sometimes see it re-arranged and written as the *likelihood function*

$$L(\mathbf{w}|\mathbf{x}_i, y_i) = p(y_i|\mathbf{x}_i, \mathbf{w}).$$

Note that its form depends on how you model the data. There are *different likelihood functions depending on what assumptions you make*.

Now let's image $\mathbf{w}$ is *fixed* (but hidden!) from the outset and extend the likelihood to the whole data set $\mathbf{s}$...

The likelihood function

The *likelihood* for the full data set is:

$$\begin{aligned} p(\mathbf{s}|\mathbf{w}) &= \prod_{i=1}^{m} p(\mathbf{x}_i, y_i|\mathbf{w}) \\ &= \prod_{i=1}^{m} p(y_i|\mathbf{x}_i, \mathbf{w}) p(\mathbf{x}_i|\mathbf{w}) \\ &= \prod_{i=1}^{m} p(y_i|\mathbf{x}_i, \mathbf{w}) p(\mathbf{x}_i) \end{aligned}$$

The last step involves the reasonable assumption that $\mathbf{x}_i$ itself never depends on $\mathbf{w}$.

Maximizing likelihood

This expression, roughly translated, tells us *how probable the data* $\mathbf{s}$ would be *if a particular vector* $\mathbf{w}$ had been used to generate it.

This immediately suggests a way of choosing $\mathbf{w}$:

Choose

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w}}{\operatorname{argmax}}\, p(\mathbf{s}|\mathbf{w}).$$

This is called (surprise surprise) a *maximum likelihood* algorithm.

How would we solve this maximization problem?

Maximizing likelihood

This is surprisingly easy:

$$\begin{aligned}
\mathbf{w}_{\text{opt}} &= \operatorname*{argmax}_{\mathbf{w}} p(\mathbf{s}|\mathbf{w}) \\
&= \operatorname*{argmax}_{\mathbf{w}} \left(\prod_{i=1}^{m} p(y_i|\mathbf{x}_i, \mathbf{w}) p(\mathbf{x}_i) \right) \\
&= \operatorname*{argmax}_{\mathbf{w}} \left(\sum_{i=1}^{m} \log p(y_i|\mathbf{x}_i, \mathbf{w}) + \sum_{i=1}^{m} \log p(\mathbf{x}_i) \right) \\
&= \operatorname*{argmax}_{\mathbf{w}} \sum_{i=1}^{m} \log p(y_i|\mathbf{x}_i, \mathbf{w})
\end{aligned}$$

We've used three standard tricks:

1. To maximize something *you can alternatively maximize its logarithm*.
2. Logarithms *turn products into sums*.
3. You can drop parts of the expression *that don't depend on the variable you're maximizing over*

Maximizing likelihood

Then:

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w}}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{i=1}^{m} \left(\frac{1}{2\sigma^2} (y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2 \right) \right]$$

$$= \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2$$

So we've just shown that:

To choose $\mathbf{w}$ by maximizing likelihood…

…we minimize the sum of squared errors.

Result!

Maximizing likelihood

It’s worth reflecting on that for a moment:

- Originally, we plucked

  $$E(\mathbf{w}) = \sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2$$

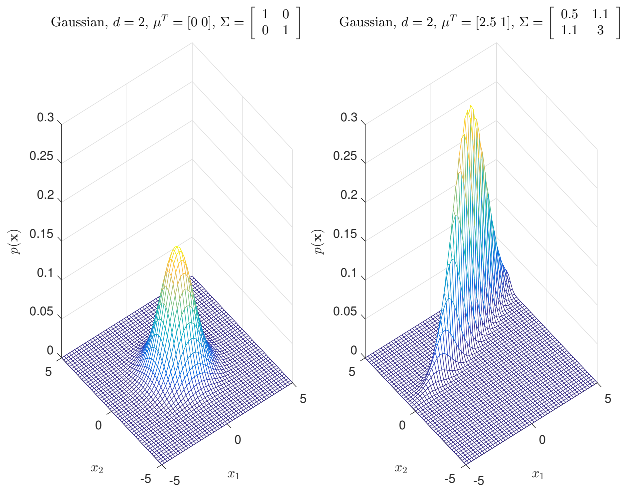
  pretty much *out of thin air* because it seemed to make sense.
- We’ve just shown that hidden inside it is an *assumption*: that *noise* in the data is *Gaussian*.
- We’ve also uncovered a *second assumption*: that maximizing the likelihood is the *right thing to do*.

Of course, assumptions such as these are open to question. . .

Maximizing the posterior

For example, what if we don't regard $\mathbf{w}$ as being *fixed in advance* but instead make it an RV as well?

That means *we need a distribution* $p(\mathbf{w})$, generally known as the *prior* on $\mathbf{w}$. How about our old friend the normal? In d dimensions $\mathbf{w} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ looks like



$$p(\mathbf{w}) = \frac{1}{\sqrt{|\boldsymbol{\Sigma}|(2\pi)^d}} \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{w} - \boldsymbol{\mu})\right)$$

with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$.

Maximizing the posterior

This suggests another natural algorithm for choosing a good $\mathbf{w}$, called the *maximum a posteriori (MAP) algorithm*. Let's choose $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \lambda^{-1}\mathbf{I})$ so

$$p(\mathbf{w}) = \frac{1}{\sqrt{\lambda^{-d}(2\pi)^d}} \exp\left(-\frac{\lambda}{2}\mathbf{w}^T\mathbf{w}\right)$$

Then

$$\begin{aligned}
\mathbf{w}_{\text{opt}} &= \operatorname*{argmax}_{\mathbf{w}} p(\mathbf{w}|\mathbf{s}) \\
&= \operatorname*{argmax}_{\mathbf{w}} \frac{p(\mathbf{s}|\mathbf{w})p(\mathbf{w})}{p(\mathbf{s})} \\
&= \operatorname*{argmax}_{\mathbf{w}} \left[\log p(\mathbf{s}|\mathbf{w}) + \log p(\mathbf{w})\right]
\end{aligned}$$

The maximization of $\log p(\mathbf{s}|\mathbf{w})$ proceeds as before, and we end up with

$$\boxed{\mathbf{w}_{\text{opt}} = \operatorname*{argmin}_{\mathbf{w}} \left[\frac{1}{2}\sum_{i=1}^{m}\left((y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2\right) + \frac{\lambda}{2}||\mathbf{w}||^2\right].}$$

Maximizing the posterior

This appears in the literature under names such as *weight decay*.

- It was often proposed, again on the basis that it seemed sensible, as a *sensible-looking way of controlling the complexity of* $h_{\mathbf{w}}$.
- The idea was to use λ to achieve this.
- We'll be seeing later how to do this.

Once again, we can now see that it *hides certain assumptions*.

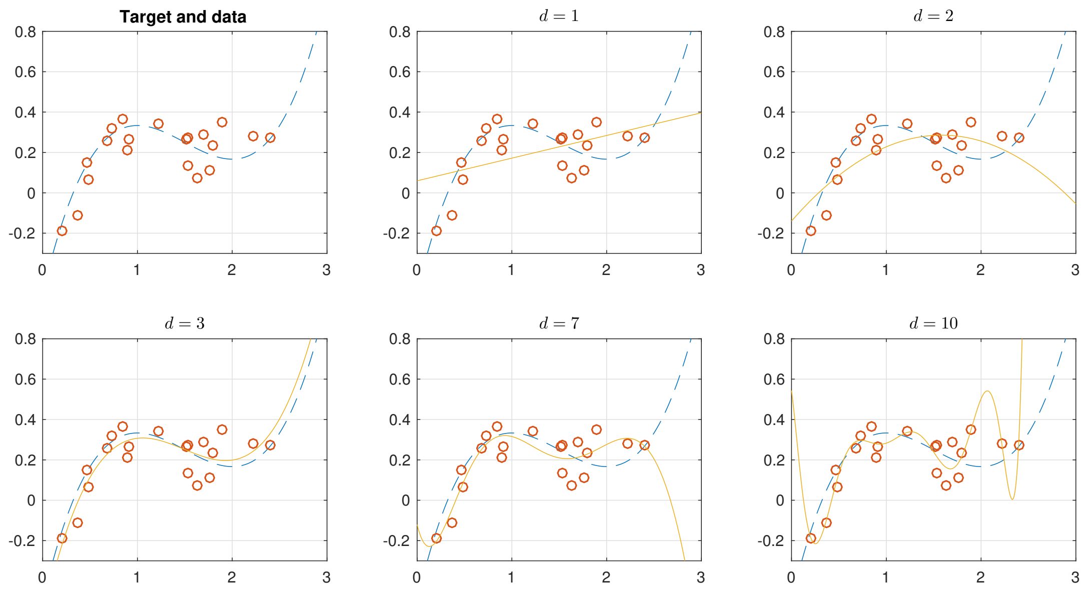
In addition to the assumptions made by maximum likelihood:

- We are assuming that *some kinds of* $\mathbf{w}$ are more likely than others.
- We are assuming that *the distribution governing this is Gaussian*.

And again, these assumptions *may or may not be appropriate*.

The likelihood for classification problems

For *regression problems* just adding noise to the labels seems reasonable:



The likelihood $p(y|\mathbf{x}, \mathbf{w})$ is in fact a *density* and can take any value in $\mathbb{R}$ as long as the density is non-negative and integrates to 1.

(Think of the Gaussian as usual…).

But what about for *classification problems*?

The likelihood for classification problems

For simplicity, let's just consider *two-class classification* with labels in $\{0, 1\}$.

For a *classification problem* the *likelihood* is now a *distribution* $\Pr(Y|\mathbf{x}, \mathbf{w})$. It has two non-negative values, and

$$\Pr(Y = 1|\mathbf{x}, \mathbf{w}) = 1 - \Pr(Y = 0|\mathbf{x}, \mathbf{w}).$$

So *you can't just add noise to the underlying* $h_{\mathbf{w}}$.

Fix: define the likelihood as

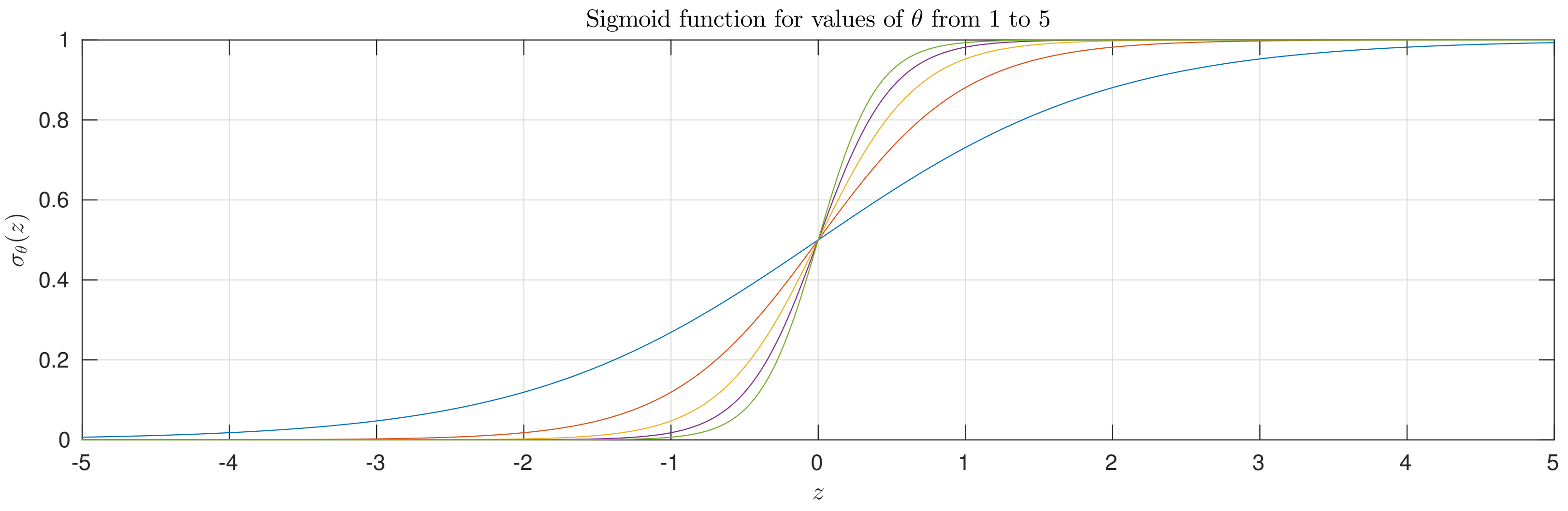
$$\Pr(Y = 1|\mathbf{x}, \mathbf{w}) = \sigma_\theta(h_{\mathbf{w}}(\mathbf{x}))$$

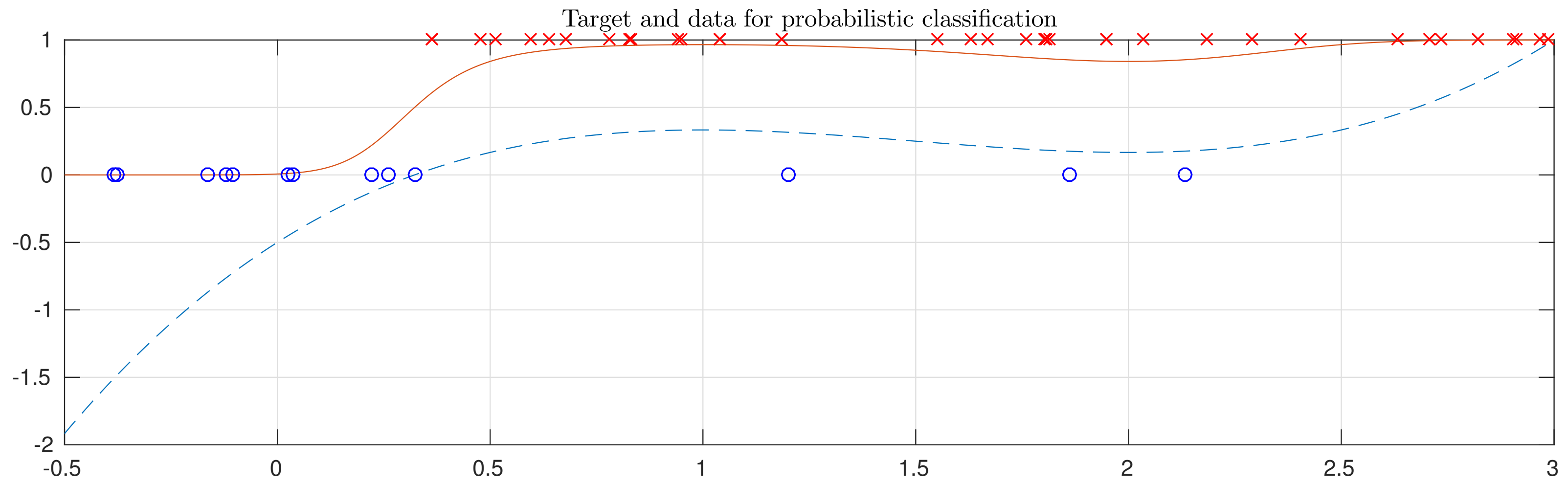
and use something like

$$\sigma_\theta(z) = \frac{1}{1 + \exp(-\theta z)}$$

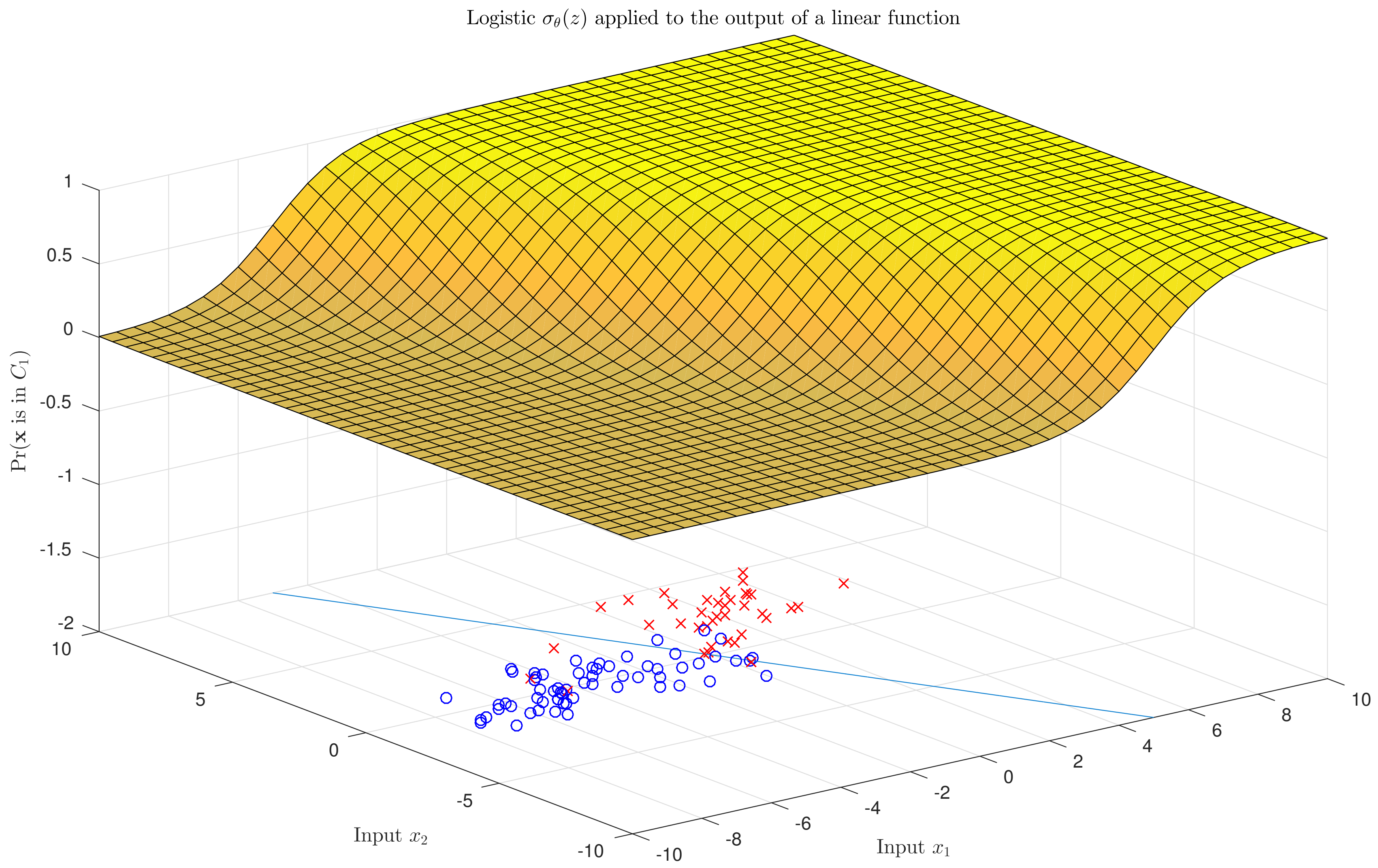
to impose the above property.

The likelihood for classification problems





The likelihood for classification problems



The likelihood for classification problems

So: if we're given a training sequence $\mathbf{s}$, *what is the probability that it was generated using some* $\mathbf{w}$?

For an example $(\mathbf{x}, y)$

$$\Pr\left(Y|\mathbf{x},\mathbf{w}\right) = \begin{cases} \sigma_\theta(h_\mathbf{w}(\mathbf{x})) & \text{if } Y = 1 \\ 1 - \sigma_\theta(h_\mathbf{w}(\mathbf{x})) & \text{if } Y = 0 \end{cases}$$

Consequently *when* Y *has a known value* we can write

$$\Pr\left(Y|\mathbf{x},\mathbf{w}\right) = \left[\sigma_\theta(h_\mathbf{w}(\mathbf{x}))\right]^Y \left[1 - \sigma_\theta(h_\mathbf{w}(\mathbf{x}))\right]^{(1-Y)}$$

If we assume that the examples are iid then the probability of seeing the labels in a training sequence $\mathbf{s}$ is straightforward.

The likelihood for classification problems

The likelihood is now

$$\begin{aligned} p(\mathbf{s}|\mathbf{w}) &= \prod_{i=1}^{m} p(y_i|\mathbf{x}_i, \mathbf{w})p(\mathbf{x}_i) \\ &= \prod_{i=1}^{m} \left[\sigma_\theta(h_{\mathbf{w}}(\mathbf{x}_i))\right]^{y_i} \left[1 - \sigma_\theta(h_{\mathbf{w}}(\mathbf{x}_i))\right]^{(1-y_i)} p(\mathbf{x}_i) \end{aligned}$$

where the first line comes straight from an earlier slide.

Note that:

- Whereas previously we had the *noise variance* σ^2 we now have the parameter θ. Both serve a *similar purpose*.
- From this expression we can directly derive *maximum-likelihood* and *MAP* learning algorithms for classifiers.

The next step...

We have so far concentrated throughout our coverage of machine learning on choosing a *single hypothesis*.

Are we asking the right question though?

Ultimately, *we want to generalise.*

This means finding a hypothesis that *works well for previously unseen examples.*

That means we have to *define what good generalization is* and *ask what method might do it the best.*

Is it reasonable to expect a *single hypothesis* to provide the optimal answer?

We need to look at what the optimal solution to this kind of problem might be...

Bayesian decision theory

What is the *optimal* approach to this problem?

Put another way: how should we make decisions in such a way that the outcome obtained is, on average, the best possible? Say we have:

- Attribute vectors $\mathbf{x} \in \mathbb{R}^d$.
- A set of *K classes* $\{c_1, \ldots, c_K\}$.
- A set of *L actions* $\{\alpha_1, \ldots, \alpha_L\}$.

There is essentially nothing new here.

The actions can be thought of as saying *'assign $\mathbf{x}$ to class c_1'* and so on. We may have further actions, for example the action *'I don't know how to classify $\mathbf{x}$'*.

There is also a *loss* λ_{ij} associated with taking action a_i when the class is in fact c_j.

Sometimes we will need to write $\lambda(a_i, c_j)$ for λ_{ij}.

Bayesian decision theory

The ability to specifiy losses in this way can be important, For example:

- In learning to *diagnose cancer* we might always assign a loss of 0 when the action is *'say the patient has cancer'*, assuming the patient does in fact have cancer.
- A loss of 0 is also appropriate if we take action *'say the patient is healthy'* when the patient actually is healthy.
- The subtlety appears when our action is *wrong*. We should probably assign a bigger penalty (higher loss) if we *tell a patient they are heathy when they're sick*, than if we *tell a patient they're sick when they're healthy*.

Having extra actions can also be useful.

Also, sometimes we want the system to *defer to a human*.

Bayesian decision theory

Say we can further *model the world* as follows:

- Classes have probabilities $\Pr(C)$ of occurring.
- There are probability densities $p(\mathbf{X}|C)$ for seeing $\mathbf{X}$ when the class is C.

So now we have a *slightly different, though equivalent* way of modelling how labelled examples are generated: nature *chooses classes at random* using $\Pr(C)$ and *selects a vector* using $p(\mathbf{X}|C)$.

$$p(\mathbf{X}, C) = \underbrace{p(\mathbf{X}|C)\Pr(C)}_{\text{current model}} = \underbrace{\Pr(C|\mathbf{X})\, p(\mathbf{X})}_{\text{previous model}}$$

As usual Bayes rule tells us that

$$\Pr(C|\mathbf{X}) = \frac{1}{Z} p(\mathbf{X}|C)\Pr(C)$$

where

$$Z = p(\mathbf{X}) = \sum_{i=1}^{K} p(\mathbf{X}|c_i)\Pr(c_i)\,.$$

Bayesian decision theory

Say *nature shows us* $\mathbf{x}$ and we *take action* a_i.

If we *always* take action a_i when we see $\mathbf{x}$ then the *average loss on seeing* $\mathbf{x}$ is

$$R(a_i|\mathbf{x}) = \mathbb{E}_{c\sim p(C|\mathbf{x})}\left[\lambda_{ij}|\mathbf{x}\right] = \sum_{j=1}^{K} \lambda_{ij}\text{Pr}\left(c_j|\mathbf{x}\right).$$

The quantity $R(a_i|\mathbf{x})$ is called the *conditional risk*.

Note that this particular $\mathbf{x}$ is *fixed*.

Bayesian decision theory

Now say we have a *decision rule* $D : \mathbb{R}^d \rightarrow \{a_1, \ldots, a_L\}$ telling us *what action to take on seeing any* $\mathbf{x} \in \mathbb{R}^d$.

The average loss, or *risk*, is

$$
\begin{aligned}
R &= \mathbb{E}_{(\mathbf{x},c)\sim p(\mathbf{X},C)}\left[\lambda(D(\mathbf{x}), c)\right] \\
&= \mathbb{E}_{\mathbf{x}\sim p(\mathbf{X})}\left[\mathbb{E}_{c\sim \Pr(C|\mathbf{x})}\left[\lambda(D(\mathbf{x}), c)|\mathbf{x}\right]\right] \\
&= \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}\left[R(D(\mathbf{x})|\mathbf{x})\right] \\
&= \int R(D(\mathbf{x})|\mathbf{x})p(\mathbf{x})d\mathbf{x}.
\end{aligned}
$$

Here we have used the standard result from probability theory that

$$\mathbb{E}\left[\mathbb{E}\left[X|Y\right]\right] = \mathbb{E}\left[X\right].$$

(See the supplementary notes for a proof.)

Bayesian decision theory

Clearly *the risk is minimised by the following decision rule*:

Given any $\mathbf{x} \in \mathbb{R}^d$: $D(\mathbf{x})$ outputs the action a_i that minimises $R(a_i|\mathbf{x})$

This D provides us with the *minimum possible risk*, or *Bayes risk* $R^\star$.

The rule specified is called the *Bayes decision rule*.

Example: minimum error rate classification

In supervised learning our aim is often to work in such a way that we *minimise the probability of making an error* when *predicting the label* for a *previously unseen example*.

What loss should we consider in these circumstances?

From basic probability theory, we know that *for any event E*

$$\Pr(E) = \mathbb{E}\left[\mathbb{I}\left[E\right]\right]$$

where $\mathbb{I}\left[\right]$ denotes the *indicator function*

$$\mathbb{I}\left[E\right] = \begin{cases} 1 & \text{if } E \text{ happens} \\ 0 & \text{otherwise} \end{cases}.$$

(See the supplementary notes for a proof.)

Example: minimum error rate classification

So if we are addressing a *supervised learning problem* with

- K classes $\{c_1, \ldots, c_K\}$.
- $L = K$ corresponding actions $\{a_1, \ldots, a_K\}$
- We interpret action a_i as meaning *'the input is in class c_i'*.
- The loss is defined as

$$\lambda_{ij} = \begin{cases} 1 & \text{if } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

then...

The risk R is

$$\begin{aligned} R &= \mathbb{E}_{(\mathbf{x},c)\sim p(\mathbf{X},C)}\left[\lambda(D(\mathbf{x}), C)\right] \\ &= \Pr\left(D(\mathbf{x}) \text{ chooses the wrong class}\right) \end{aligned}$$

so *the Bayes decision rule minimises the probability of error*.

Example: minimum error rate classification

What is the Bayes decision rule in this case?

$$\begin{aligned} R(a_i|\mathbf{x}) &= \sum_{j=1}^{K} \lambda_{ij} \mathrm{Pr}\left(c_j|\mathbf{x}\right)) \\ &= \sum_{i \neq j} \mathrm{Pr}\left(c_j|\mathbf{x}\right)) \\ &= 1 - \mathrm{Pr}\left(c_i|\mathbf{x}\right) \end{aligned}$$

so $D(\mathbf{x})$ should be *the class that maximises* $\mathrm{Pr}\left(C|\mathbf{x}\right)$.

THE IMPORTANT SUMMARY: Given a new $\mathbf{x}$ to classify, *choosing the class that maximises* $\mathrm{Pr}\left(C|\mathbf{x}\right)$ is the best strategy *if your aim is to minimize the probability of error*.

Bayesian supervised learning

But what about the *training sequence* $\mathbf{s}$?

Shouldn't the Bayes optimal classifier *depend on that as well*?

- Yes, it should *if there is uncertainty about the mechanism used to generate the data*.
- (All of the above *assumes that the mechanism is fixed*, so seeing examples has no effect on the optimal classifer.)
- In our case *we don't know what underlying* h *was used*. There is a *prior* $p(h)$.
- If you carry through the above derivation letting the *conditional risk* be conditional on *both* $\mathbf{x}$ *and* $\mathbf{s}$ then you find that...
- ...to minimize error probability you should maximize $\Pr(C|\mathbf{x}, \mathbf{s})$.

You should now work through the related exercise.

Bayesian supervised learning

But the uncertain *underlying hypothesis* h used to assign classes still doesn't appear!

Well, we want to maximize $\Pr(C|\mathbf{x},\mathbf{s})$:

$$\begin{aligned}
\Pr(C|\mathbf{x},\mathbf{s}) &= \sum_h \Pr(C,h|\mathbf{x},\mathbf{s}) \\
&= \sum_h \Pr(C|h,\mathbf{x},\mathbf{s}) \Pr(h|\mathbf{x},\mathbf{s}) \\
&= \sum_h \underbrace{\Pr(C|h,\mathbf{x})}_{\text{Likelihood}} \underbrace{\Pr(h|\mathbf{s})}_{\text{Posterior}}.
\end{aligned}$$

Here we have re-introduced h using marginalisation.

Bayesian supervised learning

So our classification should be

$$C = \underset{C \in \{c_1,\ldots,c_K\}}{\operatorname{argmax}} \sum_h \Pr(C|h,\mathbf{x}) \Pr(h|\mathbf{s})$$

Of course, when dealing with hypotheses defined by weights $\mathbf{w}$ the sum becomes an integral

$$C = \underset{C \in \{c_1,\ldots,c_K\}}{\operatorname{argmax}} \int_{\mathbb{R}^W} \Pr(C|\mathbf{w},\mathbf{x}) \Pr(\mathbf{w}|\mathbf{s})\ d\mathbf{w}$$

where W is the number of weights. The *key point*:

- You can also write these equations in the form

$$C = \underset{C \in \{c_1,\ldots,c_K\}}{\operatorname{argmax}} \mathbb{E}_{h \sim \Pr(h|\mathbf{s})} \left[\Pr(C|h,\mathbf{x})\right]$$

- We are *not choosing a single h*.
- We are *averaging* the predictions of *all possible* functions h.
- In doing this we are *weighting* according to *how probable they are*.

A word of caution

We know the optimal classifier, so we've *solved supervised learning* right?

WRONG!!!

In practice, solving

$$C = \operatorname*{argmax}_{C \in \{c_1, \ldots, c_K\}} \mathbb{E}_{h \sim \mathbf{Pr}(h|\mathbf{s})} \left[\mathbf{Pr} \left(C | h, \mathbf{x} \right) \right]$$

is *intractible in all but the simplest of cases*.

Thou shalt beware *Bayesians bearing gifts*.

They may well be too good to be true…

Machine Learning and Bayesian Inference

Major subject number two:

The road to *Support Vector Machines (SVMs)*.

It is worth remembering that *not all state-of-the-art machine learning is inherently probabilistic*.

There is *good reason* for this: you can almost *never* actually compute

$$C = \operatorname*{argmax}_{C \in \{c_1, \ldots, c_K\}} \mathbb{E}_{h \sim \Pr(h|\mathbf{s})} \left[\Pr\left(C | h, \mathbf{x} \right) \right]$$

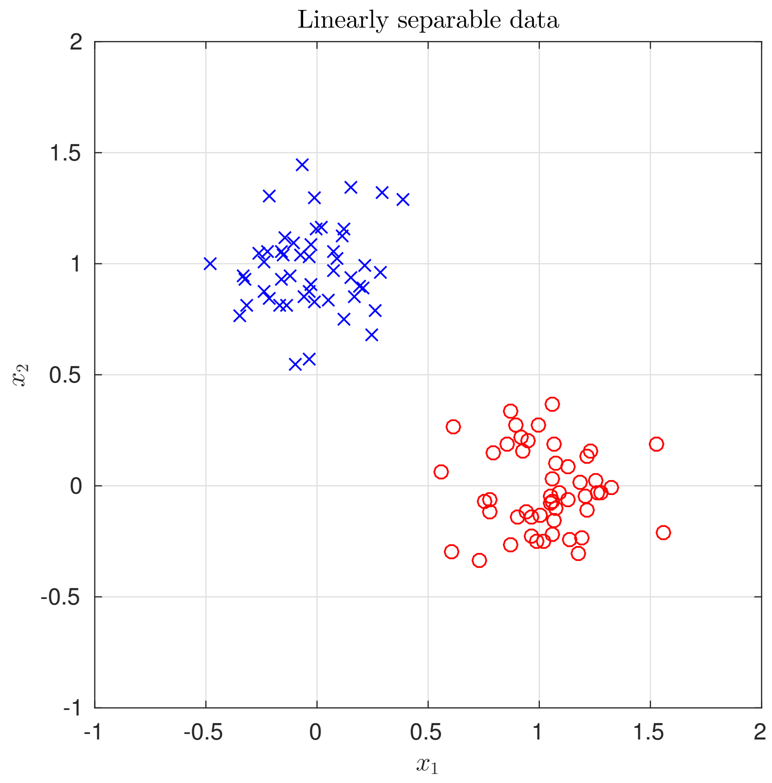
So before we go any further, let's see how far it's possible to get using only *linear* methods.

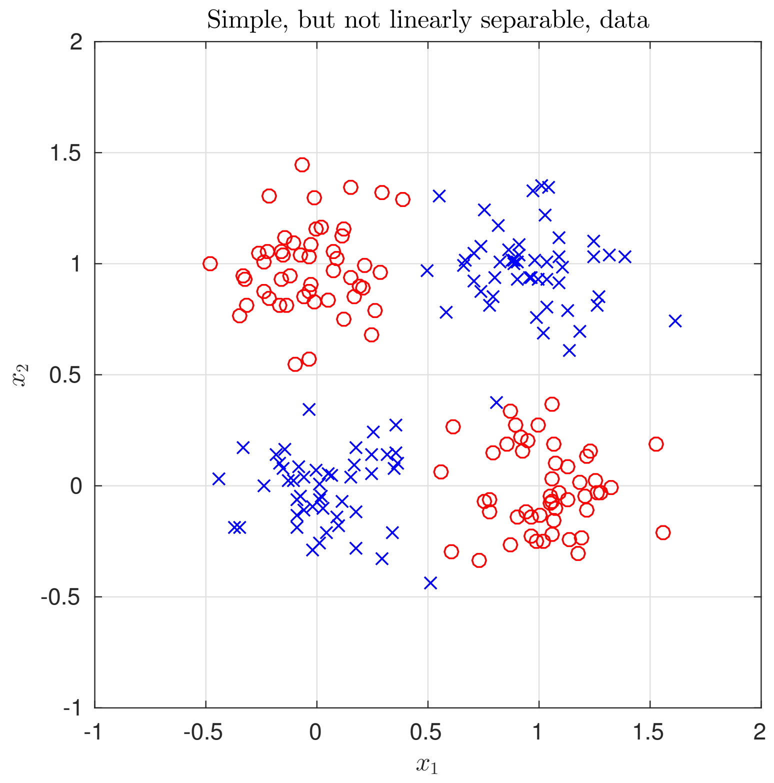
This is generally a *good idea*.

Why? Because *linear methods are EASY!*

The problem with linear classifiers

Purely linear classifiers or regressors are great for some problems *but awful for others*:



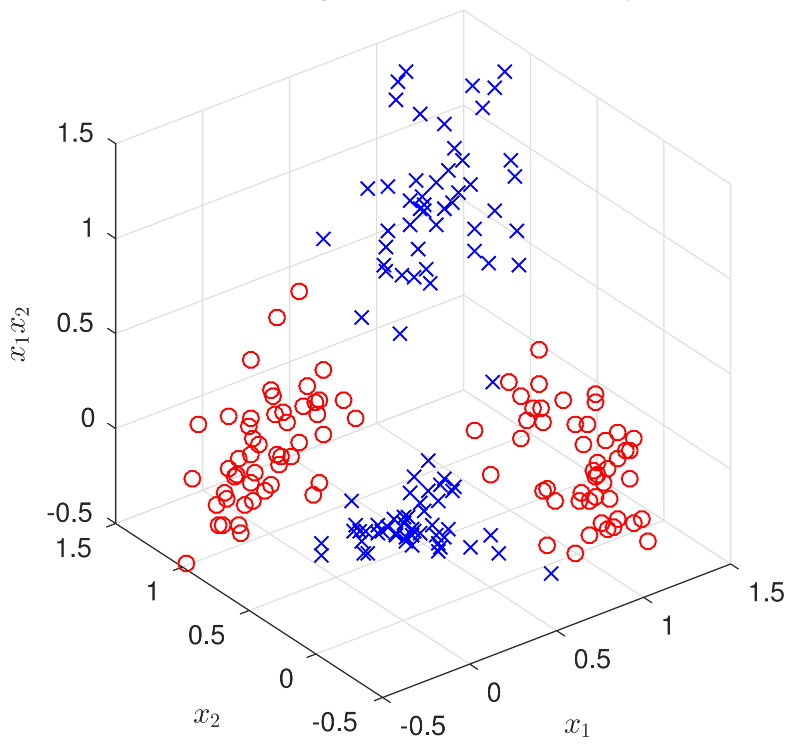


This example actually killed neural network research for many years.

The kernel trick

One way of getting around this problem is to employ the *kernel trick*:

Data from second graph, now more linearly seperable

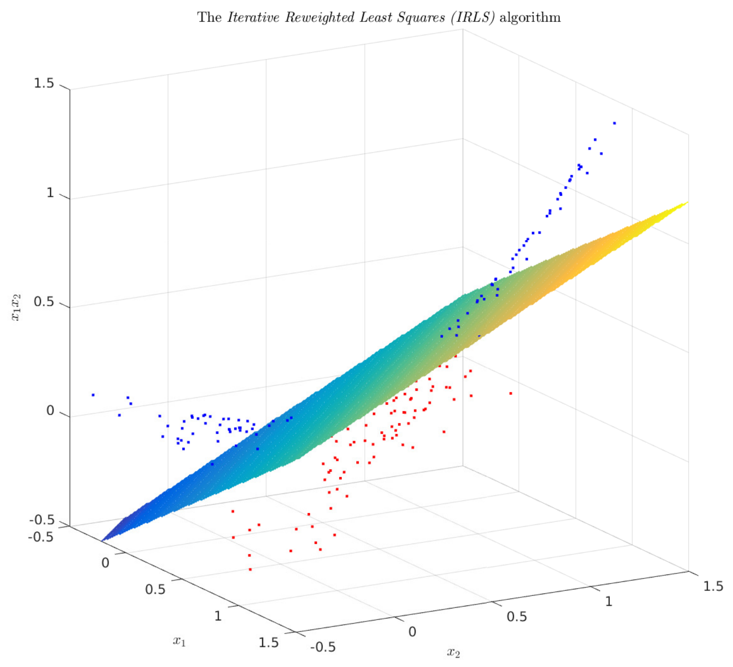


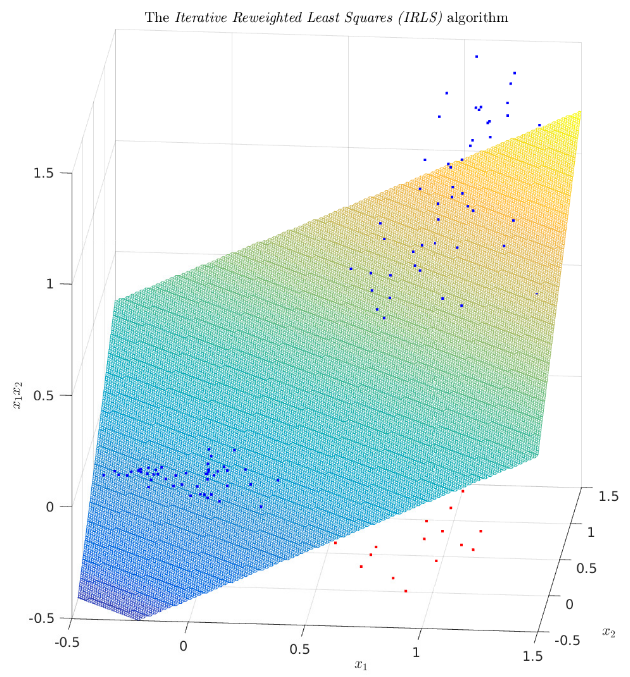
Map the data into a *bigger space* and hope it's *more separable* there.

Here, we've *added one new dimension* by introducing a *new feature* equal to $x_1 x_2$.

The kernel trick

Here is a *linear* hypothesis learned to separate the two classes in the new space.





This was obtained using the *Iterative Recursive Least Squares (IRLS)* algorithm.

We'll be deriving this in a moment...

Linear classifiers

We've already seen the linear classifier

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$$

Or $h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x})$ if we add an extra element having constant value 1 to $\mathbf{x}$.

Make it nonlinear by introducing *basis functions* ϕ_i:

$$\mathbf{\Phi}^T(\mathbf{x}) = \begin{bmatrix} \phi_1(\mathbf{x}) & \phi_2(\mathbf{x}) & \cdots & \phi_k(\mathbf{x}) \end{bmatrix}$$

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma\left(w_0 + \sum_{i=1}^{k} w_i \phi_i(\mathbf{x})\right)$$

or assuming there's a basis function $\phi(\mathbf{x}) = 1$

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{\Phi}(\mathbf{x})).$$

Linear regression

We've already seen *linear regression*. We use $\sigma(x) = x$ and we have *training data*

$$\mathbf{s}^T = \left[(\mathbf{x}_1, y_1) \;\; (\mathbf{x}_2, y_2) \;\; \cdots \;\; (\mathbf{x}_m, y_m)\right].$$

I want to minimize

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - h(\mathbf{x}_i, \mathbf{w}))^2.$$

Last year we would have found the *gradient* of $E(\mathbf{w})$ and used *gradient descent*

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}_t}.$$

But for *linear regression* there is an easier way. We can *directly* solve the equation

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{0}.$$

Calculus with matrices

It is much easier to handle this kind of calculation in matrix/vector format than by writing it out in full.

For example, if $\mathbf{a}$ and $\mathbf{x}$ are both vectors in $\mathbb{R}^n$ we can verify that

$$\frac{\partial \mathbf{a}^T\mathbf{x}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{a}^T\mathbf{x}}{\partial x_1} & \frac{\partial \mathbf{a}^T\mathbf{x}}{\partial x_2} & \cdots & \frac{\partial \mathbf{a}^T\mathbf{x}}{\partial x_n} \end{bmatrix}^T = \mathbf{a}$$

because for each element x_j

$$\frac{\partial \mathbf{a}^T\mathbf{x}}{\partial x_j} = \frac{\partial}{\partial x_j}(a_1x_1 + a_2x_2 + \cdots + a_nx_n) = a_j$$

You should verify for yourself that most standard manipulations involving derivatives carry over directly.

Exercise: Show that if $\mathbf{A} \in \mathbb{R}^{n\times n}$ is symmetric then

$$\frac{\partial \mathbf{x}^T\mathbf{A}\mathbf{x}}{\partial \mathbf{x}} = 2\mathbf{A}\mathbf{x}$$

Linear regression

Write

$$\mathbf{\Phi} = \begin{bmatrix} \mathbf{\Phi}^T(\mathbf{x}_1) \\ \mathbf{\Phi}^T(\mathbf{x}_2) \\ \vdots \\ \mathbf{\Phi}^T(\mathbf{x}_m) \end{bmatrix}$$

so

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2}(\mathbf{y} - \mathbf{\Phi}\mathbf{w})^T(\mathbf{y} - \mathbf{\Phi}\mathbf{w}) \\ &= \frac{1}{2}\left(\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{\Phi}\mathbf{w} + \mathbf{w}^T\mathbf{\Phi}^T\mathbf{\Phi}\mathbf{w}\right) \end{aligned}$$

and

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{\Phi}^T\mathbf{\Phi}\mathbf{w} - \mathbf{\Phi}^T\mathbf{y}$$

Linear regression

So the optimum solution is obtained by solving

$$\mathbf{\Phi}^T \mathbf{\Phi} \mathbf{w} = \mathbf{\Phi}^T \mathbf{y}$$

giving

$$\mathbf{w}_{\text{opt}} = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{y}$$

This is the *maximum likelihood* solution to the problem, assuming noise is Gaussian.

Recall that we can also consider the *maximum a posteriori (MAP)* solution. . .

Linear regression: the MAP solution

We saw earlier that *to get the MAP solution we minimize the error*

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}\left((y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2\right) + \frac{\lambda}{2}||\mathbf{w}||^2.$$

It is an *exercise* to show that the solution is:

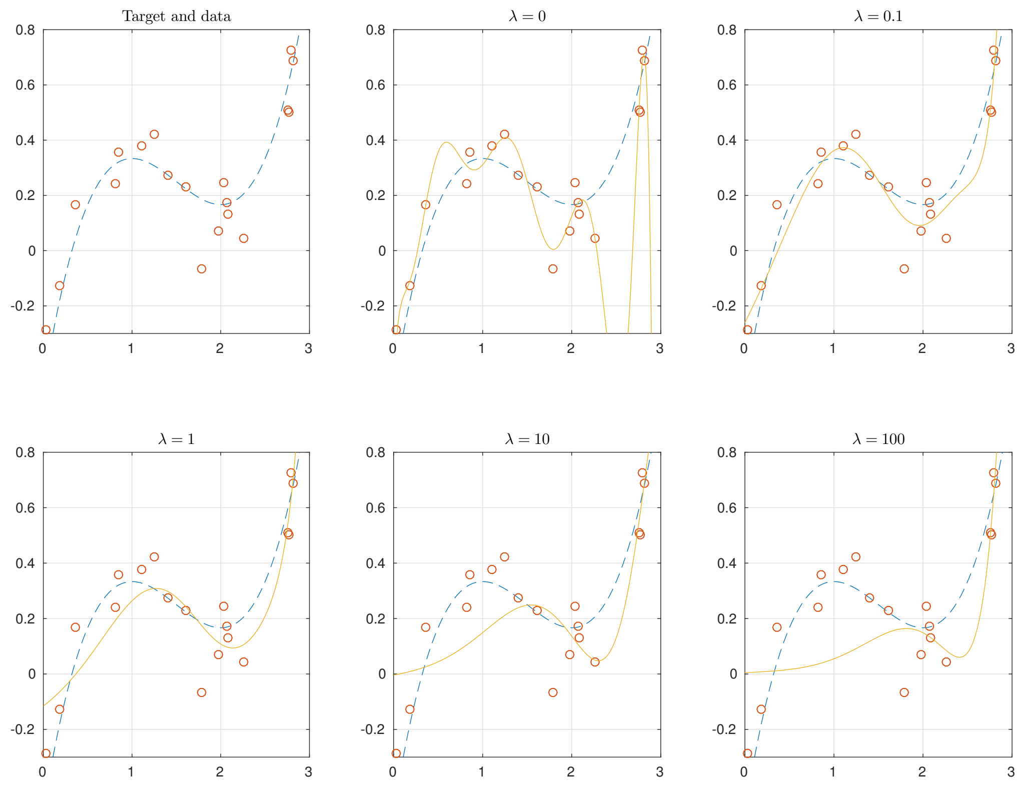
$$\mathbf{w}_{\text{opt}} = (\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \lambda\mathbf{I})^{-1}\boldsymbol{\Phi}^T\mathbf{y}$$

This is *regularized linear regression* or *ridge regression*.

Linear regression: the MAP solution

This can make a *huge difference*.

Revisiting our earlier simple example and training using *different values for λ*:



How can we choose λ? We'll address this a little later…

Iterative re-weighted least squares

What about if we're *classifying* rather than doing regression?

We now need to use a non-linear σ, typically the *sigmoid function*, so

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma_\theta(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x})).$$

We saw earlier that to get the maximum likelihood solution we should maximize the likelihood

$$p(\mathbf{s}|\mathbf{w}) = \prod_{i=1}^{m} \left[\sigma_\theta(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i))\right]^{y_i} \left[1 - \sigma_\theta(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i))\right]^{(1-y_i)} p(\mathbf{x}_i).$$

Assuming you've been *completing the exercises* you now know that this corresponds to minimizing the error

$$E(\mathbf{w}) = -\left[\sum_{i=1}^{m} y_i \log \sigma_\theta(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i)) + (1 - y_i) \log(1 - \sigma_\theta(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i)))\right].$$

Iterative re-weighted least squares

Introducing the extra nonlinearity means we can no longer minimize

$$E(\mathbf{w}) = -\left[\sum_{i=1}^{m} y_i \log \sigma_\theta(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i)) + (1 - y_i) \log(1 - \sigma_\theta(\mathbf{w}^T \mathbf{\Phi}(\mathbf{x}_i)))\right].$$

just by computing a derivative and solving. (Sad, but I suggest you *get used to it*!)

We need to go back to an iterative solution: this time using the *Newton-Raphson method*.

Given a function $f : \mathbb{R} \to \mathbb{R}$, to find where $f(x) = 0$ iterate as

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}.$$

Obviously, to find a *minimum* we can iterate as

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}.$$

This works for 1 dimension. How about many dimensions?

Iterative re-weighted least squares

The Newton-Raphson method *generalizes easily to functions of a vector*:

To minimize $E : \mathbb{R}^n \to \mathbb{R}$ iterate as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{H}^{-1}(\mathbf{w}_t) \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}.$$

Here the *Hessian* is the matrix of *second derivatives* of $E(\mathbf{w})$

$$\mathbf{H}_{ij}(\mathbf{w}) = \frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j}.$$

All we need to do now is to *work out the derivatives...*

Iterative re-weighted least squares

$$E(\mathbf{w}) = -\left[\sum_{i=1}^{m} y_i \log \sigma_\theta(\mathbf{w}^T\boldsymbol{\Phi}(\mathbf{x}_i)) + (1-y_i)\log(1-\sigma_\theta(\mathbf{w}^T\boldsymbol{\Phi}(\mathbf{x}_i)))\right].$$

Simplifying slightly we use $\theta = 1$ and define $z_i = \sigma(\mathbf{w}^T\boldsymbol{\Phi}(\mathbf{x}_i))$. So

$$\begin{aligned}\frac{\partial E(\mathbf{w})}{\partial w_k} &= -\left[\sum_{i=1}^{m} y_i \frac{1}{z_i}\frac{\partial z_i}{\partial w_k} + (1-y_i)\frac{-1}{1-z_i}\frac{\partial z_i}{\partial w_k}\right] \\ &= \sum_{i=1}^{m} \frac{\partial z_i}{\partial w_k}\left(\frac{1-y_i}{1-z_i} - \frac{y_i}{z_i}\right) \\ &= \sum_{i=1}^{m} \frac{\partial z_i}{\partial w_k}\frac{z_i - y_i}{z_i(1-z_i)}.\end{aligned}$$

Iterative re-weighted least squares

So

$$\frac{\partial E(\mathbf{w})}{\partial w_k} = \sum_{i=1}^{m} \frac{\partial z_i}{\partial w_k} \frac{z_i - y_i}{z_i(1-z_i)}.$$

Thus using the fact that

$$\sigma'(.) = \sigma(.)(1-\sigma(.))$$

we have

$$\frac{\partial z_i}{\partial w_k} = \frac{\partial}{\partial w_k}\sigma(\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i)) = z_i(1-z_i)\phi_k(\mathbf{x}_i)$$

and therefore

$$\boxed{\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{\Phi}^T(\mathbf{z}-\mathbf{y}).}$$

Iterative re-weighted least squares

It is an *exercise* to show that

$$\mathbf{H}_{ij}(\mathbf{w}) = \sum_{i=1}^{m} z_i(1 - z_i)\phi_k(\mathbf{x}_i)\phi_j(\mathbf{x}_i)$$

and therefore

$$\mathbf{H}(\mathbf{w}) = \mathbf{\Phi}^T \mathbf{Z} \mathbf{\Phi}$$

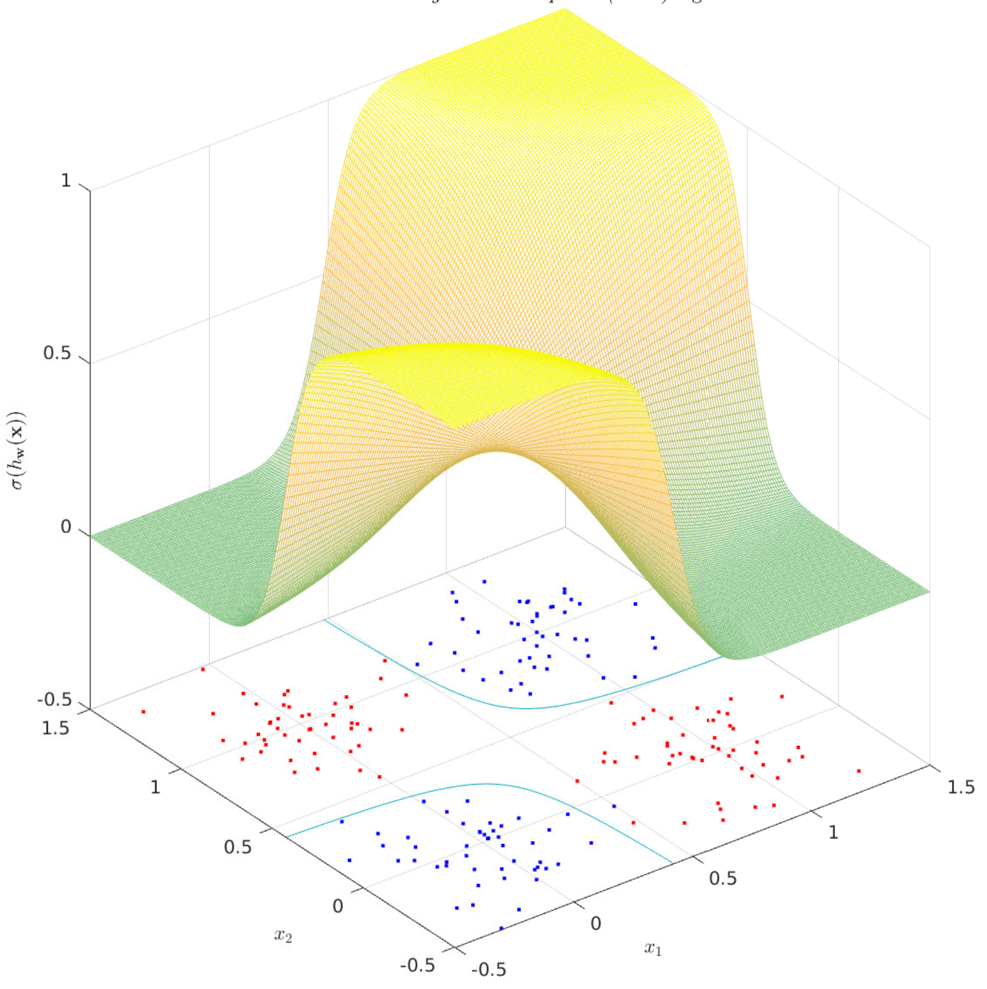
where $\mathbf{Z}$ is a diagonal matrix with diagonal elements $z_i(1 - z_i)$.

This gives us the *iterative re-weighted least squares algorithm (IRLS)*

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \left[\mathbf{\Phi}^T \mathbf{Z} \mathbf{\Phi}\right]^{-1} \mathbf{\Phi}^T (\mathbf{z} - \mathbf{y}).$$

Iterative re-weighted least squares

The *Iterative Reweighted Least Squares (IRLS)* algorithm


$\sigma(h_{\mathbf{w}}(\mathbf{x}))$
x_2
x_1