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*.

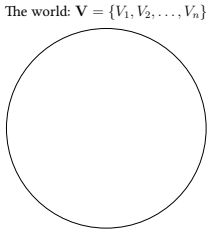
Bayesian networks and *Markov random fields* allow us to represent *probability distributions*.

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*

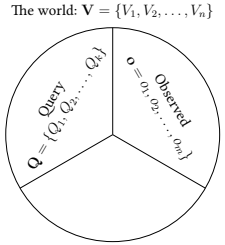
$$\boxed{\Pr(\mathbf{V}) = \Pr(V_1, V_2, \ldots, V_n)}$$

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 k *query variables*.

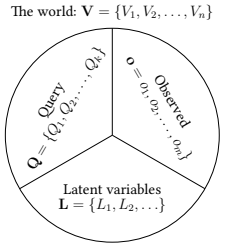


Then *inference* corresponds to computing a *conditional distribution*

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

General knowledge representation and inference: the BIG PICTURE

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

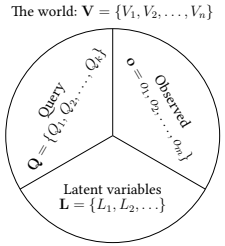


To compute a conditional distribution from a knowledge base $\Pr(\mathbf{V})$ we have to *sum over the latent variables*

$$\begin{aligned}\Pr(\mathbf{Q}|o_1, o_2, \ldots, o_m) &= \sum_{\mathbf{L}} \Pr(\mathbf{Q}, \mathbf{L}|o_1, o_2, \ldots, o_m) \\ &= \boxed{\frac{1}{Z} \sum_{\mathbf{L}} \underbrace{\Pr(\mathbf{Q}, \mathbf{L}, o_1, o_2, \ldots, o_m)}_{\text{Knowledge base}}}\end{aligned}$$

General knowledge representation and inference: the BIG PICTURE

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



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

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

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:

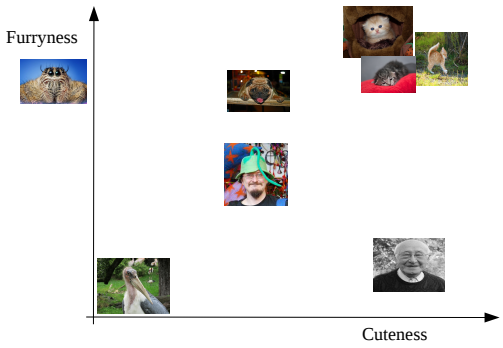
1. *Pattern Recognition and Machine Learning*. Christopher M. Bishop, Springer, 2006.
2. *Machine Learning: A Probabilistic Perspective.* Kevin P. Murphy, The MIT Press, 2012.

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*.

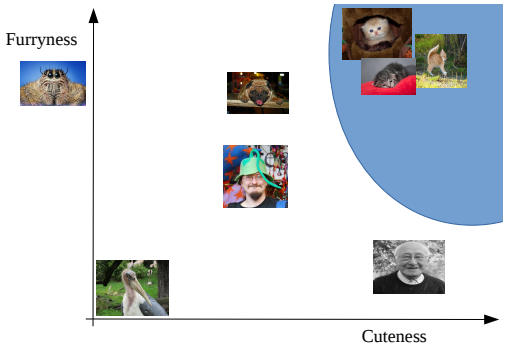


Evil Robot hates kittens, and consequently wants to build a *kitten detector*.

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

What have we done so far?

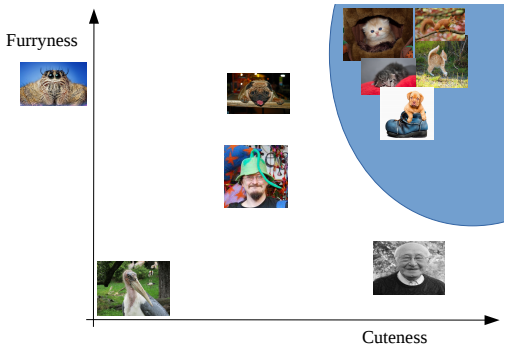
Provided he has some examples labelled as *kitten* or *not kitten*…



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

What have we done so far?

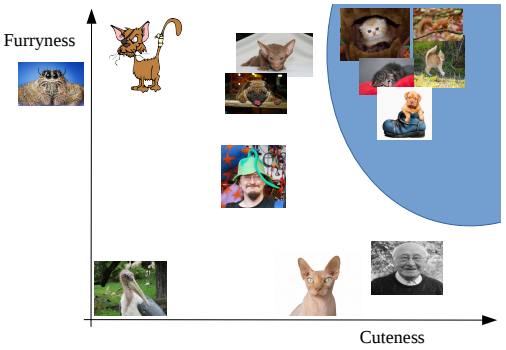
Of course, when put into practice…



…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* $h : \mathbb{R}^n \to Y$ 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 $h_{\mathbf{w}}(\mathbf{x})$.

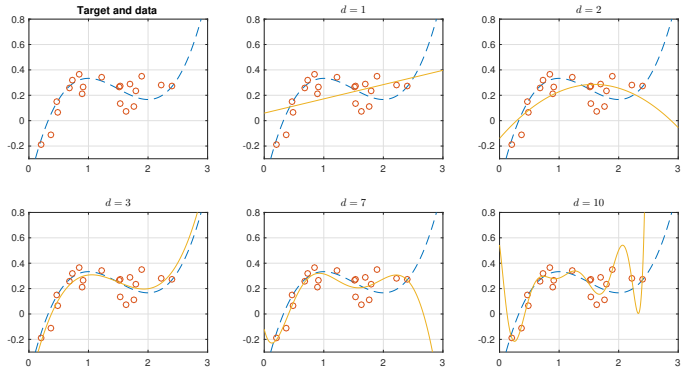
If it can do this well it *generalizes*.

- 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* $h_{\mathbf{w}} : \mathbb{R}^n \to [0, 1]$ of the form

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

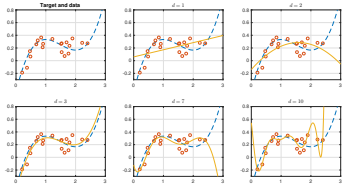
What have we done so far?

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 $\mathbf{s} = \begin{bmatrix}(x_1, y_1) & (x_2, y_2) & \cdots & (x_m, y_m)\end{bmatrix}$.

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

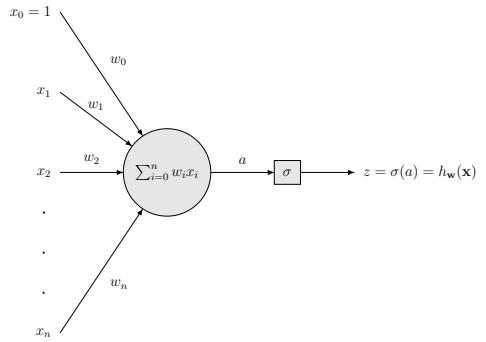
$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(x_i))^2.$$

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

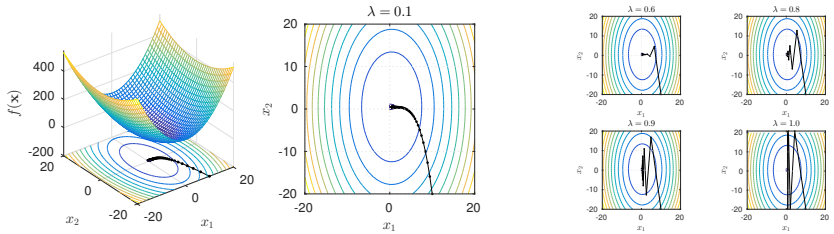
$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - \mathbf{w}^T\mathbf{x}_i)^2$$

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:

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

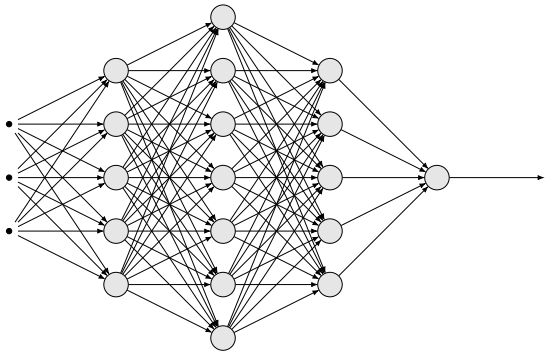
where $\mathbf{x}_i^{(j)}$ is the jth element of $\mathbf{x}_i$. So:

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

The multilayer perceptron

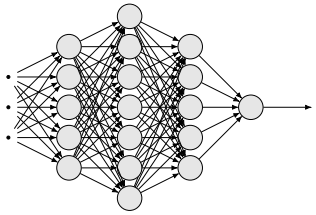
Real problems tend also to be *nonlinear*.

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.

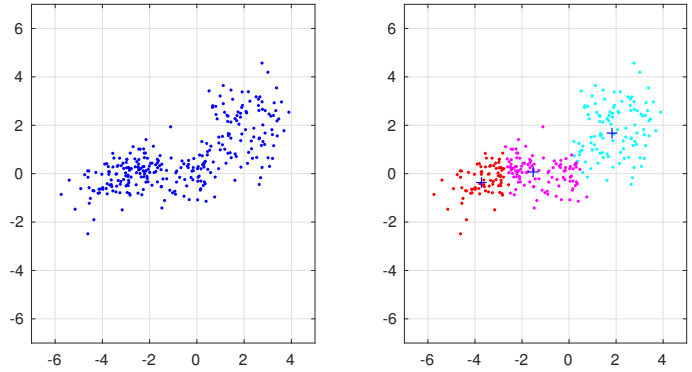
If you understand backpropagation, you already know the key idea needed for *stuff involving the word 'deep'*.

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 $\mathbb{R}^n$ …

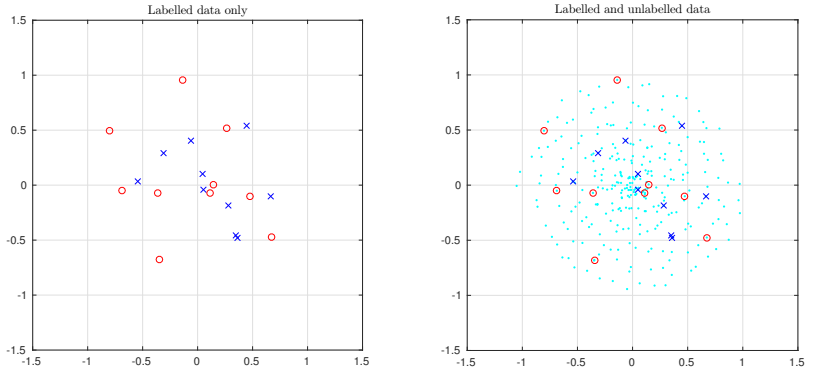


…and we want to find some *regularity*.

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

$$\mathbf{s}^T = \begin{bmatrix} (\mathbf{x}_1, y_1) & (\mathbf{x}_2, y_2) & \ldots & (\mathbf{x}_m, y_m) \end{bmatrix}.$$

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

$$\mathbf{s}^T = \begin{bmatrix}(\mathbf{x}_1, y_1) & (\mathbf{x}_2, y_2) & \cdots & (\mathbf{x}_m, y_m)\end{bmatrix}$$

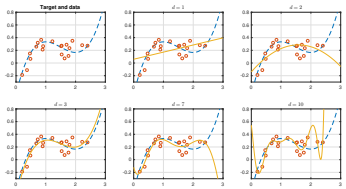
represents m iid samples from the relevant distribution.

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:

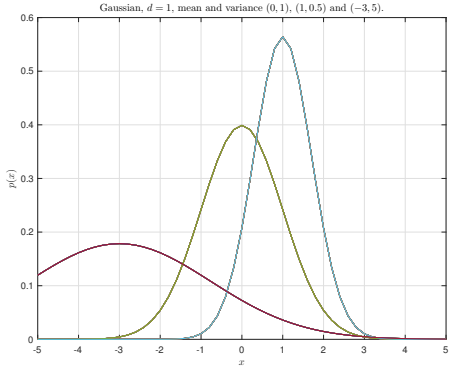


We have spoken of an *unknown underlying function* f used to generate the data. In fact, this is the *hypothesis* $h_{\mathbf{w}}$ that we *want to identify* by choosing $\mathbf{w}$.

> 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

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

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:

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

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

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

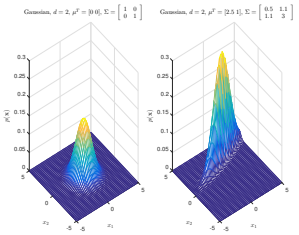
  pretty much *out of thin air* because it seemed to make sense.
- We've just shown that hidden inside it is an *assumption*: that *noise* in the data is *Gaussian.*
- We've also uncovered a *second assumption*: that maximizing the likelihood is the *right thing to do.*

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

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

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

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

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.*

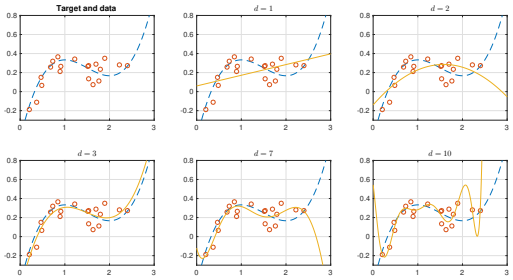
In addition to the assumptions made by maximum likelihood:

- 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

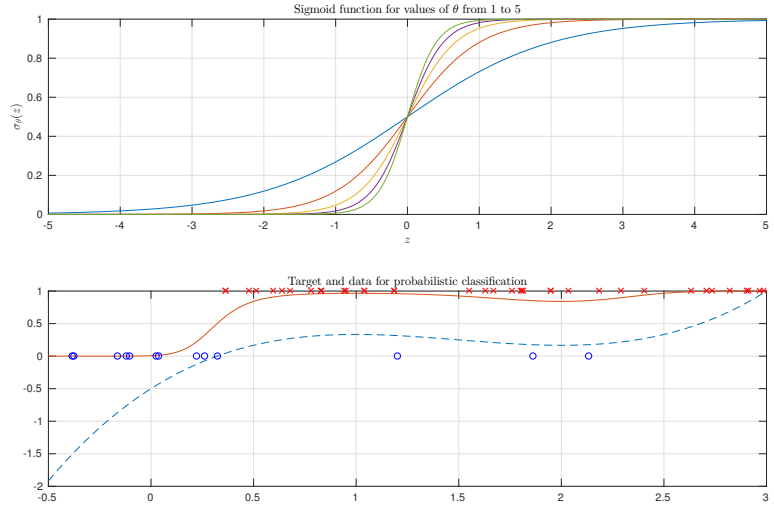
$$\Pr(Y = 1|\mathbf{x}, \mathbf{w}) = \sigma_\theta(h_{\mathbf{w}}(\mathbf{x}))$$

and use something like

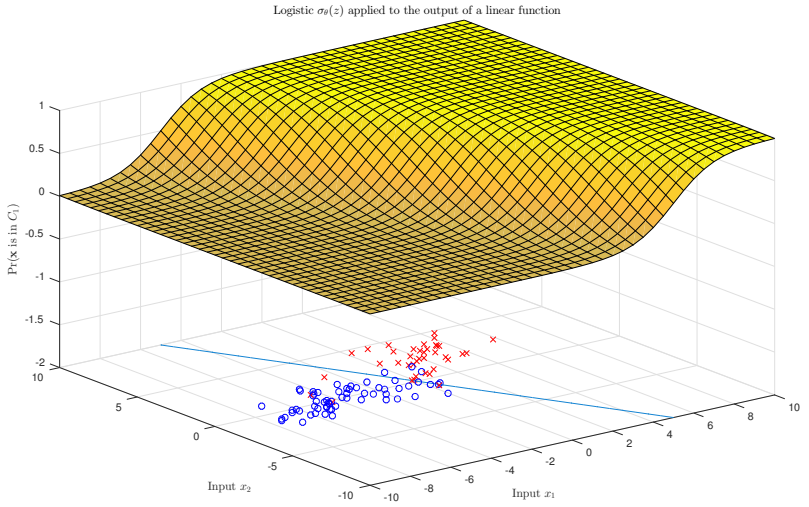
$$\sigma_\theta(z) = \frac{1}{1 + \exp(-\theta z)}$$

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 $(\mathbf{x}, y)$

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

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

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

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

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

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

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

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* $D : \mathbb{R}^d \to \{a_1, \ldots, a_L\}$ telling us *what action to take on seeing any* $\mathbf{x} \in \mathbb{R}^d$.

The average loss, or *risk*, is

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

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

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

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?

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

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

THE IMPORTANT SUMMARY: Given a new $\mathbf{x}$ to classify, *choosing the class that maximises* $\Pr\left(C|\mathbf{x}\right)$ 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 $\Pr\left(C|\mathbf{x}, \mathbf{s}\right)$.

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 $\Pr\left(C|\mathbf{x}, \mathbf{s}\right)$:

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

Here we have re-introduced h using marginalisation.

Bayesian supervised learning

So our classification should be

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

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

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

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

- You can also write these equations in the form

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

- 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

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

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

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

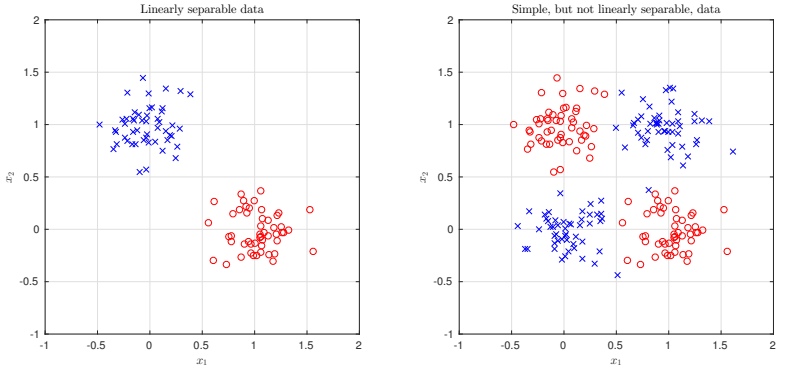
So before we go any further, let's see how far it's possible to get using only *linear* methods.

This is generally a *good idea.*

Why? Because *linear methods are EASY!*

The problem with linear classifiers

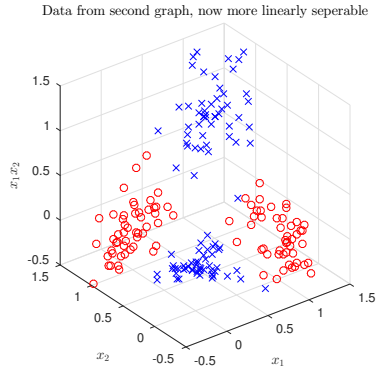
Purely linear classifiers or regressors are great for some problems *but awful for others*:



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*:



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_1x_2.

machine Learning Commandments

Thou shalt not *rely on toy data.*

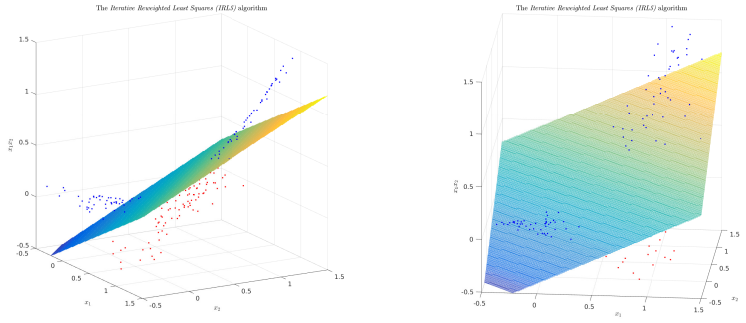
Resources such as the *UCI Machine Learning Repository* are there for a very good reason:

`https://archive.ics.uci.edu/ml/index.php`

Thou shalt not *rebrand the kernel trick.*

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*

$$\mathbf{s}^T = \begin{bmatrix}(\mathbf{x}_1, y_1) & (\mathbf{x}_2, y_2) & \cdots & (\mathbf{x}_m, y_m)\end{bmatrix}.$$

I want to minimize

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

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:

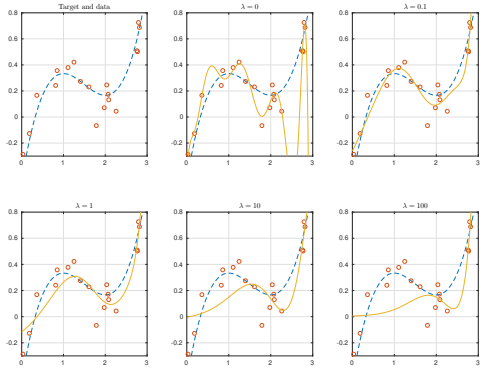
$$\mathbf{w}_{\text{opt}} = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{\Phi}^T\mathbf{y}$$

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

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

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

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

Iterative re-weighted least squares

It is an *exercise* to show that

$$\mathbf{H}_{ij}(\mathbf{w}) = \sum_{k=1}^{m} z_k(1-z_k)\phi_i(\mathbf{x}_k)\phi_j(\mathbf{x}_k)$$

and therefore

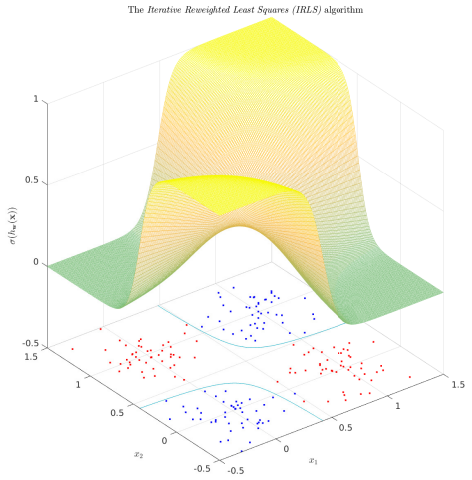
$$\mathbf{H}(\mathbf{w}) = \mathbf{\Phi}^T\mathbf{Z}\mathbf{\Phi}$$

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

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



Gaussian processes: inference with functions instead of parameters

There is an alternative approach to Bayesian regression and classification:

The fundamental idea is to *not* think in terms of *weights* $\mathbf{w}$ *that specify functions.*

Instead the idea is to *deal with functions directly.*

Fundamental to this is the concept of a *Gaussian process.*

Gaussian processes: inference with functions instead of parameters

We will continue to omit the dependencies on $\mathbf{x}$ and $\mathbf{X}$ to keep the notation simple.

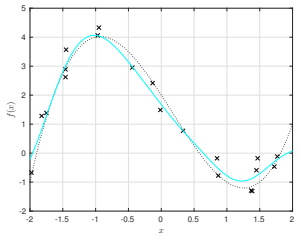
We have seen that *inference* can be performed by:

1. Computing the *posterior density* $p(\mathbf{w}|\mathbf{y})$ of the parameters given the observed labels.
2. Computing the *Bayes-optimal* prediction
$$p(Y|\mathbf{y}) = \int p(Y|\mathbf{w})p(\mathbf{w}|\mathbf{y})\ d\mathbf{w}$$
which is the expected value of the *likelihood* for a new point $\mathbf{x}$.
3. Choosing any *hyperparameters* $\mathbf{p}$ using the *evidence* $p(\mathbf{y}|\mathbf{p})$.

But shouldn't we deal with *functions* directly, not via parameters?

Gaussian processes: inference with functions instead of parameters

What happens if we *deal directly with functions* f, rather than choosing them via *parameters*?



Can we change the equation for prediction to

$$p(Y|\mathbf{y}) = \int p(Y|f)p(f|\mathbf{y})\ df$$

in any sensible way?

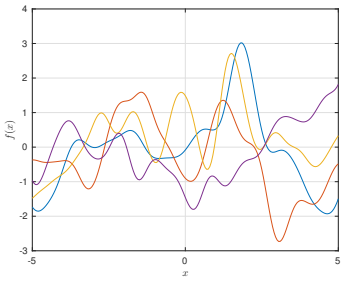
Can we change the equation for prediction to

$$p(Y|\mathbf{y}) = \int p(Y|f)p(f|\mathbf{y})\ df$$

in any sensible way?

This obviously requires us to talk about *probability densities over functions*. That is probably not something you have ever seen before.

In the diagram: four samples $f \sim p(F)$ from a *probability density defined on functions*.

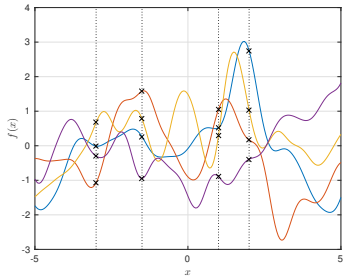


This is quite straightforward, using the concept of a *Gaussian process (GP)*.

Definition: say we have a set of RVs. This set forms a *Gaussian process* if any *finite subset* of them is *jointly Gaussian distributed*.

The same four samples $f \sim p(F)$, where F is in fact a GP.

The crosses mark the values of the sampled functions at four different values of x.



Because F is a GP *any* such finite set of values has a jointly Gaussian distribution.

What happens when we randomly select a function that is a GP?

- We are only ever interested in a *finite number of its values*.
- This is because we only need to deal with the values in the *training set* and for *any new points* we want to predict.
- Consequently we can use a GP as a *prior* rather than having a prior $p(\mathbf{w})$.

Note again the key point: we are randomly selecting *functions* and we can say something about their behaviour for *any finite collection of arguments*.

And that is enough, as we only ever have *finite quantities of data*.

To specify a GP on vectors in $\mathbb{R}^n$, we just need:

1. A *mean function* $m : \mathbb{R}^n \to \mathbb{R}$.
2. A *covariance function* $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$.

$$m(\mathbf{x}) = \mathbb{E}_{f\sim F}\left[f(\mathbf{x})\right]$$
$$k(\mathbf{x}_1, \mathbf{x}_2) = \mathbb{E}_{f\sim F}\left[(f(\mathbf{x}_1) - m(\mathbf{x}_1))(f(\mathbf{x}_2) - m(\mathbf{x}_2))\right]$$

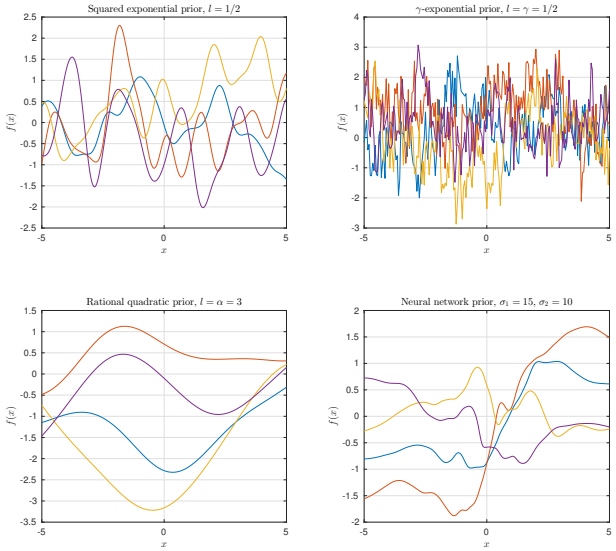
We then write

$$F \sim \mathrm{GP}(m, k)$$

to denote that F is a GP.

By specifying m and k we get different kinds of function when sampling F.

GP priors



Covariance functions

Polynomial;

$$k(\mathbf{x}_1, \mathbf{x}_2) = (c + \mathbf{x}_1^T \mathbf{x}_2)^k$$

Exponential:

$$k(\mathbf{x}_1, \mathbf{x}_2) = \exp\left(-\frac{|\mathbf{x}_1 - \mathbf{x}_2|}{l}\right)$$

Squared exponential:

$$k(\mathbf{x}_1, \mathbf{x}_2) = \exp\left(-\frac{|\mathbf{x}_1 - \mathbf{x}_2|^2}{2l^2}\right)$$

Gamma exponential:

$$k(\mathbf{x}_1, \mathbf{x}_2) = \exp\left(-\left(\frac{|\mathbf{x}_1 - \mathbf{x}_2|}{l}\right)^\gamma\right)$$

Covariance functions

Rational quadratic;

$$k(\mathbf{x}_1, \mathbf{x}_2) = \left(1 + \frac{|\mathbf{x}_1 - \mathbf{x}_2|^2}{2\alpha l^2}\right)^{-\alpha}$$

Exponential:

$$k(\mathbf{x}_1, \mathbf{x}_2) = \sin^{-1}\left(\frac{2(\mathbf{x}_1')^T \boldsymbol{\Sigma} \mathbf{x}_2'}{((1 + 2(\mathbf{x}_1')^T \boldsymbol{\Sigma} \mathbf{x}_1')(1 + 2(\mathbf{x}_2')^T \boldsymbol{\Sigma} \mathbf{x}_2'))^{1/2}}\right)$$

where $(\mathbf{x}')^T = \begin{bmatrix}1 & \mathbf{x}^T\end{bmatrix}$.

As usual these have associated *hyperparameters*.

These have to be dealt with correctly as always.

Gaussian processes: generating data

Say we have some data

$$y_i = f(\mathbf{x}_i)$$

for $i = 1, \ldots, m$ and $f \sim \mathrm{GP}(m, k)$. (Remember, the $\mathbf{x}_i$ are *fixed*, not RVs.)

Any *finite set of points* must be *jointly Gaussian*. So

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{m}, \mathbf{K})$$

where

$$\mathbf{m}^T = \begin{bmatrix}m(\mathbf{x}_1) & \cdots & m(\mathbf{x}_m)\end{bmatrix}$$

and $\mathbf{K}$ is the *Gram matrix* $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$.

Note 1: this is *not* $p(\mathbf{y}|f)$. We can completely remove the need for integration!

Note 2: from now on we will assume $m(\mathbf{x}) = 0$. (It is straightforward to incorporate a non-zero mean.)

Gaussian processes: generating data with noise

Now add noise to the data.

Say we add Gaussian noise so

$$y_i = f(\mathbf{x}_i) + \epsilon_i.$$

Again, $i = 1, \ldots, m$ and $f \sim \text{GP}(m, k)$, but now we also have

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2).$$

As we are *adding Gaussian RVs*, we have

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{0}, \mathbf{K} + \sigma^2\mathbf{I}).$$

BUT: in order to do *prediction* we actually need to involve a new point $\mathbf{x}'$, for which we want to predict the corresponding value y'.

Gaussian processes: prediction

SO: we incorporate $\mathbf{x}'$, for which we want to predict the corresponding value y'.

By exactly the same argument

$$p(y', \mathbf{y}) = \mathcal{N}(\mathbf{0}, \mathbf{K}')$$

where

$$\mathbf{K}' = \begin{bmatrix} k & \mathbf{k}^T \\ \mathbf{k} & \mathbf{K} + \sigma^2\mathbf{I} \end{bmatrix}$$
$$\mathbf{k}^T = \begin{bmatrix} k(\mathbf{x}', \mathbf{x}_1) & \cdots & k(\mathbf{x}', \mathbf{x}_m) \end{bmatrix}$$
$$k = k(\mathbf{x}', \mathbf{x}') + \sigma^2.$$

Note 1: all we've done here is to expand the Gram matrix by an extra row and column to get $\mathbf{K}'$.

Note 2: whether or not you include σ^2 in k is a matter of choice. What difference does it make? (This is an *Exercise.*)

Gaussian density: marginals and conditionals

For a normal RV $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

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

Split $\mathbf{x}$ so

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$$

and correspondingly

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix} \qquad \boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix}.$$

What are $p(\mathbf{x}_1)$ and $p(\mathbf{x}_1|\mathbf{x}_2)$?

Gaussian density: marginals and conditionals

Define the *precision matrix*

$$\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{bmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{bmatrix}.$$

It is possible to show that

$$p(\mathbf{x}_1) = \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11})$$
$$p(\mathbf{x}_1|\mathbf{x}_2) = \mathcal{N}(\boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1}\boldsymbol{\Lambda}_{12}(\mathbf{x}_2 - \boldsymbol{\mu}_2), \boldsymbol{\Lambda}_{11}^{-1}).$$

Inverting a block matrix

In the last slide, we see:

$$\boldsymbol{\Sigma}^{-1} = \begin{bmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{bmatrix}.$$

Re-writing $\boldsymbol{\Sigma}$ as

$$\boldsymbol{\Sigma} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

it is possible to show (it is an *Exercise* to do this) that

$$\begin{aligned} \boldsymbol{\Lambda}_{11} &= \mathbf{A}' \\ \boldsymbol{\Lambda}_{12} &= -\mathbf{A}'\mathbf{B}\mathbf{D}^{-1} \\ \boldsymbol{\Lambda}_{21} &= -\mathbf{D}^{-1}\mathbf{C}\mathbf{A}' \\ \boldsymbol{\Lambda}_{22} &= \mathbf{D}^{-1} + \mathbf{D}^{-1}\mathbf{C}\mathbf{A}'\mathbf{B}\mathbf{D}^{-1} \end{aligned}$$

where

$$\mathbf{A}' = (\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}.$$

GP regression

To do *prediction* all that's left is to compute $p(y'|\mathbf{y})$.

Because everything is Gaussian this turns out to be easy:

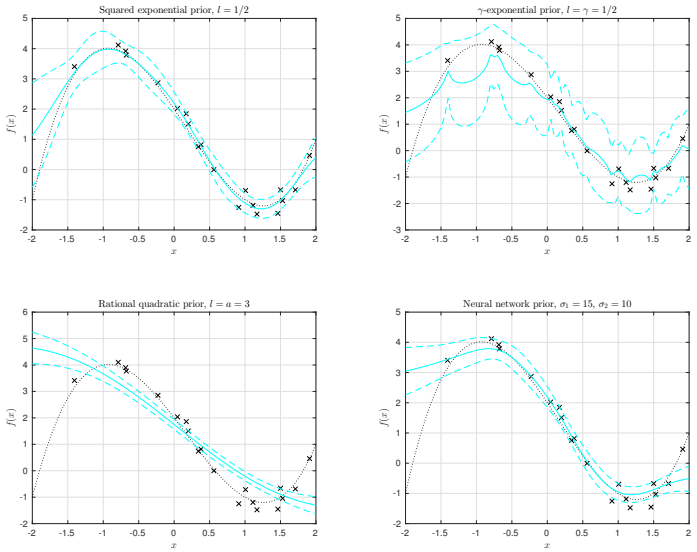
$$\begin{aligned} p(y', \mathbf{y}) &= \mathcal{N}(\mathbf{0}, \mathbf{K}') \\ \mathbf{K}' &= \begin{bmatrix} k & \mathbf{k}^T \\ \mathbf{k} & \mathbf{L} \end{bmatrix} \\ \mathbf{L} &= \mathbf{K} + \sigma^2\mathbf{I}. \end{aligned}$$

From these we want to know $p(y'|\mathbf{y})$.

Only two things are needed: the *inverse formula* for a block matrix and the *formula for obtaining a conditional* from a joint Gaussian. Using these we can show (it is an *Exercise* to derive this) that

$$p(y'|\mathbf{y}) = \mathcal{N}(\underbrace{\mathbf{k}^T\mathbf{L}^{-1}\mathbf{y}}_{\text{Mean}}, \underbrace{k - \mathbf{k}^T\mathbf{L}^{-1}\mathbf{k}}_{\text{Variance}}).$$

GP regression



Learning the hyperparameters

A nice side-effect of this formulation is that we get *a usable expression for the marginal likelihood.*

If we incorporate the hyperparameters $\mathbf{p}$, which in this case are any parameters associated with $k(\mathbf{x}_1, \mathbf{x}_2)$ along with σ^2, then we've just computed

$$p(y'|\mathbf{y}, \mathbf{p}) = \frac{p(y', \mathbf{y}|\mathbf{p})}{p(\mathbf{y}|\mathbf{p})}.$$

The denominator is the marginal likelihood, and we computed it above on *slide 44*:

$$p(\mathbf{y}|\mathbf{p}) = \mathcal{N}(\mathbf{0}, \mathbf{L}) = \frac{1}{\sqrt{(2\pi)^m|L|}} \exp\left(-\frac{1}{2}\mathbf{y}^T\mathbf{L}^{-1}\mathbf{y}\right).$$

Learning the hyperparameters

As usual this looks nicer if we consider its $\log$

$$\log p(\mathbf{y}|\mathbf{p}) = -\frac{1}{2}\log|\mathbf{L}| - \frac{1}{2}\mathbf{y}^T\mathbf{L}^{-1}\mathbf{y} - \frac{m}{2}\log 2\pi.$$

This is a *rare beast*:

1. It's a sensible formula that tells you how good a set $\mathbf{p}$ of hyperparameters is.
2. That means you can use it as an *alternative to cross-validation* to search for hyperparameters.
3. As a bonus *you can generally differentiate it* so it's possible to use *gradient-based search.*

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/

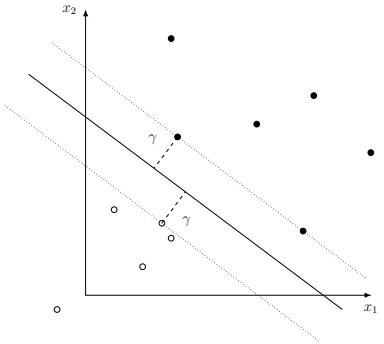
Part II

Support vector machines

General methodology

The maximum margin classifier

Suggestion: why not drop all this probability nonsense and just do this:



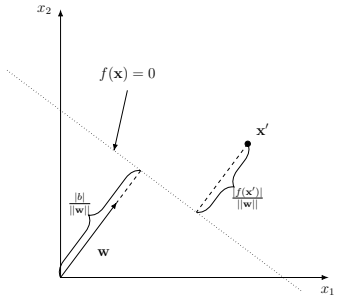
Draw the boundary *as far away from the examples as possible.*

The distance γ is the *margin*, and this is the *maximum margin classifier*.

The maximum margin classifier

If you completed the *exercises for AI I* then you'll know that linear classifiers have a very simple geometry. For

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$



For $\mathbf{x}'$ on one side of the line $f(\mathbf{x}) = 0$ we have $f(\mathbf{x}') > 0$ and on the other side $f(\mathbf{x}') < 0$.

The maximum margin classifier

Problems:

- Given the usual training data $\mathbf{s}$, can we now find a *training algorithm* for obtaining the weights?
- What happens when the data is not *linearly separable*?

To derive the necessary training algorithm we need to know something about *constrained optimization.*

We can address the second issue with a simple modification. This leads to the *Support Vector Machine (SVM).*

Despite being decidedly *"non-Bayesian"* the SVM is currently a *gold-standard*:

Do we need hundreds of classifiers to solve real world classification problems, Fernández-Delgardo at al., Journal of Machine Learning Research 2014.

Constrained optimization

You are familiar with *maximizing* and *minimizing* a function $f(\mathbf{x})$. This is *unconstrained optimization.* We want to extend this:

1. Minimize a function $f(\mathbf{x})$ with the constraint that $g(\mathbf{x}) = 0$.
2. Minimize a function $f(\mathbf{x})$ with the constraints that $g(\mathbf{x}) = 0$ and $h(\mathbf{x}) \geq 0$.

Ultimately we will need to be able to solve problems of the form: find $\mathbf{x}_{\text{opt}}$ such that

$$\mathbf{x}_{\text{opt}} = \underset{\mathbf{x}}{\operatorname{argmin}}\, f(\mathbf{x})$$

under the constraints

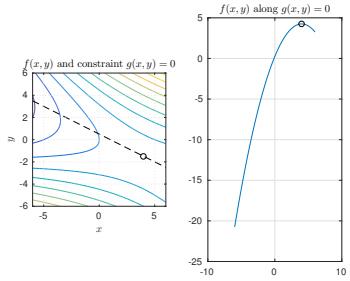
$$g_i(\mathbf{x}) = 0 \text{ for } i = 1, 2, \ldots, n$$

and

$$h_j(\mathbf{x}) \geq 0 \text{ for } j = 1, 2, \ldots, m.$$

Constrained optimization

For example:



Minimize the function

$$f(x, y) = -\left(2x + y^2 + xy\right)$$

subject to the constraint

$$g(x, y) = x + 2y - 1 = 0.$$

Constrained optimization

Step 1: introduce the *Lagrange multiplier* λ and form the *Langrangian*

$$L(x, y, \lambda) = f(x, y) - \lambda g(x, y)$$

Necessary condition: it can be shown that if (x', y') is a solution then $\exists \lambda'$ such that

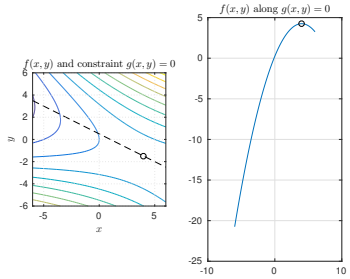
$$\frac{\partial L(x', y', \lambda')}{\partial x} = 0 \qquad \frac{\partial L(x', y', \lambda')}{\partial y} = 0$$

So for our example we need

$$\begin{aligned} 2 + y + \lambda &= 0 \\ 2y + x + 2\lambda &= 0 \\ x + 2y - 1 &= 0 \end{aligned}$$

where the last is just the constraint.

Step 2: solving these equations tells us that the solution is at:



$$(x, y) = (4, -\frac{3}{2})$$

With multiple constraints we follow the same approach, with a *Lagrange multiplier for each constraint*.

How about the full problem? Find

$$\mathbf{x}_{\text{opt}} = \underset{\mathbf{x}}{\text{argmin}}\, f(\mathbf{x}) \text{ such that } g_i(\mathbf{x}) = 0 \text{ for } i = 1, 2, \ldots, n$$
$$h_j(\mathbf{x}) \geq 0 \text{ for } j = 1, 2, \ldots, m$$

The Lagrangian is now

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}) - \sum_{i=1}^{n} \lambda_i g_i(\mathbf{x}) - \sum_{j=1}^{m} \alpha_j h_j(\mathbf{x})$$

and the relevant necessary conditions are more numerous.

The necessary conditions now require that when $\mathbf{x}'$ is a solution $\exists \boldsymbol{\lambda}', \boldsymbol{\alpha}'$ such that

1. $$\frac{\partial L(\mathbf{x}', \boldsymbol{\lambda}', \boldsymbol{\alpha}')}{\partial \mathbf{x}} = 0.$$
2. The equality and inequality constraints are satisfied at $\mathbf{x}'$.
3. $\boldsymbol{\alpha}' \geq \mathbf{0}$.
4. $\alpha_j' h_j(\mathbf{x}') = 0$ for $j = 1, \ldots, m$.

These are called the *Karush-Kuhn-Tucker (KKT) conditions.*

The *KKT conditions* tell us some important things about the solution.

We will only need to address this problem when the constraints are *all inequalities.*

What we've seem so far is called the *primal problem.*

There is also a *dual* version of the problem. Simplifying a little by dropping the equality constraints.

1. The *dual objective function* is
$$\tilde{L}(\boldsymbol{\alpha}) = \inf_{\mathbf{x}} L(\mathbf{x}, \boldsymbol{\alpha}).$$
2. The *dual optimization problem* is
$$\max_{\boldsymbol{\alpha}} \tilde{L}(\alpha) \text{ such that } \boldsymbol{\alpha} \geq \mathbf{0}.$$

Sometimes it is *easier to work by solving the dual problem* and this allows us to obtain actual learning algorithms.

We won't be looking in detail at methods for solving such problems, only the *minimum needed to see how SVMs work.*

For the full story see *Numerical Optimization*, Jorge Nocedal and Stephen J. Wright, Second Edition, Springer 2006.

The maximum margin classifier

It turns out that with SVMs we get particular benefits when using the *kernel trick*.

So we work, as before, in the *extended space*, but now with:

$$f_{\mathbf{w},w_0}(\mathbf{x}) = w_0 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x})$$
$$h_{\mathbf{w},w_0}(\mathbf{x}) = \text{sgn}\left(f_{\mathbf{w},w_0}(\mathbf{x})\right)$$

where

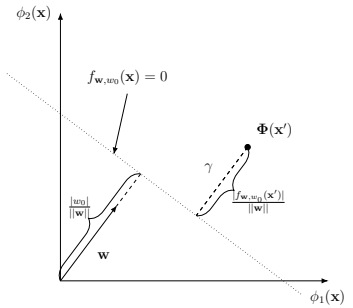
$$\text{sgn}(z) = \begin{cases} +1 & \text{if } z > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Note the following:

1. Things are easier for SVMs if we use labels $\{+1, -1\}$ for the two classes. (Previously we used $\{0, 1\}$.)
2. It also turns out to be easier if we keep w_0 separate rather than rolling it into $\mathbf{w}$.
3. We now classify using a "hard" threshold sgn, rather than the "soft" threshold σ.

The maximum margin classifier

Consider the geometry again. *Step 1:*



1. We're classifying using the sign of the function

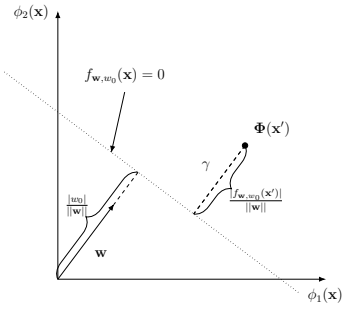
$$f_{\mathbf{w},w_0}(\mathbf{x}) = w_0 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}).$$

2. The distance from any point $\mathbf{\Phi}(\mathbf{x}')$ in the extended space to the line is

$$\frac{|f_{\mathbf{w},w_0}(\mathbf{x}')|}{||\mathbf{w}||}.$$

The maximum margin classifier

Step 2:



- But we also want the examples to fall on the correct *side* of the line according to their *label*.
- *Noting that for any labelled example* $(\mathbf{x}_i, y_i)$ *the quantity* $y_i f_{\mathbf{w},w_0}(\mathbf{x}_i)$ *will be positive* if the resulting classification is correct…
- …the aim is to solve:

$$(\mathbf{w}, w_o) = \underset{\mathbf{w},w_0}{\text{argmax}} \left[\min_i \frac{y_i f_{\mathbf{w},w_0}(\mathbf{x}_i)}{||\mathbf{w}||} \right].$$

The maximum margin classifier

YUK!!!

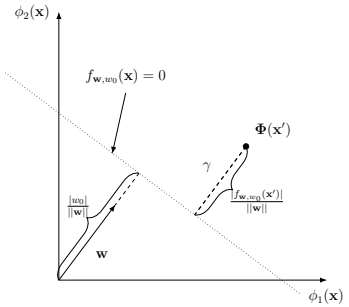
(With bells on…)

The maximum margin classifier

Solution, version 1: convert to a *constrained optimization.* For *any* $c \in \mathbb{R}$

$$f_{\mathbf{w},w_0}(\mathbf{x}) = 0 \iff w_0 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x}) = 0$$
$$\iff cw_0 + c\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}) = 0.$$

That means you can fix $||\mathbf{w}||$ to be *anything you like*! (Actually, fix $||\mathbf{w}||^2$ to avoid a square root.)



Version 1:

$$(\mathbf{w}, w_o, \gamma) = \underset{\mathbf{w},w_0,\gamma}{\operatorname{argmax}}\, \gamma$$

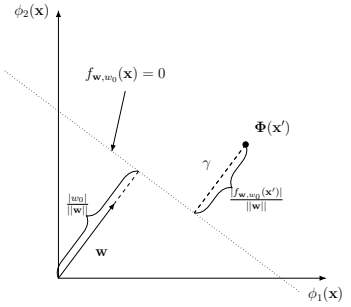
subject to the constraints

$$y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) \geq \gamma, i = 1, 2, \ldots, m$$
$$||\mathbf{w}||^2 = 1.$$

The maximum margin classifier

Solution, version 2: still, convert to a *constrained optimization*, but instead of fixing $||\mathbf{w}||$:

Fix $\min\{y_i f_{\mathbf{w},w_0}(\mathbf{x}_i)\}$ to be *anything you like*!



Version 2:

$$(\mathbf{w}, w_o) = \underset{\mathbf{w},w_0}{\operatorname{argmin}} \frac{1}{2}||\mathbf{w}||^2$$

subject to the constraints

$$y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) \geq 1, i = 1, 2, \ldots, m.$$

(This works because maximizing γ now corresponds to *minimizing* $||\mathbf{w}||$.)

The maximum margin classifier

We'll use the second formulation. (You can work through the first as an *exercise.*)

The *constrained optimization problem* is:

Minimize $\frac{1}{2}||\mathbf{w}||^2$

such that

$$y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) \geq 1 \text{ for } i = 1, \ldots, m\ .$$

Referring back, this means the *Lagrangian* is

$$L(\mathbf{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m} \alpha_i \left(y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) - 1\right)$$

and a *necessary condition* for a solution is that

$$\frac{\partial L(\mathbf{w}, w_0, \boldsymbol{\alpha})}{\partial \mathbf{w}} = 0 \qquad \frac{\partial L(\mathbf{w}, w_0, \boldsymbol{\alpha})}{\partial w_0} = 0.$$

The maximum margin classifier

Working these out is easy:

$$\begin{aligned}\frac{\partial L(\mathbf{w}, w_0, \boldsymbol{\alpha})}{\partial \mathbf{w}} &= \frac{\partial}{\partial \mathbf{w}}\left(\frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m} \alpha_i \left(y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) - 1\right)\right) \\ &= \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \frac{\partial}{\partial \mathbf{w}}\left(\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i) + w_0\right) \\ &= \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}(\mathbf{x}_i)\end{aligned}$$

and

$$\begin{aligned}\frac{\partial L(\mathbf{w}, w_0, \boldsymbol{\alpha})}{\partial w_0} &= -\frac{\partial}{\partial w_0}\left(\sum_{i=1}^{m} \alpha_i y_i f_{\mathbf{w},w_0}(\mathbf{x}_i)\right) \\ &= -\frac{\partial}{\partial w_0}\left(\sum_{i=1}^{m} \alpha_i y_i \left(\mathbf{w}^T\mathbf{\Phi}(\mathbf{x}_i) + w_0\right)\right) \\ &= -\sum_{i=1}^{m} \alpha_i y_i.\end{aligned}$$

Equating those to 0 and adding the *KKT conditions* tells us several things:

1. The weight vector can be expressed as
$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}(\mathbf{x}_i)$$
with $\boldsymbol{\alpha} \geq \mathbf{0}$. This is important: we'll return to it in a moment.
2. There is a constraint that
$$\sum_{i=1}^{m} \alpha_i y_i = 0.$$
This will be needed for working out the *dual Lagrangian.*
3. For each example
$$\alpha_i [y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) - 1] = 0.$$

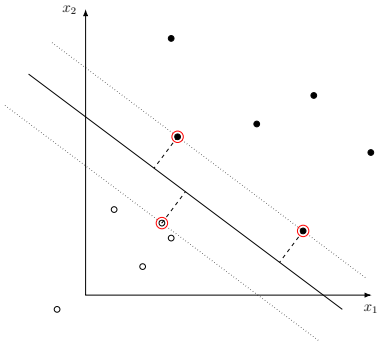
The fact that for each example
$$\alpha_i [y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) - 1] = 0$$
means that:

Either $y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) = 1$ *or* $\alpha_i = 0$.

This means that examples fall into two groups.

1. Those for which $y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) = 1$.
As the contraint used to maximize the margin was $y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) \geq 1$ these are *the examples that are closest to the boundary.*
They are called *support vectors* and they can have *non-zero weights.*
2. Those for which $y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) \neq 1$.
These are non-support vectors *and in this case it must be that* $\alpha_i = 0$.

Support vectors:



1. *Circled examples:* support vectors with $\alpha_i > 0$.
2. *Other examples:* have $\alpha_i = 0$.

Remember that

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}(\mathbf{x}_i).$$

so *the weight vector* $\mathbf{w}$ *only depends on the support vectors.*

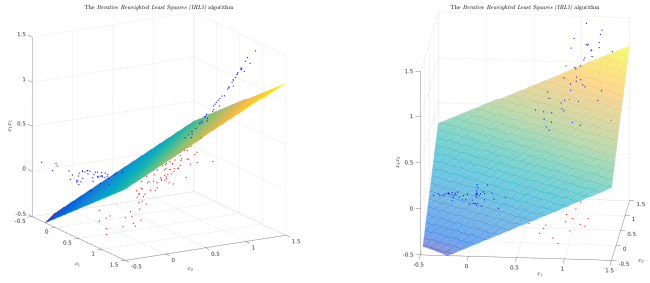
ALSO: the dual parameters $\boldsymbol{\alpha}$ can be used as an *alternative* set of weights. The overall classifier is

$$\begin{aligned} h_{\mathbf{w},w_0}(\mathbf{x}) &= \operatorname{sgn}\left(w_0 + \mathbf{w}^T \mathbf{\Phi}(\mathbf{x})\right) \\ &= \operatorname{sgn}\left(w_0 + \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}^T(\mathbf{x}_i)\mathbf{\Phi}(\mathbf{x})\right) \\ &= \operatorname{sgn}\left(w_0 + \sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right) \end{aligned}$$

where $K(\mathbf{x}_i, \mathbf{x}) = \mathbf{\Phi}^T(\mathbf{x}_i)\mathbf{\Phi}(\mathbf{x})$ is called the *kernel.*

The maximum margin classifier

Remember where this process started:



The kernel is computing

$$\begin{aligned} K(\mathbf{x}, \mathbf{x}') &= \mathbf{\Phi}^T(\mathbf{x})\mathbf{\Phi}(\mathbf{x}') \\ &= \sum_{i=1}^{k} \phi_i(\mathbf{x})\phi_i(\mathbf{x}') \end{aligned}$$

This is generally called an *inner product*.

The maximum margin classifier

If it's a *hard problem* then you'll probably want *lots of basis functions* so k *is BIG:*

$$\begin{aligned} h_{\mathbf{w},w_0}(\mathbf{x}) &= \text{sgn}\left(w_0 + \mathbf{w}^T\mathbf{\Phi}(\mathbf{x})\right) \\ &= \text{sgn}\left(w_0 + \sum_{i=1}^{k} w_i\phi_i(\mathbf{x})\right) \\ &= \text{sgn}\left(w_0 + \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}^T(\mathbf{x}_i)\mathbf{\Phi}(\mathbf{x})\right) \\ &= \text{sgn}\left(w_0 + \sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right) \end{aligned}$$

What if $K(\mathbf{x}, \mathbf{x}')$ is easy to compute even if k is *HUGE*? (In particular $k >> m$.)

1. We get a definite computational advantage by using the dual version with weights $\boldsymbol{\alpha}$.
2. *Mercer's theorem* tells us exactly when a function K has a corresponding set of *basis functions* $\{\phi_i\}$.

The maximum margin classifier

Designing good kernels K is a subject in itself.

Luckily *for the majority of the time* you will tend to see one of the following:

1. *Polynomial:*
$$K_{c,d}(\mathbf{x}, \mathbf{x}') = (c + \mathbf{x}^T\mathbf{x}')^d$$
where c and d are parameters.
2. *Radial basis function (RBF):*
$$K_{\sigma^2}(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2\sigma^2}||\mathbf{x} - \mathbf{x}'||^2\right)$$
where σ^2 is a parameter.

The last is particularly prominent. Interestingly, the corresponding set of basis functions is *infinite*. (So we get an improvement in computational complexity from infinite to *linear in the number of examples*!)

Maximum margin classifier: the dual version

Collecting together some of the results up to now:

1. The Lagrangian is
$$L(\mathbf{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_i \alpha_i(y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) - 1).$$
2. The weight vector is
$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{\Phi}(\mathbf{x}_i).$$
3. The KKT conditions require
$$\sum_i \alpha_i y_i = 0.$$

It's easy to show (this is an *exercise*) that the *dual optimization problem* is to maximize

$$\tilde{L}(\boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

such that $\boldsymbol{\alpha} \geq \mathbf{0}$.

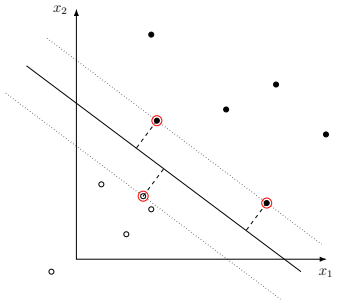
Support Vector Machines

There is one thing still missing:

Problem: so far we've only covered the *linearly separable* case.

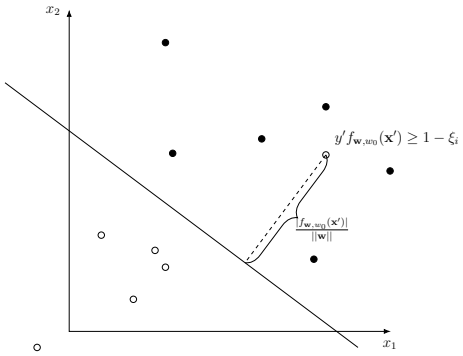
Even though that means linearly separable *in the extended space* it's still not enough.

By dealing with this we get the *Support Vector Machine (SVM).*



Support Vector Machines

Fortunately a small modification allows us to let *some* examples be misclassified.



We introduce the *slack variables* ξ_i, one for *each example.*

Although $y' f_{\mathbf{w},w_0}(\mathbf{x}') < 0$ we have $y' f_{\mathbf{w},w_0}(\mathbf{x}') \geq 1 - \xi_i$ and we try to force ξ_i to be small.

Support Vector Machines

The *constrained optimization problem* was:

$\text{argmin}_{\mathbf{w},w_0} \frac{1}{2}||\mathbf{w}||^2$ such that $y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) \geq 1$ for $i = 1, \ldots, m$.

The *constrained optimization problem* is now modified to:

$$\underset{\mathbf{w},w_0,\boldsymbol{\xi}}{\text{argmin}} \underbrace{\frac{1}{2}||\mathbf{w}||^2}_{\text{Maximize the margin}} + \underbrace{C\sum_{i=1}^{m}\xi_i}_{\text{Control misclassification}}$$

such that

$$y_i f_{\mathbf{w},w_0}(\mathbf{x}_i) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \text{ for } i = 1, \ldots, m.$$

There is a *further new parameter* C that controls the trade-off between *maximizing the margin* and *controlling misclassification.*

Support Vector Machines

Once again, the theory of *constrained optimization* can be employed:

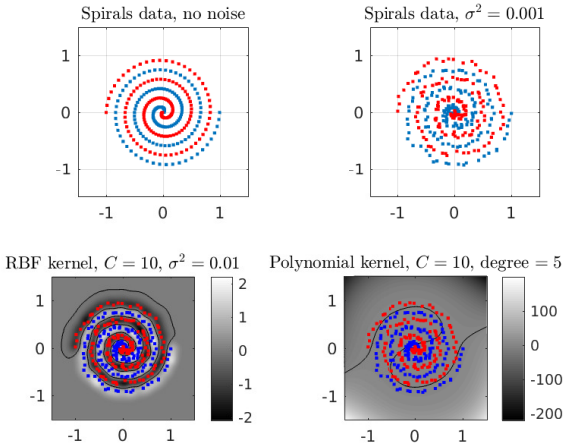
1. We get the *same insights* into the solution of the problem, and the *same conclusions.*
2. The development is exactly analogous to what we've just seen.

However as is often the case it is not straightforward to move all the way to having a functioning training algorithm.

For this some attention to good *numerical computing* is required. See:

Fast training of support vector machine using sequential minimal optimization, J. C. Platt, *Advances in Kernel Methods*, MIT Press 1999.

Support Vector Machines



Supervised learning in practice

We now look at several issues that need to be considered when *applying machine learning algorithms in practice*:

- We often have more examples from some classes than from others.
- The *obvious* measure of performance is not always the *best*.
- Much as we'd love to have an optimal method for *finding hyperparameters*, we don't have one, and it's *unlikely that we ever will*.
- We need to exercise care if we want to claim that one approach is superior to another.

This part of the course has an *unusually large number of Commandments*.

That's because *so many people get so much of it wrong!*.

Supervised learning

As usual, we want to design a *classifier*.

It should take an attribute vector

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

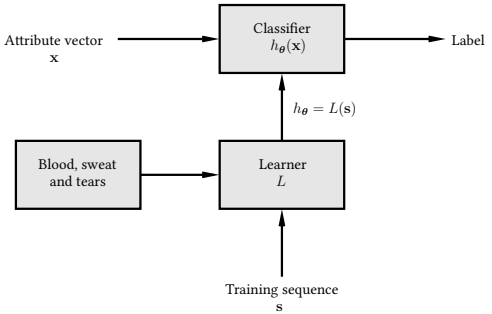
and label it.

We now denote a classifier by $h_{\boldsymbol{\theta}}(\mathbf{x})$ where $\boldsymbol{\theta}^T = \begin{pmatrix} \mathbf{w} & \mathbf{p} \end{pmatrix}$ denotes any weights $\mathbf{w}$ and (hyper)parameters $\mathbf{p}$.

To keep the discussion and notation simple we assume a *classification problem* with *two classes* labelled $+1$ *(positive examples)* and -1 *(negative examples)*.

Supervised learning

Previously, the learning algorithm was a box labelled L.



Unfortunately that turns out not to be enough, so *a new box has been added*.

Machine Learning Commandments

We've already come across the Commandment:

> Thou shalt *try a simple method*. Preferably *many* simple methods.

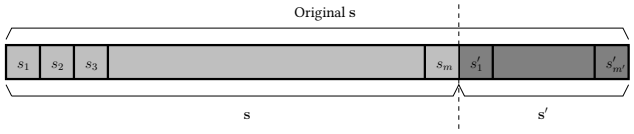
Now we will add:

> Thou shalt use an *appropriate measure of performance*.

Measuring performance

How do you assess the performance of your classifier?

1. That is, *after training*, how do you know how well you've done?
2. In general, the only way to do this is to divide your examples into a smaller *training set* $\mathbf{s}$ of m examples and a *test set* $\mathbf{s}'$ of m' examples.



> The *GOLDEN RULE*: *data used to assess performance must NEVER have been seen during training*.

This might seem obvious, but it was a major flaw in a lot of early work.

Measuring performance

How do we choose m and m'? Trial and error!

Assume the training is complete, and we have a classifier $h_{\boldsymbol{\theta}}$ obtained using only $\mathbf{s}$. How do we use $\mathbf{s}'$ to assess our method's performance?

The obvious way is to see how many examples in $\mathbf{s}'$ the classifier classifies correctly:

$$\hat{\text{er}}_{\mathbf{s}'}(h_{\boldsymbol{\theta}}) = \frac{1}{m'} \sum_{i=1}^{m'} \mathbb{I}\left[h_{\boldsymbol{\theta}}(\mathbf{x}'_i) \neq y'_i\right]$$

where

$$\mathbf{s}' = \left[(\mathbf{x}'_1, y'_1) \;\; (\mathbf{x}'_2, y'_2) \;\; \cdots \;\; (\mathbf{x}'_{m'}, y'_{m'})\right]^T$$

and

$$\mathbb{I}[z] = \begin{cases} 1 & \text{if } z = \text{true} \\ 0 & \text{if } z = \text{false} \end{cases}.$$

This is just an estimate of the *probability of error* and is often called the *accuracy*.

Unbalanced data

Unfortunately it is often the case that we have *unbalanced data* and this can make such a measure misleading. For example:

> If the data is naturally such that *almost all examples are negative* (medical diagnosis for instance) then simply *classifying everything as negative* gives a high performance using this measure.

We need more subtle measures.

For a classifier h and any set $\mathbf{s}$ of size m containing m^+ positive examples and m^- negative examples...

Unbalanced data

Define

1. The *true positives*
$$P^+ = \{(\mathbf{x}, +1) \in \mathbf{s} | h(\mathbf{x}) = +1\}, \text{ and } p^+ = |P^+|$$
2. The *false positives*
$$P^- = \{(\mathbf{x}, -1) \in \mathbf{s} | h(\mathbf{x}) = +1\}, \text{ and } p^- = |P^-|$$
3. The *true negatives*
$$N^+ = \{(\mathbf{x}, -1) \in \mathbf{s} | h(\mathbf{x}) = -1\}, \text{ and } n^+ = |N^+|$$
4. The *false negatives*
$$N^- = \{(\mathbf{x}, +1) \in \mathbf{s} | h(\mathbf{x}) = -1\}, \text{ and } n^- = |N^-|$$

Thus $\hat{\text{er}}_{\mathbf{s}}(h) = (p^+ + n^+)/m$.

This allows us to define more discriminating measures of performance.

Performance measures

Some standard performance measures:

1. Precision/Positive predictive value (PPV) $\frac{p^+}{p^+ + p^-}$.
2. Negative predictive value (NPR) $\frac{n^+}{n^+ + n^-}$.
3. Recall/Sensitivity/True positive rate (TPR) $\frac{p^+}{p^+ + n^-}$.
4. Specificity/True negative rate (TNR) $\frac{n^+}{n^+ + p^-}$.
5. False positive rate (FPR) $\frac{p^-}{p^- + n^+}$.
6. False negative rate (FNR) $\frac{n^-}{n^- + p^+}$
7. False discovery rate $\frac{p^-}{p^- + p^+}$.

In addition, plotting sensitivity (true positive rate) against the false positive rate while a parameter is varied gives the *receiver operating characteristic (ROC)* curve.

Performance measures

The following specifically take account of unbalanced data:

1. Matthews Correlation Coefficient (MCC)
$$\text{MCC} = \frac{p^+ n^+ - p^- n^-}{\sqrt{(p^+ + p^-)(n^+ + n^-)(p^+ + n^-)(n^+ + p^-)}}$$
2. F1 score
$$\text{F1} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

When data is unbalanced these are preferred over the accuracy.

Machine Learning Commandments

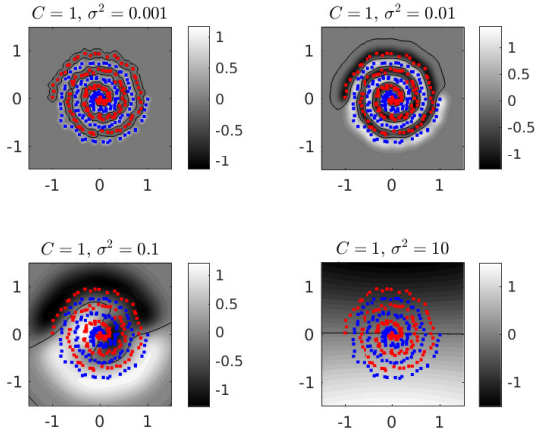
Thou shalt not use *default parameters*.

Thou shalt not use parameters chosen by an *unprincipled formula*.

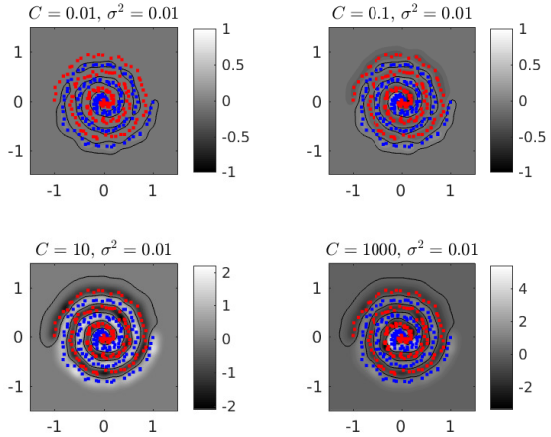
Thou shalt not avoid this issue by clicking on 'Learn' and *hoping it works*.

Thou shalt either *choose them carefully* or *integrate them out*.

Bad hyperparameters give bad performance



Bad hyperparameters give bad performance



Validation and crossvalidation

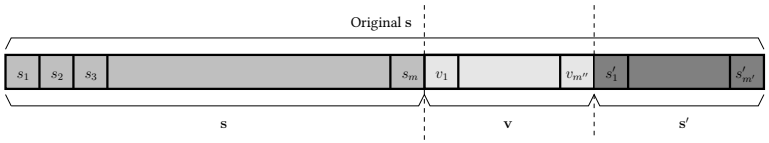
The next question: how do we choose hyperparameters?

Answer: *try different values and see which values give the best (estimated) performance.*

There is however a problem:

If I use my test set $\mathbf{s}'$ to find good hyperparameters, *then I can't use it to get a final measure of performance.* (See the Golden Rule above.)

Solution 1: make a further division of the complete set of examples to obtain a third, *validation* set:



Validation and crossvalidation

Now, to choose the value of a hyperparameter p:

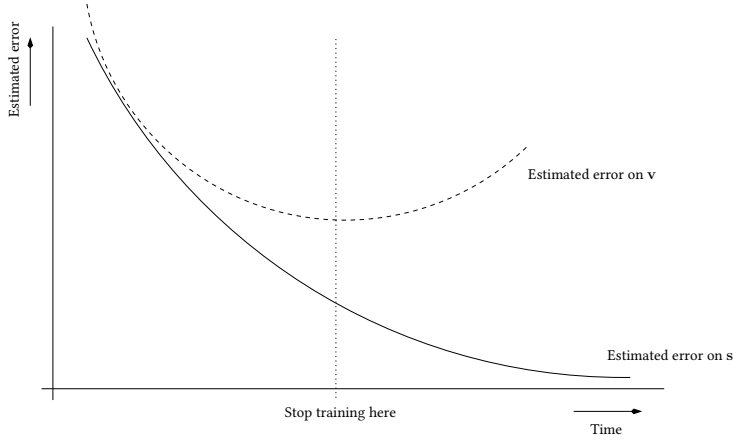
For some range of values $p_1, p_2, \ldots, p_n$

1. Run the training algorithm using training data $\mathbf{s}$ and with the hyperparameter set to p_i.
2. Assess the resulting $h_{\boldsymbol{\theta}}$ by computing a suitable measure (for example accuracy, MCC or F1) using $\mathbf{v}$.

Finally, select the $h_{\boldsymbol{\theta}}$ with maximum estimated performance and assess its *actual* performance using $\mathbf{s}'$.

Validation and crossvalidation

This was originally used in a similar way when deciding the best point at which to *stop training* a neural network.



The figure shows the typical scenario.

Crossvalidation

The method of *crossvalidation* takes this a step further.

We our complete set into training set $\mathbf{s}$ and testing set $\mathbf{s}'$ as before.

But now instead of further subdividing $\mathbf{s}$ just once we divide it into n *folds* $\mathbf{s}^{(i)}$ each having m/n examples.

Typically $n = 10$ although other values are also used, for example if $n = m$ we have *leave-one-out* cross-validation.

Crossvalidation

Let $\mathbf{s}_{-i}$ denote the set obtained from $\mathbf{s}$ by *removing* $\mathbf{s}^{(i)}$.

Let $\hat{\mathbf{er}}_{\mathbf{s}^{(i)}}(h)$ denote any suitable error measure, such as accuracy, MCC or F1, computed for h using fold i.

Let $L_{\mathbf{s}_{-i},\mathbf{p}}$ be the classifier obtained by running learning algorithm L on examples $\mathbf{s}_{-i}$ using hyperparameters p.

Then,

$$\frac{1}{n}\sum_{i=1}^{n}\hat{\mathbf{er}}_{\mathbf{s}^{(i)}}(L_{\mathbf{s}_{-i},\mathbf{p}})$$

is the *n-fold crossvalidation error estimate*.

So for example, let $\mathbf{s}_j^{(i)}$ denote the jth example in the ith fold. Then using accuracy as the error estimate we have

$$\frac{1}{m}\sum_{i=1}^{n}\sum_{j=1}^{m/n}\mathbb{I}\left[L_{\mathbf{s}_{-i},\mathbf{p}}(\mathbf{x}_j^{(i)}) \neq y_j^{(i)}\right]$$

Crossvalidation

Two further points:

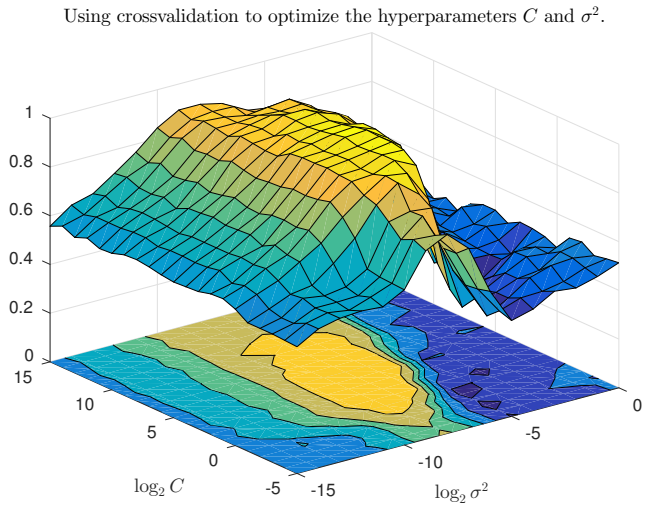
1. What if the data are unbalanced? *Stratified crossvalidation* chooses folds such that the proportion of positive examples in each fold matches that in $\mathbf{s}$.
2. Hyperparameter choice can be done just as above, using a basic search.

What happens however if we have multiple hyperparameters?

1. We can search over all combinations of values for specified ranges of each parameter.
2. This is the *standard method in choosing parameters for support vector machines (SVMs)*.
3. With SVMs it is generally limited to the case of only two hyperparameters.
4. Larger numbers quickly become infeasible.

Crossvalidation

This is what we get for an *SVM* applied to the *two spirals*:



Machine Learning Commandments

Thou shalt *provide evidence* before claiming that *thy method is the best*.

Thou shalt take extra notice of this Commandment if *thou considers thyself a True And Pure Bayesian.*

Thou shalt even take notice of this Commandment if *thou considers thyself to be DEEP*.

Comparing classifiers

Imagine I have compared the *AIMLBlockChain Classificator* and the *DeepHype Discriminotron* and found that:

1. The Classificator has estimated accuracy 0.981 on the test set.
2. The Discriminotron has estimated accuracy 0.982 on the test set.

Can I claim that the Discriminotron is the better classifier?

Answer:

NO! NO! NO! NO! NO! NO! NO! NO! NO!!!!!!!!!!

Comparing classifiers

NO!!!!!!

Note for next year: include photo of grumpy-looking cat.

Assessing a single classifier

From *Mathematical Methods for Computer Science*:

The *Central Limit Theorem*: If we have independent identically distributed (iid) random variables $X_1, X_2, \ldots, X_n$ with mean

$$\mathbb{E}[X] = \mu$$

and variance

$$\mathbb{E}\left[(X-\mu)^2\right] = \sigma^2$$

then as $n \to \infty$

$$\frac{\hat{X}_n - \mu}{\sigma/\sqrt{n}} \to N(0,1)$$

where

$$\hat{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

Assessing a single classifier

We have tables of values z_p such that if $x \sim N(0,1)$ then

$$\Pr(-z_p \leq x \leq z_p) > p.$$

Rearranging this using the equation from the previous slide we have that with probability p

$$\mu \in \left[\hat{X}_n \pm z_p\sqrt{\frac{\sigma^2}{n}}\right].$$

We don't know σ^2 but it can be estimated using

$$\sigma^2 \simeq \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \hat{X}_n\right)^2.$$

Alternatively, when X takes only values 0 or 1

$$\sigma^2 = \mathbb{E}\left[(X-\mu)^2\right] = \mathbb{E}\left[X^2\right] - \mu^2 = \mu(1-\mu) \simeq \hat{X}_n(1-\hat{X}_n).$$

Assessing a single classifier

The *actual probability of error* for a classifier h is

$$\mathbf{er}(h) = \mathbb{E}\left[\mathbb{I}\left[h(\mathbf{x}) \neq y\right]\right]$$

and we are *estimating* $\mathbf{er}(h)$ using the *accuracy*

$$\hat{\mathbf{er}}_{\mathbf{s}}(h) = \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}\left[h(\mathbf{x}_i) \neq y_i\right]$$

for a test set $\mathbf{s}$.

We can find a confidence interval for this estimate using precisely the derivation above, simply by noting that the X_i are the random variables

$$X_i = \mathbb{I}\left[h(\mathbf{x}_i) \neq y_i\right].$$

Assessing a single classifier

Typically we are interested in a 95% confidence interval, for which $z_p = 1.96$.

Thus, when $m > 30$ (so that the central limit theorem applies) we know that, with probability 0.95

$$\boxed{\mathbf{er}(h) = \hat{\mathbf{er}}_{\mathbf{s}}(h) \pm 1.96\sqrt{\frac{\hat{\mathbf{er}}_{\mathbf{s}}(h)(1-\hat{\mathbf{er}}_{\mathbf{s}}(h)))}{m}}}.$$

Example: I have 100 test examples and my classifier makes 18 errors. With probability 0.95 I know that

$$\begin{aligned}\mathbf{er}(h) &= 0.18 \pm 1.96\sqrt{\frac{0.18(1-0.18)}{100}} \\ &= 0.18 \pm 0.075.\end{aligned}$$

This should perhaps *raise an alarm* regarding our suggested comparison of classifiers above.

Assessing a single classifier

There is an important distinction to be made here:

1. The *mean of* X is μ and the *variance of* X is σ^2.
2. We can also ask about the mean and variance of $\hat{X}_n$.
3. The *mean of* $\hat{X}_n$ is

$$\begin{aligned}\mathbb{E}\left[\hat{X}_n\right] &= \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] \\ &= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\left[X_i\right] \\ &= \mu.\end{aligned}$$

4. It is left as an *exercise* to show that the *variance of* $\hat{X}_n$ is

$$\sigma^2_{\hat{X}_n} = \frac{\sigma^2}{n}.$$

Comparing classifiers

We are using the values z_p such that if $x \sim N(0,1)$ then

$$\Pr(-z_p \leq x \leq z_p) > p.$$

There is an *alternative* way to think about this.

1. Say we have a random variable Y with variance σ_Y^2 and mean μ_Y.
2. The random variable $Y - \mu_Y$ has variance σ_Y^2 and mean 0.
3. It is a straightforward exercise to show that dividing a random variable having variance σ^2 by σ gives us a new random variable with variance 1.
4. Thus the random variable $\frac{Y-\mu_Y}{\sigma_Y}$ has mean 0 and variance 1.

So: with probability p

$$Y = \mu_Y \pm z_p\sigma_Y$$
$$\mu_Y = Y \pm z_p\sigma_Y.$$

Compare this with what we saw earlier. *You need to be careful to keep track of whether you are considering the mean and variance of a single RV or a sum of RVs.*

Comparing classifiers

Now say I have classifiers h_1 *(Bloggs Classificator 2000)* and h_2 *(CleverCorp Discriminotron)* and I want to know something about the quantity

$$d = \mathrm{er}(h_1) - \mathrm{er}(h_2).$$

I estimate d using

$$\hat{d} = \hat{\mathrm{er}}_{\mathbf{s}_1}(h_1) - \hat{\mathrm{er}}_{\mathbf{s}_2}(h_2)$$

where $\mathbf{s}_1$ and $\mathbf{s}_2$ are *two* independent test sets.

Notice:

1. The estimate of d is a sum of random variables, and *we can apply the central limit theorem.*
2. The estimate is *unbiased*

$$\mathbb{E}\left[\hat{\mathrm{er}}_{\mathbf{s}_1}(h_1) - \hat{\mathrm{er}}_{\mathbf{s}_2}(h_2)\right] = d.$$

Comparing classifiers

Also notice:

1. The two parts of the estimate $\hat{\mathrm{er}}_{\mathbf{s}_1}(h_1)$ and $\hat{\mathrm{er}}_{\mathbf{s}_2}(h_2)$ are each sums of random variables and *we can apply the central limit theorem to each.*
2. The variance of the estimate is the sum of the variances of $\hat{\mathrm{er}}_{\mathbf{s}_1}(h_1)$ and $\hat{\mathrm{er}}_{\mathbf{s}_2}(h_2)$.
3. Adding Gaussians gives another Gaussian.
4. *We can calculate a confidence interval for our estimate.*

With probability 0.95

$$\boxed{d = \hat{d} \pm 1.96\sqrt{\frac{\hat{\mathrm{er}}_{\mathbf{s}_1}(h_1)(1-\hat{\mathrm{er}}_{\mathbf{s}_1}(h_1))}{m_1} + \frac{\hat{\mathrm{er}}_{\mathbf{s}_2}(h_2)(1-\hat{\mathrm{er}}_{\mathbf{s}_2}(h_2))}{m_2}}}.$$

In fact, if we are using a split into training set $\mathbf{s}$ and test set $\mathbf{s}'$ we can generally obtain h_1 and h_2 using $\mathbf{s}$ and use the estimate

$$\hat{d} = \hat{\mathrm{er}}_{\mathbf{s}'}(h_1) - \hat{\mathrm{er}}_{\mathbf{s}'}(h_2).$$

Comparing classifiers—hypothesis testing

This still doesn't tell us directly about *whether one classifier is better than another*—whether h_1 is better than h_2.

What we actually want to know is whether

$$d = \mathrm{er}(h_1) - \mathrm{er}(h_2) > 0.$$

Say we've measured $\hat{D} = \hat{d}$. Then:

- Imagine the *actual value* of d is 0.
- Recall that the *mean* of $\hat{D}$ is d.
- So *larger* measured values $\hat{d}$ are *less likely*, even though some random variation is inevitable.
- If it is highly *unlikely* that when $d = 0$ a measured value of $\hat{d}$ would be observed, then we can be confident that $d > 0$.
- Thus we are interested in
$$\Pr(\hat{D} > d + \hat{d}).$$

This is known as a *one-sided bound*.

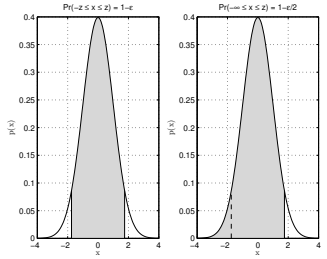
One-sided bounds

Given the *two-sided bound*

$$\Pr(-z_\epsilon \leq x \leq z_\epsilon) = 1 - \epsilon$$

we actually need to know the *one-sided bound*

$$\Pr(x \leq z_\epsilon).$$



Clearly, if our random variable is *Gaussian* then $\Pr(x \leq z_\epsilon) = 1 - \epsilon/2$.

Comparing algorithms: paired t-tests

We now know how to compare *hypotheses* h_1 and h_2.

But we still haven't properly addressed the comparison of *algorithms*.

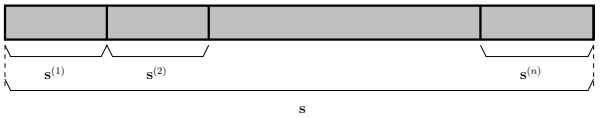
- Remember, a learning algorithm L maps training data $\mathbf{s}$ to hypothesis h.
- So we *really* want to know about the quantity
$$d = \mathbb{E}_{\mathbf{s}\in S^m}\left[\mathrm{er}(L_1(\mathbf{s})) - \mathrm{er}(L_2(\mathbf{s}))\right].$$
- This is the *expected difference* between the *actual errors* of the *two different* algorithms L_1 and L_2 .

Unfortunately, we have *only one set of data* $\mathbf{s}$ available and we *can only estimate* errors $\mathrm{er}(h)$—we don't have access to the *actual quantities*.

We can however use the idea of *crossvalidation*.

Comparing algorithms: paired t-tests

Recall, we subdivide $\mathbf{s}$ into n *folds* $\mathbf{s}^{(i)}$ each having m/n examples



and denote by $\mathbf{s}_{-i}$ the set obtained from $\mathbf{s}$ by *removing* $\mathbf{s}^{(i)}$. Then

$$\frac{1}{n}\sum_{i=1}^{n} \hat{\mathrm{er}}_{\mathbf{s}^{(i)}}(L(\mathbf{s}_{-i}))$$

is the n*-fold crossvalidation error estimate*. Now we estimate d using

$$\hat{d} = \frac{1}{n}\sum_{i=1}^{n}\left[\hat{\mathrm{er}}_{\mathbf{s}^{(i)}}(L_1(\mathbf{s}_{-i})) - \hat{\mathrm{er}}_{\mathbf{s}^{(i)}}(L_2(\mathbf{s}_{-i}))\right].$$

Comparing algorithms: paired t-tests

As usual, there is a *statistical test* allowing us to assess *how likely this estimate is to mislead us*.

We will not consider the derivation in detail. With probability p

$$d \in \left[\hat{d} \pm t_{p,n-1}\sigma_{\hat{d}}\right].$$

This is analogous to the equations seen above, however:

- The parameter $t_{p,n-1}$ is analogous to z_p.
- The parameter $t_{p,n-1}$ is related to the area under the *Student's t-distribution* whereas z_p is related to the area under the normal distribution.
- The relevant estimate of *standard deviation* is

$$\sigma_{\hat{d}} = \sqrt{\frac{1}{n(n-1)}\sum_{i=1}^{n}\left(d_i - \hat{d}\right)^2}$$

where

$$d_i = \hat{\mathrm{er}}_{\mathbf{s}^{(i)}}(L_1(\mathbf{s}_{-i})) - \hat{\mathrm{er}}_{\mathbf{s}^{(i)}}(L_2(\mathbf{s}_{-i})).$$

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/

Part III: back to Bayes

Bayesian neural networks

Gaussian processes

Where now?

There are some simple *take-home messages* from the study of SVMs:

You can get *state-of-the-art* performance.

You can do this using the *kernel trick* to obtain a *non-linear model*.

You can do this without invoking the *full machinery of the Bayes-optimal classifier*.

BUT:

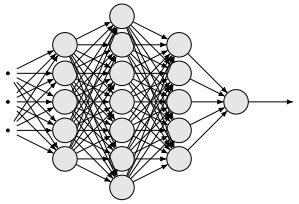
You don't have anything *keeping you honest* regarding *which assumptions you're making*.

As we shall see, by using the *full-strength probabilistic framework* we gain some useful extras.

In particular, the ability to *assign confidences* to our predictions.

The Bayesian approach to neural networks

We're now going to see how the idea of the *Bayes-optimal classifier* can be applied to *neural networks*.



We have:

- A *neural network* computing a function $h_{\mathbf{w}}(\mathbf{x})$. (In fact this can be *pretty much any* parameterized function we like.)
- A training sequence $\mathbf{s}^T = \left[(\mathbf{x}_1, y_1) \ \ldots \ (\mathbf{x}_m, y_m)\right]$, split into

$$\mathbf{y} = (\ y_1 \ \ y_2 \ \cdots \ \ y_m\ )$$

and

$$\mathbf{X} = (\ \mathbf{x}_1 \ \ \mathbf{x}_2 \ \cdots \ \ \mathbf{x}_m\ ).$$

The Bayesian approach to neural networks

We're *only going to consider regression.* Classification can also be done this way, but it's a bit more complicated.

For *classification* we derived the Bayes-optimal classifier as the *maximizer* of:

$$\Pr(C|\mathbf{x},\mathbf{s}) = \int \Pr(C|\mathbf{w},\mathbf{x})\, p(\mathbf{w}|\mathbf{s})\, d\mathbf{w}$$

For regression the *Bayes-optimal classifier* ends up having the same expression as we've already seen. We want to compute:

$$p(Y|\mathbf{x},\mathbf{s}) = \int \underbrace{p(Y|\mathbf{w},\mathbf{x})}_{\text{Likelihood}} \underbrace{p(\mathbf{w}|\mathbf{s})}_{\text{Posterior}}\, d\mathbf{w}$$

$\mathbf{s}$ is the *training set*.

$\mathbf{x}$ is a *new example* to be classified.

Y is the RV representing the *prediction* for $\mathbf{x}$.

The Bayesian approach to neural networks

It turns out that *if you try to incorporate* the density $p(\mathbf{x})$ modelling how *feature vectors* are generated, things can get complicated. So:

1. We regard all input vectors as *fixed*: they are *not* treated as random variables.
2. This means that, *strictly speaking*, they should no longer appear in expressions like $p(Y|\mathbf{w},\mathbf{x})$.
3. However, this seems to be uniformly disliked—writing $p(Y|\mathbf{w})$ for an expression that still depends on $\mathbf{x}$ seems confusing.
4. Solution: write $p(Y|\mathbf{w};\mathbf{x})$ instead. Note the *semi-colon*!

So we're actually going to look at

$$p(Y|\mathbf{y};\mathbf{x},\mathbf{X}) = \int \underbrace{p(Y|\mathbf{w};\mathbf{x})}_{\text{Likelihood}} \underbrace{p(\mathbf{w}|\mathbf{y};\mathbf{X})}_{\text{Posterior}}\, d\mathbf{w}$$

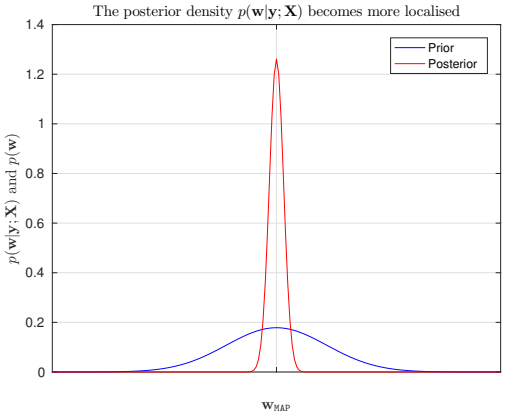
NOTE: this is a *notational hack*. There's nothing new, just an attempt at clarity.

What's going on? Turning prior into posterior

Let's make a brief sidetrack into what's going on with the posterior density

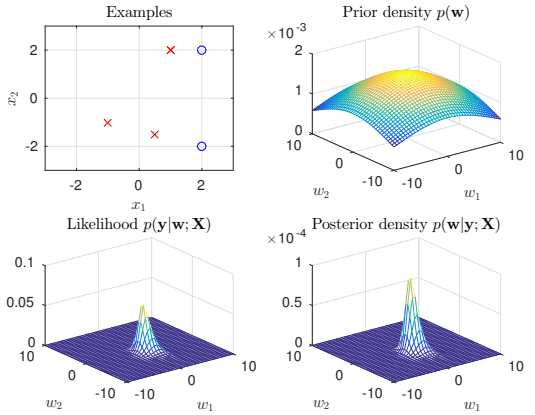
$$p(\mathbf{w}|\mathbf{y};\mathbf{X}) \propto p(\mathbf{y}|\mathbf{w};\mathbf{X})p(\mathbf{w}).$$

Typically, the *prior starts wide* and as we see more data the *posterior narrows*



What's going on? Turning prior into posterior

This can be seen very clearly if we use real numbers:



The Bayesian approach to neural networks

So now we have three things to do:

1. *STEP 1:* remind ourselves what $p(Y|\mathbf{w};\mathbf{x})$ is.
2. *STEP 2:* remind ourselves what $p(\mathbf{w}|\mathbf{y};\mathbf{X})$ is.
3. *STEP 3:* do the integral. (*This is the fun bit...*)

The first two steps are straightforward as *we've already derived them* when looking at *maximum-likelihood* and *MAP* learning.

The Bayesian approach to neural networks

STEP 1: assuming Gaussian noise is added to the labels so

$$y = h_{\mathbf{w}}(\mathbf{x}) + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, \sigma_n^2)$ we have the usual likelihood

$$p(Y|\mathbf{w};\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma_n^2}} \exp\left(-\frac{1}{2\sigma_n^2}\left(Y - h_{\mathbf{w}}(\mathbf{x})\right)^2\right).$$

Here, the subscript in σ_n^2 reminds us that it's the variance of the *noise*.

Traditionally this is re-written using the *hyperparameter*

$$\beta = \frac{1}{\sigma_n^2}$$

so the *likelihood* is

$$p(Y|\mathbf{w};\mathbf{x}) \propto \exp\left(-\frac{\beta}{2}\left(Y - h_{\mathbf{w}}(\mathbf{x})\right)^2\right).$$

The Bayesian approach to neural networks

STEP 2: the *posterior* is also exactly as it was when we derived the *MAP* learning algorithms.

$$p(\mathbf{w}|\mathbf{y};\mathbf{X}) \propto p(\mathbf{y}|\mathbf{w};\mathbf{X})p(\mathbf{w})$$

and as before, the likelihood is

$$p(\mathbf{y}|\mathbf{w};\mathbf{X}) \propto \exp\left(-\frac{\beta}{2}\sum_{i=1}^{m}\left(y_i - h_{\mathbf{w}}(\mathbf{x}_i)\right)^2\right)$$
$$= \exp\left(-\beta E(\mathbf{w})\right)$$

and using a Gaussian prior with mean $\mathbf{0}$ and covariance $\mathbf{\Sigma} = \sigma^2\mathbf{I}$ gives

$$p(\mathbf{w}) \propto \exp\left(-\frac{\alpha}{2}||\mathbf{w}||^2\right)$$

where traditionally the second *hyperparameter* is $\alpha = 1/\sigma^2$. Combining these

$$p(\mathbf{w}|\mathbf{y};\mathbf{X}) = \frac{1}{Z(\alpha,\beta)} \exp\left(-\left(\frac{\alpha||\mathbf{w}||^2}{2} + \beta E(\mathbf{w})\right)\right).$$

What's going on? Turning prior into posterior

Considering the central part of $p(\mathbf{w}|\mathbf{y};\mathbf{X})$:

$$\frac{\alpha||\mathbf{w}||^2}{2} + \beta E(\mathbf{w}).$$

What happens as the number m of examples increases?

- The first term *corresponding to the prior* remains fixed.
- The second term *corresponding to the likelihood* increases.

So for small training sequences the prior dominates, but for large ones $\mathbf{w}_{\text{ML}}$ is a good approximation to $\mathbf{w}_{\text{MAP}}$.

The Bayesian approach to neural networks

Step 3: putting together steps 1 and 2, the integral we need to evaluate is:

$$I \propto \int \underbrace{\exp\left(-\frac{\beta}{2}(Y - h_{\mathbf{w}}(\mathbf{x}))^2\right)}_{\text{Likelihood}} \underbrace{\exp\left(-\left(\frac{\alpha||\mathbf{w}||^2}{2} + \beta E(\mathbf{w})\right)\right)}_{\text{Posterior}} d\mathbf{w}.$$

Obviously this *gives us all a sad face* because there is *no solution.*

So what can we do now…?

The Bayesian approach to neural networks

In order to make further progress it's necessary to perform integrals of the general form

$$\int F(\mathbf{w}) p(\mathbf{w}|\mathbf{y}; \mathbf{X})\, d\mathbf{w}$$

for various functions F and this is generally not possible.

There are two ways to get around this:

1. We can use an *approximate form* for $p(\mathbf{w}|\mathbf{y}; \mathbf{X})$.
2. We can use *Monte Carlo* methods.

We'll be taking a look at both possibilities.

Method 1: approximation to $p(\mathbf{w}|\mathbf{y}; \mathbf{X})$

$$I \propto \int \underbrace{\exp\left(-\frac{\beta}{2}(Y - h_{\mathbf{w}}(\mathbf{x}))^2\right)}_{\text{Likelihood } p(Y|\mathbf{w};\mathbf{x})} \underbrace{\exp\left(-\left(\frac{\alpha||\mathbf{w}||^2}{2} + \beta E(\mathbf{w})\right)\right)}_{\text{Posterior } p(\mathbf{w}|\mathbf{y};\mathbf{X})} d\mathbf{w}.$$

The first approach introduces a *Gaussian approximation* to $p(\mathbf{w}|\mathbf{y}; \mathbf{X})$ by using a *Taylor expansion* of

$$S(\mathbf{w}) = \frac{\alpha||\mathbf{w}||^2}{2} + \beta E(\mathbf{w})$$

at the *maximum a posteriori* weights $\mathbf{w}_{\text{MAP}}$.

This allows us to use a *standard integral.*

The result will be *approximate* but we hope it's good!

Let's recall how Taylor series work…

Reminder: Taylor expansion

In *one dimension* the *Taylor expansion* about a point $x_0 \in \mathbb{R}$ for a function $f : \mathbb{R} \to \mathbb{R}$ is

$$\begin{aligned} f(x) \approx f(x_0) &+ \frac{1}{1!}(x - x_0) f'(x_0) \\ &+ \frac{1}{2!}(x - x_0)^2 f''(x_0) \\ &+ \cdots + \frac{1}{k!}(x - x_0)^k f^k(x_0). \end{aligned}$$

What does this look like for the kinds of function we're interested in? As an *example* We can try to approximate

$$\exp\left(-f(x)\right)$$

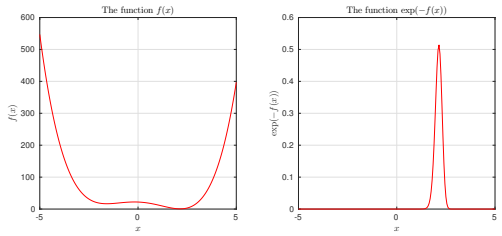
where

$$f(x) = x^4 - \frac{1}{2}x^3 - 7x^2 - \frac{5}{2}x + 22.$$

This has a *form similar to* $S(\mathbf{w})$, but in one dimension.

Reminder: Taylor expansion

The functions of interest look like this:



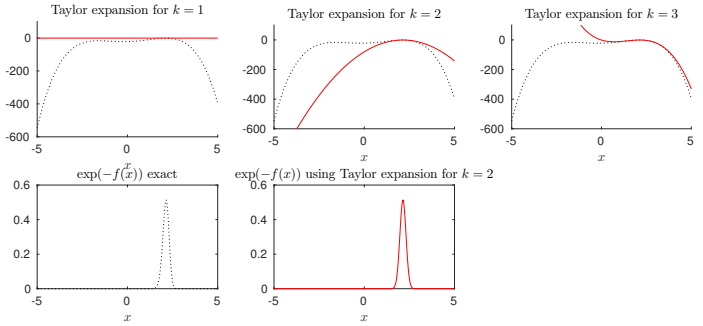
By replacing $-f(x)$ with its *Taylor expansion about its maximum*, which is at

$$x_{\max} = 2.1437$$

we can see what the *approximation to* $\exp(-f(x))$ looks like. Note that the $\exp$ *hugely emphasises peaks.*

Reminder: Taylor expansion

Here are the approximations for $k = 1$, $k = 2$ and $k = 3$.



The use of $k = 2$ looks promising...

Reminder: Taylor expansion

In *multiple dimensions* the Taylor expansion for $k = 2$ is

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \frac{1}{1!}(\mathbf{x} - \mathbf{x}_0)^T \nabla f(\mathbf{x})\big|_{\mathbf{x}_0} + \frac{1}{2!}(\mathbf{x} - \mathbf{x}_0)^T \nabla^2 f(\mathbf{x})\big|_{\mathbf{x}_0} (\mathbf{x} - \mathbf{x}_0)$$

where ∇ denotes *gradient*

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f(\mathbf{x})}{\partial x_1} \quad \frac{\partial f(\mathbf{x})}{\partial x_2} \quad \cdots \quad \frac{\partial f(\mathbf{x})}{\partial x_n} \right)$$

and $\nabla^2 f(\mathbf{x})$ is the matrix with elements

$$M_{ij} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}$$

(Looks complicated, but it's just the obvious extension of the 1-dimensional case.)

Method 1: approximation to $p(\mathbf{w}|\mathbf{y}; \mathbf{X})$

Applying this to $S(\mathbf{w})$ and expanding around $\mathbf{w}_{\text{MAP}}$

$$S(\mathbf{w}) = \frac{\alpha||\mathbf{w}||^2}{2} + \beta E(\mathbf{w}) \approx S(\mathbf{w}_{\text{MAP}}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_{\text{MAP}})^T \mathbf{A} (\mathbf{w} - \mathbf{w}_{\text{MAP}}).$$

- As $\mathbf{w}_{\text{MAP}}$ *minimises* the function the first derivatives are zero and the corresponding term in the Taylor expansion *disappears.*
- The quantity $\mathbf{A} = \nabla\nabla S(\mathbf{w})|_{\mathbf{w}_{\text{MAP}}}$ can be simplified.

This is because

$$\mathbf{A} = \nabla\nabla \left(\frac{\alpha||\mathbf{w}||^2}{2} + \beta E(\mathbf{w}) \right)\Big|_{\mathbf{w}_{\text{MAP}}} = \alpha \mathbf{I} + \beta \nabla\nabla E(\mathbf{w}_{\text{MAP}}).$$

Method 1: approximation to $p(\mathbf{w}|\mathbf{y};\mathbf{X})$

We actually already know something about how to get $\mathbf{w}_{\text{MAP}}$:

1. A method such as *backpropagation* can be used to compute $\nabla S(\mathbf{w})$.
2. The vector $\mathbf{w}_{\text{MAP}}$ can then be obtained using any standard optimisation method (such as *gradient descent*).

It's also likely to be straightforward to compute $\nabla\nabla E(\mathbf{w})$:

The quantity $\nabla\nabla E(\mathbf{w})$ can be evaluated using an *extended form of backpropagation.*

A useful integral

Dropping *for this slide only* the special meaning usually given to the vector $\mathbf{x}$, here is a useful standard integral:

If $\mathbf{A} \in \mathbb{R}^{n\times n}$ is symmetric then for $\mathbf{b} \in \mathbb{R}^n$ and $c \in \mathbb{R}$

$$\int_{\mathbb{R}^n} \exp\left(-\frac{1}{2}\left(\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{x}^T\mathbf{b} + c\right)\right) d\mathbf{x} = (2\pi)^{n/2}|\mathbf{A}|^{-1/2}\exp\left(-\frac{1}{2}\left(c - \frac{\mathbf{b}^T\mathbf{A}^{-1}\mathbf{b}}{4}\right)\right).$$

You're not expected to know how to evaluate this, but *see the handout on the course web page* if you're curious[1].

To make this easy to refer to, let's call it the *BIG INTEGRAL*.

[1] No, I won't ask you to evaluate it in the exam…

Method 1: approximation to $p(\mathbf{w}|\mathbf{y};\mathbf{X})$

Defining

$$\Delta\mathbf{w} = \mathbf{w} - \mathbf{w}_{\text{MAP}}$$

we now have an approximation

$$p(\mathbf{w}|\mathbf{y};\mathbf{X}) \approx \frac{1}{Z}\exp\left(-S(\mathbf{w}_{\text{MAP}}) - \frac{1}{2}\Delta\mathbf{w}^T\mathbf{A}\Delta\mathbf{w}\right).$$

Using the *BIG INTEGRAL*

$$Z = (2\pi)^{W/2}|\mathbf{A}|^{-1/2}\exp(-S(\mathbf{w}_{\text{MAP}}))$$

where W is the number of weights.

Let's plug this approximation back into the *expression for the Bayes-optimum* and see what we get…

Method 1: approximation to $p(\mathbf{w}|\mathbf{y};\mathbf{X})$

$$I \propto \int \underbrace{\exp\left(-\frac{\beta}{2}\left(Y - h_{\mathbf{w}}(\mathbf{x})\right)^2\right)}_{\text{Likelihood } p(Y|\mathbf{w};\mathbf{x})} \underbrace{\exp\left(-\frac{1}{2}\Delta\mathbf{w}^T\mathbf{A}\Delta\mathbf{w}\right)}_{\text{Approximation to } p(\mathbf{w}|\mathbf{y};\mathbf{X})} d\mathbf{w}.$$

There is still *no solution!* We need *another approximation…*

We can introduce a *linear approximation*[2] of $h_{\mathbf{w}}(\mathbf{x})$ at $\mathbf{w}_{\text{MAP}}$:

$$h_{\mathbf{w}}(\mathbf{x}) \approx h_{\mathbf{w}_{\text{MAP}}}(\mathbf{x}) + \mathbf{g}^T\Delta\mathbf{w}$$

where $\mathbf{g} = \nabla h_{\mathbf{w}}(\mathbf{x})\big|_{\mathbf{w}_{\text{MAP}}}$.

(By linear approximation we just mean the Taylor expansion for $k = 1$.)

[2] We really are making assumptions here—this is OK if we assume that $p(\mathbf{w}|\mathbf{y};\mathbf{X})$ is *narrow*, which depends on $\mathbf{A}$.

Method 1: second approximation

This leads to

$$p(Y|\mathbf{y};\mathbf{x},\mathbf{X}) \propto \int \exp\left(-\frac{\beta}{2}\left(Y - h_{\mathbf{w}_{\text{MAP}}}(\mathbf{x}) - \mathbf{g}^T\Delta\mathbf{w}\right)^2 - \frac{1}{2}\Delta\mathbf{w}^T\mathbf{A}\Delta\mathbf{w}\right) d\mathbf{w}.$$

SUCCESS!!!

This integral can be evaluated (this is an *exercise*) using the *BIG INTEGRAL* to give *THE ANSWER...*

$$p(Y|\mathbf{y};\mathbf{x},\mathbf{X}) \simeq \frac{1}{\sqrt{2\pi\sigma_Y^2}} \exp\left(-\frac{(Y - h_{\mathbf{w}_{\text{MAP}}}(\mathbf{x}))^2}{2\sigma_Y^2}\right)$$

where

$$\sigma_Y^2 = \frac{1}{\beta} + \mathbf{g}^T\mathbf{A}^{-1}\mathbf{g}.$$

Method 1: final expression

Hooray! But what does it mean?

This is a *Gaussian density*, so we can now see that:

$p(Y|\mathbf{y};\mathbf{x},\mathbf{X})$ *peaks* at $h_{\mathbf{w}_{\text{MAP}}}(\mathbf{x})$.

That is, the *MAP solution.*

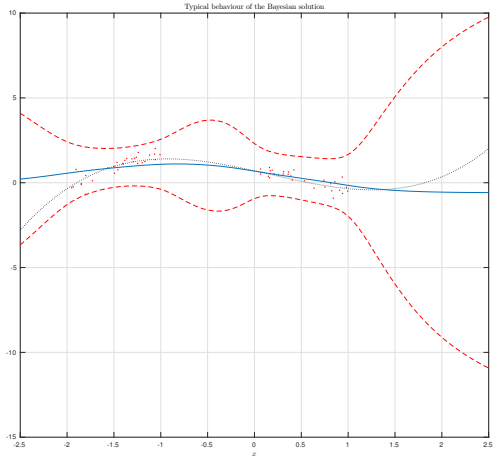
The *variance* σ_Y^2 can be interpreted as a measure of *certainty*:

The first term of σ_Y^2 is $1/\beta$ and corresponds to the noise.

The second term of σ_Y^2 is $\mathbf{g}^T\mathbf{A}^{-1}\mathbf{g}$ and corresponds to the width of $p(\mathbf{w}|\mathbf{y};\mathbf{X})$.

Method 1: final expression

Hooray! But what does it mean? Interpreted graphically:



Plotting $\pm 2\sigma_Y$ around the prediction gives a *measure of certainty*.

Method II: Markov chain Monte Carlo (MCMC) methods

The second solution to the problem of performing integrals

$$I = \int F(\mathbf{w})p(\mathbf{w}|\mathbf{y};\mathbf{X})d\mathbf{w}$$

is to use *Monte Carlo* methods. The basic approach is to make the approximation

$$I \approx \frac{1}{N}\sum_{i=1}^{N} F(\mathbf{w}_i)$$

where the $\mathbf{w}_i$ have distribution $p(\mathbf{w}|\mathbf{y};\mathbf{X})$. Unfortunately, generating $\mathbf{w}_i$ with a *given distribution* can be non-trivial.

MCMC methods

A simple technique is to introduce a random walk, so

$$\mathbf{w}_{i+1} = \mathbf{w}_i + \epsilon$$

where ϵ is *zero mean spherical Gaussian* and has *small variance*. Obviously the sequence $\mathbf{w}_i$ does not have the required distribution. However, we can use the *Metropolis algorithm*, which does *not* accept all the steps in the random walk:

1. If $p(\mathbf{w}_{i+1}|\mathbf{y};\mathbf{X}) > p(\mathbf{w}_i|\mathbf{y};\mathbf{X})$ then accept the step.
2. Else accept the step with probability $\frac{p(\mathbf{w}_{i+1}|\mathbf{y};\mathbf{X})}{p(\mathbf{w}_i|\mathbf{y};\mathbf{X})}$.

In practice, the Metropolis algorithm has several shortcomings, and a great deal of research exists on improved methods, see:

R. Neal, "Probabilistic inference using Markov chain Monte Carlo methods," University of Toronto, Department of Computer Science Technical Report CRG-TR-93-1, 1993.

A (very) brief introduction to how to learn hyperparameters

So far in our coverage of the Bayesian approach to neural networks, the *hyperparameters* α and β were assumed to be known and fixed.

- But this is not a good assumption because...
- ...α corresponds to the width of the prior and β to the noise variance.
- So we really want to learn these from the data as well.
- How can this be done?

We now take a look at one of several ways of addressing this problem.

Note: from now on I'm going to leave out the dependencies on $\mathbf{x}$ and $\mathbf{X}$ as leaving them in starts to make everything cluttered.

The Bayesian approach to neural networks

The prior and likelihood depend on α and β respectively so we now make this clear and write

$$p(\mathbf{w}|\mathbf{y},\alpha,\beta) = \frac{p(\mathbf{y}|\mathbf{w},\beta)p(\mathbf{w}|\alpha)}{p(\mathbf{y}|\alpha,\beta)}.$$

Don't worry about recalling the *actual expressions* for the prior and likelihood—we're not going to delve deep enough to need them.

Let's write down directly something that might be useful to know:

$$p(\alpha,\beta|\mathbf{y}) = \frac{p(\mathbf{y}|\alpha,\beta)p(\alpha,\beta)}{p(\mathbf{y})}.$$

Hierarchical Bayes and the evidence

If we know $p(\alpha,\beta|\mathbf{y})$ then a straightforward approach is to *use the values for* α *and* β *that maximise it*:

$$\operatorname*{argmax}_{\alpha,\beta} p(\alpha,\beta|\mathbf{y}).$$

Here is a standard trick: *assume that the prior* $p(\alpha,\beta)$ *is flat*, so that we can just maximise

$$p(\mathbf{y}|\alpha,\beta).$$

This is called *type II maximum likelihood* and is one common way of doing the job.

Hierarchical Bayes and the evidence

The quantity

$$p(\mathbf{y}|\alpha, \beta)$$

is called the *evidence* or *marginal likelihood*.

When we re-wrote our earlier equation for the posterior density of the weights, making α and β explicit, we found

$$p(\mathbf{w}|\mathbf{y}, \alpha, \beta) = \frac{p(\mathbf{y}|\mathbf{w}, \beta)p(\mathbf{w}|\alpha)}{p(\mathbf{y}|\alpha, \beta)}.$$

So *the evidence is the denominator in this equation.*

This is the *common pattern* and leads to the idea of *hierarchical Bayes*: the *evidence for the hyperparameters* at one level is the *denominator in the relevant application of Bayes' theorem.*

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/

Part IV

Unsupervised learning

Machine Learning and Bayesian Inference

The next major subject:

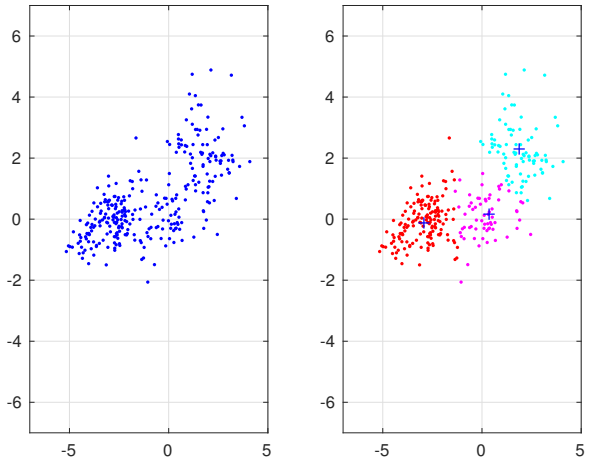
Unsupervised learning

In which we see that

- We can learn from *unlabelled* data. This kind of learning is often known as *clustering*.
- We can do this using a simple, obvious algorithm known as K*-means*.
- We can also approach it *probabilistically* using *maximum likelihood*.
- This is less straightforward, but there is a general algorithm called *Expectation Maximization (EM)* that can be applied.

Unsupervised learning

Can we find *regularity in data* without the aid of *labels*?



Is this *one cluster*? Or *three*? Or some other number?

The K-means algorithm

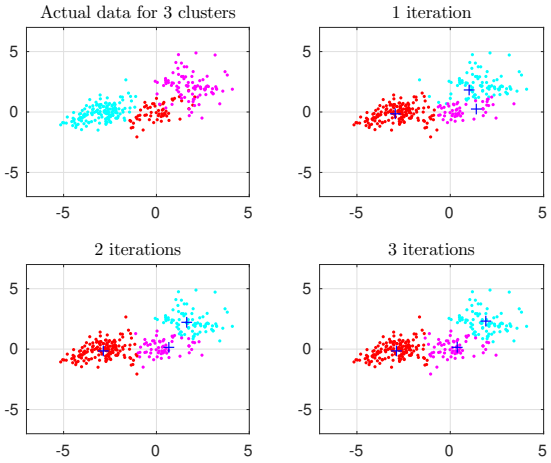
The example on the last slide was obtained using the classical *K-means algorithm*.

Given a set $\{\mathbf{x}_i\}$ of m points, guess that there are K clusters. Here $K = 3$.

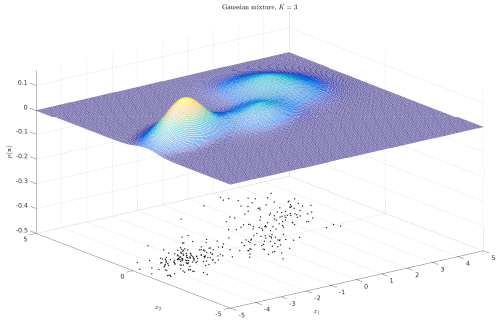
Chose at random K centre points $\mathbf{c}_j$ for the clusters. Then *iterate as follows*:

1. Divide $\{\mathbf{x}_i\}$ into K clusters, so *each point is associated with the closest centre*:
$$\mathbf{x}_i \in C_j \iff \forall k \; ||\mathbf{x}_i - \mathbf{c}_j|| \leq ||\mathbf{x}_i - \mathbf{c}_k||.$$
Call these clusters $C_1, \ldots, C_K$.
2. Update the cluster centres to be the *average of the associated points*:
$$\mathbf{c}_j = \frac{1}{|C_j|} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i.$$

The K-means algorithm



Clustering as maximum-likelihood

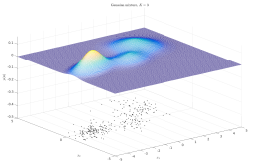


We saw in the introductory lectures that data from K clusters can be modelled probabilistically as

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

where $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_K\}$ and typically $p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

Clustering as maximum-likelihood



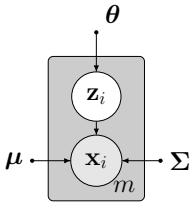
This leads to a log-likelihood for m points of

$$\begin{aligned}\log p(\mathbf{X}|\boldsymbol{\theta}) &= \log \prod_{i=1}^{m} p(\mathbf{x}_i|\boldsymbol{\theta}) \\ &= \sum_{i=1}^{n} \log p(\mathbf{x}_i|\boldsymbol{\theta}) \\ &= \sum_{i=1}^{n} \log \sum_{k=1}^{K} \pi_k p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\end{aligned}$$

which tends to be hard to maximise directly. (You can find stationary points but they depend on one-another.)

Clustering as maximum-likelihood

We can however introduce some *latent variables*.



For each $\mathbf{x}_i$ introduce the latent variable $\mathbf{z}_i$ where

$$\mathbf{z}_i^T = \begin{bmatrix} z_i^{(1)} & \cdots & z_i^{(K)} \end{bmatrix}$$

and

$$z_i^{(j)} = \begin{cases} 1 & \text{if } \mathbf{x}_i \text{ was generated by cluster } j \\ 0 & \text{otherwise} \end{cases}$$

Clustering as maximum-likelihood

Having introduced the $\mathbf{z}_i$ we can use the marginalization trick and write

$$\begin{aligned} \log p(\mathbf{X}|\boldsymbol{\theta}) &= \log \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) \\ &= \log \sum_{\mathbf{Z}} p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\theta}) p(\mathbf{Z}|\boldsymbol{\theta}) \end{aligned}$$

where the final step has given us probabilities that are reasonably tractable.

Why is this?

First, if I know *which cluster* generated $\mathbf{x}$ then its probability is just that for the *corresponding Gaussian*

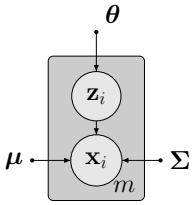
$$p(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}) = \prod_{k=1}^{K} \left[p(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right]^{z_i^{(k)}}$$

and similarly

$$p(\mathbf{z}|\boldsymbol{\theta}) = \prod_{k=1}^{K} \left[\pi_k \right]^{z_i^{(k)}}$$

Clustering as maximum-likelihood

In other words, if you treat the $\mathbf{z}_i$ as *observed* rather than *latent*



then you can write

$$p(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta}) = \prod_{k=1}^{K} \left[p(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \pi_k \right]^{z_i^{(k)}}$$

$$\begin{aligned} \log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) &= \log \prod_{i=1}^{m} p(\mathbf{x}_i, \mathbf{z}_i|\boldsymbol{\theta}) \\ &= \log \prod_{i=1}^{m} \prod_{k=1}^{K} \left[p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \pi_k \right]^{z_i^{(k)}} \end{aligned}$$

Clustering as maximum-likelihood

Consequently

$$\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) = \sum_{i=1}^{m} \sum_{k=1}^{K} \mathbf{z}_i^{(k)} \left(\log p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) + \log \pi_k \right)$$

What have we achieved so far?

1. We want to *maximize the log-likelihood* $\log p(\mathbf{X}|\boldsymbol{\theta})$ but this is intractable.
2. We introduce some *latent variables* $\mathbf{Z}$.
3. That gives us a *tractable* log-likelihood $\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$.

But how do we link them together?

The EM algorithm

The *Expectation Maximization (EM)* algorithm provides a general way of maximizing likelihood for problems like this.

Let's do something a little strange. Let $q(\mathbf{Z})$ be *any* distribution on the *latent variables*. Write

$$\begin{aligned}\sum_{\mathbf{Z}} q(\mathbf{Z}) \log \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} &= \sum_{\mathbf{Z}} q(\mathbf{Z}) \log \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}) p(\mathbf{X}|\boldsymbol{\theta})}{q(\mathbf{Z})} \\ &= \sum_{\mathbf{Z}} q(\mathbf{Z}) \left(\log \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})} + \log p(\mathbf{X}|\boldsymbol{\theta}) \right) \\ &= -D_{KL}[q(\mathbf{Z})||p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})] + \sum_{\mathbf{Z}} q(\mathbf{Z}) \log p(\mathbf{X}|\boldsymbol{\theta}) \\ &= -D_{KL}[q(\mathbf{Z})||p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})] + \log p(\mathbf{X}|\boldsymbol{\theta})\end{aligned}$$

D_{KL} is the *Kullback-Leibler (KL) distance*.

The Kullback-Leibler (KL) distance

The *Kullback-Leibler (KL) distance* measures the distance between two probability distributions. For discrete distributions p and q it is

$$D_{\mathrm{KL}}[p||q] = \sum_{\mathbf{x}} p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})}.$$

It has the important properties that:

1. It is non-negative
$$D_{\mathrm{KL}}(p||q) \geq 0.$$
2. It is 0 precisely when the distributions are equal
$$D_{\mathrm{KL}}[p||q] = 0 \text{ if and only if } p = q.$$

The EM algorithm

If we also define

$$L[q, \boldsymbol{\theta}] = \sum_{\mathbf{Z}} q(\mathbf{Z}) \log \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})}$$

then we can re-arrange the last expression to get

$$\log p(\mathbf{X}|\boldsymbol{\theta}) = L[q, \boldsymbol{\theta}] + D_{KL}[q||p]$$

and we know that $D_{KL}[q||p] \geq 0$ so that gives us an upper bound

$$L[q, \boldsymbol{\theta}] \leq \log p(\mathbf{X}|\boldsymbol{\theta}).$$

The EM algorithm works as follows:

- We iteratively maximize $L[q, \boldsymbol{\theta}]$.
- We do this by alternately maximizing with respect to q and $\boldsymbol{\theta}$ while keeping the other fixed.
- Maximizing with respect to q is the *E step*.
- Maximizing with respect to $\boldsymbol{\theta}$ is the *M step*.

The EM algorithm

Let's look at the two steps separately.

Say we have $\boldsymbol{\theta}_t$ at time t in the iteration.

For the *E step*, we have $\boldsymbol{\theta}_t$ fixed and

$$\log p(\mathbf{X}|\boldsymbol{\theta}_t) = L[q, \boldsymbol{\theta}_t] + D_{KL}[q||p]$$

so this is easy!

1. As $\boldsymbol{\theta}_t$ is fixed, so is $\log p(\mathbf{X}|\boldsymbol{\theta}_t)$.
2. So to maximize $L[q, \boldsymbol{\theta}_t]$ we must minimize $D_{KL}[q||p]$.
3. And we know that $D_{KL}[q||p]$ is minimized and equal to 0 when $q = p$.

So in the E step we just choose

$$q_{t+1}(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}_t).$$

For the *M step* we have

$$L[q, \boldsymbol{\theta}] = \sum_{\mathbf{Z}} q(\mathbf{Z}) \log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) - \sum_{\mathbf{Z}} q(\mathbf{Z}) \log q(\mathbf{Z})$$

where the second term (the entropy of $q(\mathbf{Z})$) doesn't depend on $\boldsymbol{\theta}$.

We fix $q_{t+1}(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}_t)$. We now choose $\boldsymbol{\theta}_{t+1}$ as

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}_t) \log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}_t) \\ &= \operatorname*{argmax}_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{Z}} \left[\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}_t)\right]\end{aligned}$$

We saw earlier that

$$\log p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}_t) = \sum_{i=1}^{m} \sum_{k=1}^{K} \mathbf{z}_i^{(k)} \left(\log p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) + \log \pi_k\right)$$

where $\boldsymbol{\theta}$ collects all the parameters

$$\boldsymbol{\theta}_t = \{\boldsymbol{\pi}, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_K\}.$$

Note that the parameters $\boldsymbol{\pi}$, $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}_i$ all have an implicit time t attached, but we avoid writing it to keep the notation managable.

So: this step looks a little tricker: we need to maximize the expected value of this expression for the distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}_t)$.

It's not as bad as it looks:

- Take the expected value inside the sums.
- The *only* part of the expression that depends on $\mathbf{Z}$ is $\mathbf{z}_i^{(k)}$.
- So we *only* have to compute $\mathbb{E}_{\mathbf{Z}}\left[\mathbf{z}_i^{(k)}\right]$.

Thus

$$\begin{aligned}\mathbb{E}_{\mathbf{Z}}\left[\mathbf{z}_i^{(k)}\right] &= \sum_{\mathbf{Z}} \mathbf{z}_i^{(k)} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}_t) \\ &= \sum_{\mathbf{z}_1} \cdots \sum_{\mathbf{z}_m} \mathbf{z}_i^{(k)} p(\mathbf{z}_1, \ldots, \mathbf{z}_m|\mathbf{X}, \boldsymbol{\theta}_t) \\ &= \sum_{\mathbf{z}_i} \mathbf{z}_i^{(k)} p(\mathbf{z}_i|\mathbf{X}, \boldsymbol{\theta}_t) \text{ (marginalizing)} \\ &= \sum_{\mathbf{z}_i^{(k)} \in \{0,1\}} \mathbf{z}_i^{(k)} p(\mathbf{z}_i^{(k)}|\mathbf{X}, \boldsymbol{\theta}_t) \text{ (marginalizing again)} \\ &= p(\mathbf{z}_i^{(k)} = 1|\mathbf{X}, \boldsymbol{\theta}_t)\end{aligned}$$

So

$$\begin{aligned}\mathbb{E}_{\mathbf{Z}}\left[\mathbf{z}_i^{(k)}\right] &= p(\mathbf{z}_i^{(k)} = 1|\mathbf{X}, \boldsymbol{\theta}_t) \\ &= \frac{p(\mathbf{z}_i^{(k)} = 1, \mathbf{x}_i|\boldsymbol{\theta}_t)}{p(\mathbf{x}_i|\boldsymbol{\theta})} \text{ using conditional independence} \\ &= \frac{\pi_k p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k=1}^{K} \pi_k p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}\end{aligned}$$

As a shorthand, define

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

so the expression we've arrived at is

$$\boldsymbol{\theta}_{t+1} = \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{i=1}^{m} \sum_{k=1}^{K} \gamma_i^{(k)} \left(\log p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) + \log \pi_k\right)$$

The EM algorithm

The EM algorithm for a mixture model summarized:

- We want to find $\boldsymbol{\theta}$ to maximize $\log p(\mathbf{X}|\boldsymbol{\theta})$.
- But that's not tractable.
- So we introduce an arbitrary distribution q and obtain a lower bound

$$L(q, \boldsymbol{\theta}) \leq \log p(\mathbf{X}|\boldsymbol{\theta}).$$

- We maximize the lower bound iteratively in two steps:
  1. *E step:* keep $\boldsymbol{\theta}$ fixed and maximize with respect to q. This always results in $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$.
  2. *M step:* keep q fixed and maximize with respect to $\boldsymbol{\theta}$. For the mixture model this is

$$\boldsymbol{\theta}_{t+1} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{m} \sum_{k=1}^{K} \gamma_i^{(k)} \left(\log p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) + \log \pi_k\right)$$

The EM algorithm for a mixture of Gaussians

We leave the derivation of the rest of the *M Step* as an *exercise.*

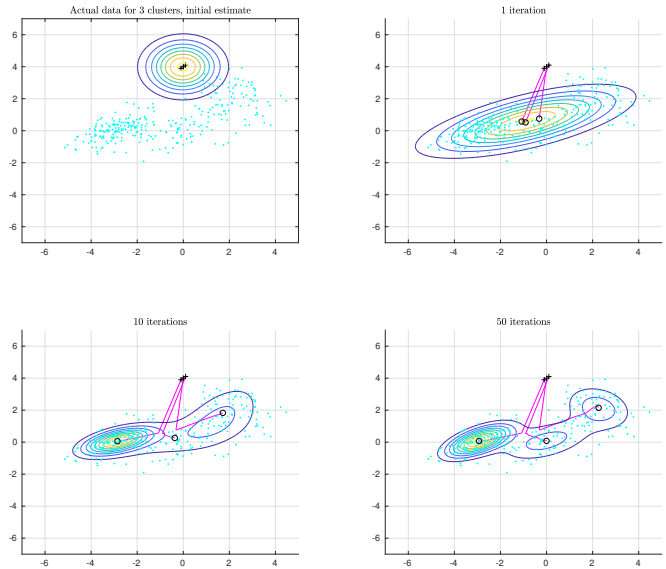
You will find that the relevant updates to obtain

$$\boldsymbol{\theta}_{t+1} = \{\boldsymbol{\pi}', \boldsymbol{\mu}'_1, \boldsymbol{\Sigma}'_1, \ldots, \boldsymbol{\mu}'_K, \boldsymbol{\Sigma}'_K\}.$$

are:

$$\begin{aligned}
\pi'_j &= \frac{\sum_{i=1}^{m} \gamma_i^{(j)}}{m} \\
\boldsymbol{\mu}'_j &= \frac{\sum_{i=1}^{m} \gamma_i^{(j)} \mathbf{x}_i}{\sum_{i=1}^{m} \gamma_i^{(j)}} \\
\boldsymbol{\Sigma}'_j &= \frac{\sum_{i=1}^{m} \gamma_i^{(j)} (\mathbf{x}_i - \boldsymbol{\mu}'_j)(\mathbf{x}_i - \boldsymbol{\mu}'_j)^T}{\sum_{i=1}^{m} \gamma_i^{(j)}}.
\end{aligned}$$

The EM algorithm for a mixture of Gaussians



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/

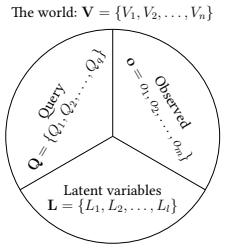
Part V

Bayesian networks

Markov random fields

Uncertainty: Probability as Degree of Belief

At the start of the course, I presented a *uniform approach* to *knowledge representation and reasoning* using *probability*.

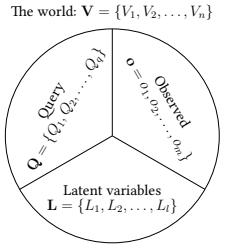


The world is represented by RVs $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$. These are partitioned:

1. Query variables $\mathbf{Q} = \{Q_1, Q_2, \ldots, Q_q\}$. We want to *compute a distribution over these.*
2. Observed variables $\mathbf{O} = \{o_1, o_2, \ldots, o_m\}$. We *know the values* of these.
3. Latent variables $\mathbf{L} = \{L_1, L_2, \ldots, L_l\}$. *Everything else.*

General knowledge representation and inference: the BIG PICTURE

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



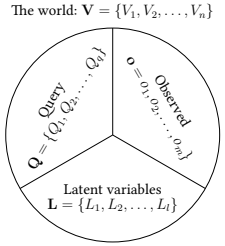
To compute a conditional distribution from a knowledge base $\Pr(\mathbf{V})$ we have to *sum over the latent variables*

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

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

General knowledge representation and inference: the BIG PICTURE

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



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

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

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 intractable.

Bayesian Networks

Having seen that in principle, if not in practice, the full joint distribution alone can be used to perform any inference of interest, we now examine a *practical technique*.

- We introduce the *Bayesian Network (BN)* as a compact representation of the full joint distribution.
- We examine the way in which a BN can be *constructed*.
- We examine the *semantics* of BNs.
- We look briefly at how *inference* can be performed.
- We briefly introduce the *Markov random field (MRF)* as an alternative means of representing a distribution.

Conditional probability—a brief aside…

A brief aside on the dangers of interpreting *implication* versus *conditional probability*:

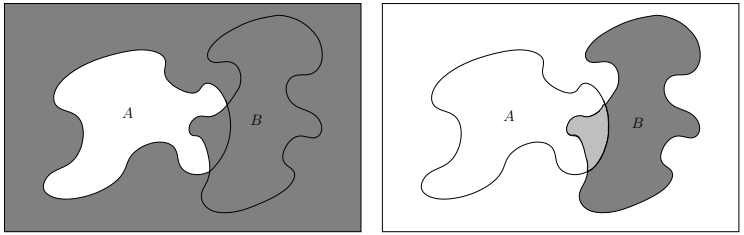
- $\Pr(X = x|Y = y) = 0.1$ does *not* mean that if $Y = y$ is then $\Pr(X = x) = 0.1$.
- $\Pr(X)$ is a *prior probability*. It applies when you *haven't seen* the value of Y.
- The notation $\Pr(X|Y = y)$ is for use when y is the *entire evidence*.
- $\Pr(X|Y = y \wedge Z = z)$ might be very different.

Conditional probability is *not* analogous to *logical implication*.

Implication and conditional probability

In general, it is difficult to relate *implication* to *conditional probability*.

$\Pr(A \to B) = \Pr(\neg A \vee B)$ $\qquad$ $\Pr(A|B) = \frac{\Pr(A \wedge B)}{\Pr(B)}$



Imagine that `fish` are very rare, and most fish can `swim`.

With implication,

$$\Pr(\texttt{fish} \to \neg\texttt{swim}) = \Pr(\neg\texttt{fish} \vee \neg\texttt{swim}) = \text{LARGE!}$$

With conditional probability,

$$\Pr(\neg\texttt{swim}|\texttt{fish}) = \frac{\Pr(\neg\texttt{swim} \wedge \texttt{fish})}{\Pr(\texttt{fish})} = \text{SMALL!}$$

Bayesian networks: exploiting independence

One of the key reasons for the introduction of *Bayesian networks* is to let us *exploit independence*.

The initial pay-off is that this *makes it easier to represent* $\Pr(\mathbf{V})$.

A further pay-off is that it *introduces structure* that can lead to *more efficient inference*.

Here is a *very simple* example.

If I toss a coin and roll a die, the full joint distribution of outcomes requires $2 \times 6 = 12$ numbers to be specified.

	⚀	⚁	⚂	⚃	⚄	⚅
H	0.014	0.028	0.042	0.057	0.071	0.086
T	0.033	0.067	0.1	0.133	0.167	0.2

Here $\Pr(\texttt{Coin} = H) = 0.3$ and the die has probability $i/21$ for the ith outcome.

Exploiting independence

BUT: if we assume the outcomes are independent then

$$\Pr(\texttt{Coin}, \texttt{Dice}) = \Pr(\texttt{Coin}) \Pr(\texttt{Dice})$$

Where $\Pr(\texttt{Coin})$ has two numbers and $\Pr(\texttt{Dice})$ has six.

So instead of 12 numbers we only need 8.

Exploiting independence

A slightly more complex example:

	CP		¬CP	
	SB	¬SB	SB	¬SB
HD	0.024	0.006	0.016	0.004
¬HD	0.0019	0.0076	0.1881	0.7524

- HD = Heart disease
- CP = Chest pain
- SB = Shortness of breath

Similarly, say instead of just considering HD, SB and CP we also consider the outcome of the *Oxford versus Cambridge tiddlywinks competition* TC:

$$\text{TC} = \{\text{Oxford}, \text{Cambridge}, \text{Draw}\}.$$

Exploiting independence

Now

$$\Pr(\text{HD}, \text{SB}, \text{CP}, \text{TC}) = \Pr(\text{TC}|\text{HD}, \text{SB}, \text{CP}) \Pr(\text{HD}, \text{SB}, \text{CP}).$$

Assuming that the patient is not an *extraordinarily keen fan of tiddlywinks*, their cardiac health has nothing to do with the outcome, so

$$\Pr(\text{TC}|\text{HD}, \text{SB}, \text{CP}) = \Pr(\text{TC})$$

and $2 \times 2 \times 2 \times 3 = 24$ numbers has been reduced to $3 + 8 = 11$.

Conditional independence

However although in this case we might not be able to exploit independence directly we *can* say that

$$\Pr(\text{CP}, \text{SB}|\text{HD}) = \Pr(\text{CP}|\text{HD}) \Pr(\text{SB}|\text{HD})$$

which simplifies matters.

Conditional independence: $A \perp B|C$

- A is *conditionally independent of* B *given* C, written $A \perp B|C$, if
$$\Pr(A, B|C) = \Pr(A|C) \Pr(B|C).$$
- If we know that C is the case then A and B are independent.
- Equivalently $\Pr(A|B, C) = \Pr(A|C)$. (Prove this!)

Although CP and SB are *not* independent, they do not directly influence one another *in a patient known to have heart disease.*

This is much nicer!

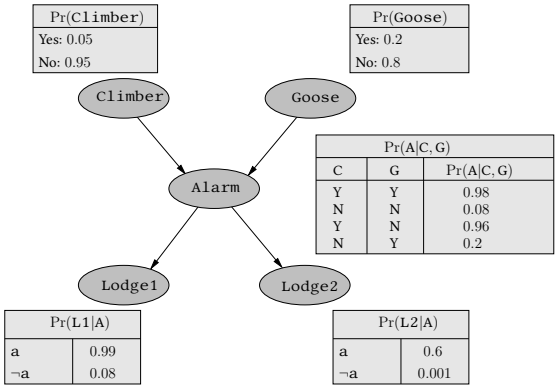
$$\Pr(\text{HD}|\text{CP}, \text{SB}) \propto \Pr(\text{CP}|\text{HD}) \Pr(\text{SB}|\text{HD}) \Pr(\text{HD})$$

Bayesian networks

After a *regrettable incident* involving an *inflatable gorilla*, a famous College has decided to install an alarm for the detection of roof climbers.

- The alarm is *very* good at detecting climbers.
- Unfortunately, it is also sometimes triggered when one of the *extremely fat geese* that lives in the College lands on the roof.
- One porter's lodge is near the alarm, and inhabited by a chap with *excellent hearing* and a *pathological hatred* of roof climbers: he *always* reports an alarm. His hearing is so good that he sometimes thinks he hears an alarm, *even when there isn't one*.
- Another porter's lodge is a good distance away and inhabited by an *old chap* with *dodgy hearing* who likes to listen to his collection of *DEATH METAL* with the sound turned up.

Bayesian networks



Bayesian networks

Also called *probabilistic/belief/causal networks* or *knowledge maps*.

- Each node is a *random variable (RV)*.
- Each node N_i has a distribution

$$\Pr\left(N_i|\texttt{parents}(N_i)\right)$$

- A Bayesian network is a *directed acyclic graph*.
- Roughly speaking, an arrow from N to M means N directly affects M.

Bayesian networks

Note that:

- In the present example all RVs are *discrete* (in fact Boolean) and so in all cases $\Pr\left(N_i|\text{parents}(N_i)\right)$ can be represented as a *table of numbers*.
- `Climber` and `Goose` have only *prior* probabilities.
- All RVs here are Boolean, so a node with p parents requires 2^p numbers.

A BN with n nodes represents the full joint probability distribution for those nodes as

$$\Pr\left(N_1 = n_1, N_2 = n_2, \ldots, N_n = n_n\right) = \prod_{i=1}^{n} \Pr\left(N_i = n_i|\text{parents}(N_i)\right).$$

For example

$$\begin{aligned}\Pr\left(\neg\mathsf{C}, \neg\mathsf{G}, \mathsf{A}, \mathsf{L1}, \mathsf{L2}\right) &= \Pr\left(\mathsf{L1}|\mathsf{A}\right)\Pr\left(\mathsf{L2}|\mathsf{A}\right)\Pr\left(\mathsf{A}|\neg\mathsf{C}, \neg\mathsf{G}\right)\Pr\left(\neg\mathsf{C}\right)\Pr\left(\neg\mathsf{G}\right)\\ &= 0.99 \times 0.6 \times 0.08 \times 0.95 \times 0.8.\end{aligned}$$

Semantics

In general $\Pr(A, B) = \Pr(A|B)\Pr(B)$ so

$$\Pr(N_1, \ldots, N_n) = \Pr(N_n|N_{n-1}, \ldots, N_1)\boxed{\Pr(N_{n-1}, \ldots, N_1)}.$$

Repeating this gives

$$\begin{aligned}\Pr(N_1, \ldots, N_n) &= \Pr(N_n|N_{n-1}, \ldots, N_1)\boxed{\Pr(N_{n-1}|N_{n-2}, \ldots, N_1)\cdots\Pr(N_1)} \\ &= \prod_{i=1}^{n}\Pr(N_i|N_{i-1}, \ldots, N_1).\end{aligned}$$

Now compare equations. We see that BNs make the assumption

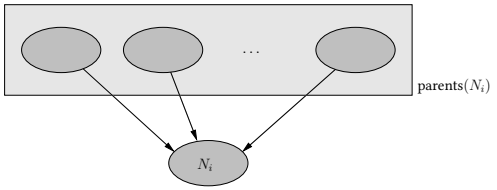
$$\Pr(N_i|N_{i-1}, \ldots, N_1) = \Pr(N_i|\text{parents}(N_i))$$

for each node, assuming that $\text{parents}(N_i) \subseteq \{N_{i-1}, \ldots, N_1\}$.

Each N_i is conditionally independent of its predecessors given its parents .

Semantics

- When constructing a BN we want to make sure the preceding property holds.
- This means we need to take care over *ordering*.
- In general *causes should directly precede effects.*

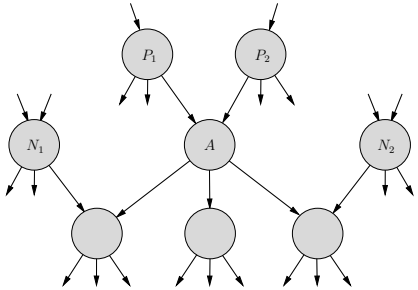


Here, $\text{parents}(N_i)$ contains all preceding nodes having a *direct influence* on N_i.

Semantics

But its not quite that straightforward: what if we want to talk about nodes *other than predecessors and parents*?

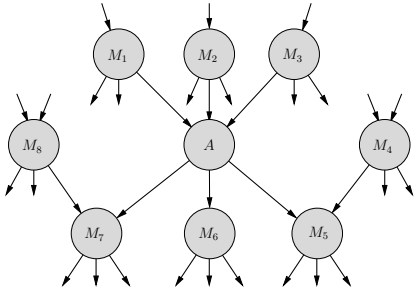
For example, it is possible to show:



Any node A is conditionally independent of the N_i—its *non-descendants*—given the P_i—its parents.

Semantics

It is also possible to show:



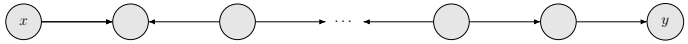
Any node A is conditionally independent of all other nodes given the *Markov blanket* M_i—that is, its *parents*, its *children* and its *childrens' parents.*

Semantics: what's REALLY going on here?

There is a *general method* for inferring exactly *what conditional independences are implied by a Bayesian network*.

Let X, Y and Z be disjoint subsets of the RVs.

Consider a *path* p consisting of directed (in any orientation) edges from some $x \in X$ to some $y \in Y$. For example

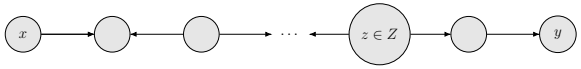


The path p is said to be *blocked* by Z if one of *three conditions* holds…

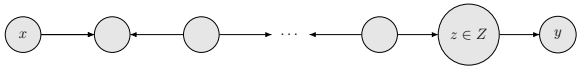
Semantics: what's REALLY going on here?

Path p is *blocked* with respect to Z if:

1. p contains a node $z \in Z$ that is *tail-to-tail*:

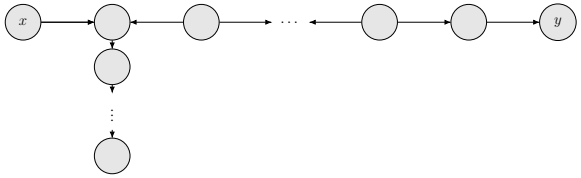


2. p contains a node $z \in Z$ that is *head-to-tail*:



   (Similarly if the node is *tail-to-head*.)

3. p contains a node N that is *head-to-head*, $N \notin Z$, and none of N's descendents is in Z:



Semantics: what's REALLY going on here?

Finally:

1. X and Y are *d-separated by* Z if *all paths* p from some $x \in X$ to some $y \in Y$ are blocked.
2. If X and Y are *d-separated by* Z then $X \perp Y|Z$.

More complex nodes

How do we represent

$$\Pr\left(N_i|\text{parents}(N_i)\right)$$

when nodes can denote *general discrete and/or continuous RVs*?

- BNs containing both kinds of RV are called *hybrid BNs*.
- Naive *discretisation* of continuous RVs tends to result in both a reduction in accuracy and large tables.
- $O(2^p)$ might still be large enough to be unwieldy.
- We can instead attempt to use *standard and well-understood* distributions, such as the *Gaussian*.
- This will typically require only a small number of parameters to be specified.

More complex nodes

Example: a continuous RV with one continuous and one discrete parent.

$$\Pr(\texttt{Speed of car}|\texttt{Throttle position}, \texttt{Tuned engine})$$

where SC and TP are continuous and TE is Boolean.

- For a specific setting of ET $=$ `true` it might be the case that SC increases with TP, but that some uncertainty is involved

$$\Pr(\text{SC}|\text{TP}, \texttt{et}) = N(g_{\texttt{et}}\text{TP} + c_{\texttt{et}}, \sigma^2_{\texttt{et}}).$$

- For an un-tuned engine we might have a similar relationship with a different behaviour

$$\Pr(\text{SC}|\text{TP}, \neg\texttt{et}) = N(g_{\neg\texttt{et}}\text{TP} + c_{\neg\texttt{et}}, \sigma^2_{\neg\texttt{et}}).$$

There is a set of parameters $\{g, c, \sigma\}$ for each possible value of the discrete RV.

More complex nodes

Example: a discrete RV with a continuous parent

$$\Pr(\texttt{Go roofclimbing}|\texttt{Size of fine}).$$

We could for example use the *probit distribution*

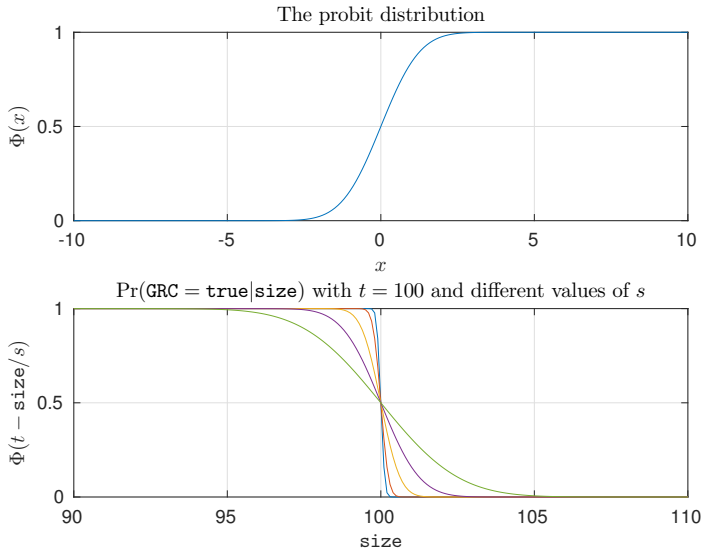
$$\Pr(\texttt{Go roofclimbing} = \texttt{true}|\texttt{size}) = \Phi\left(\frac{t - \texttt{size}}{s}\right)$$

where

$$\Phi(x) = \int_{-\infty}^{x} N(y)dy$$

and N is the Gaussian density with *zero mean and variance* 1.

More complex nodes



Basic inference

We saw earlier that the full joint distribution can be used to perform *all inference tasks*:

$$\Pr(\mathbf{Q}|o_1, o_2, \ldots, o_m) = \frac{1}{Z}\sum_{\mathbf{L}} \Pr(\mathbf{Q}, \mathbf{L}, o_1, o_2, \ldots, o_m)$$

where

- $\mathbf{Q}$ is the query.
- $o_1, o_2, \ldots, o_m$ are the observations.
- $\mathbf{L}$ are the latent variables.
- $1/Z$ normalises the distribution.
- The query, observations and latent variables are a partition of the set $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$ of all variables.

Basic inference

As the BN fully describes the full joint distribution

$$\Pr(\mathbf{Q}, \mathbf{L}, o_1, o_2, \ldots, o_m) = \prod_{i=1}^{n} \Pr(V_i|\text{parents}(V_i))$$

it can be used to perform inference in the *obvious* way

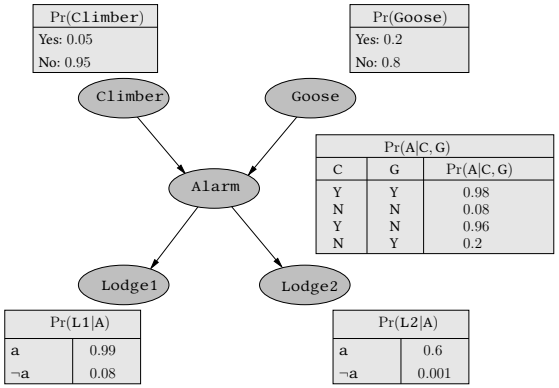
$$\Pr(\mathbf{Q}|o_1, o_2, \ldots, o_m) \propto \sum_{\mathbf{L}} \prod_{i=1}^{n} \Pr(V_i|\text{parents}(V_i))$$

but this is *in practice problematic* for obvious reasons.

- More sophisticated algorithms aim to achieve this *more efficiently*.
- For complex BNs we resort to *approximation techniques*.

Performing exact inference

$\Pr(\mathbf{Q}, \mathbf{L}, o_1, \ldots, o_m)$ has a particular form expressing conditional independences:



$$\Pr(C, G, A, L1, L2) = \Pr(C)\Pr(G)\Pr(A|C, G)\Pr(L1|A)\Pr(L2|A)\,.$$

Performing exact inference

Consider the computation of the query $\Pr(C|l1, l2)$

We have

$$\Pr(C|l1, l2) \propto \sum_{A}\sum_{G} \Pr(C)\Pr(G)\Pr(A|C, G)\Pr(l1|A)\Pr(l2|A)\,.$$

Here there are 5 multiplications for each set of values that appears for summation, and there are 4 such values.

In general this gives time complexity $O(n2^n)$ for n Boolean RVs.

The naive implementation of this approach yields the *Enumerate-Joint-Ask* algorithm, which unfortunately requires $O(2^n)$ time and space for n Boolean RVs.

The *enumeration-ask* algorithm improves matters to $O(2^n)$ time and $O(n)$ space by performing the computation *depth-first*.

However matters can be improved further by avoiding *duplication of computations*.

Performing exact inference

Looking more closely we see that

$$\begin{aligned}\Pr(C|l1, l2) &\propto \sum_{A}\sum_{G} \Pr(C)\Pr(G)\Pr(A|C, G)\Pr(l1|A)\Pr(l2|A)\\ &= \frac{1}{Z}\Pr(C)\sum_{A}\Pr(l1|A)\Pr(l2|A)\sum_{G}\Pr(G)\Pr(A|C, G)\\ &= \frac{1}{Z}\Pr(C)\sum_{G}\Pr(G)\sum_{A}\Pr(A|C, G)\Pr(l1|A)\Pr(l2|A)\,.\end{aligned}$$

There is some freedom in terms of how we *factorize* the expression.

This is a result of introducing *assumptions about conditional independence*.

Performing exact inference: variable elimination

Taking the second possibility:

$$\underbrace{\Pr(C)}_{C}\sum_{G}\underbrace{\Pr(G)}_{G}\sum_{A}\underbrace{\Pr(A|C,G)}_{A}\underbrace{\Pr(l1|A)}_{L1}\underbrace{\Pr(l2|A)}_{L2}$$

where C, G, A, $L1$, $L2$ denote the relevant *factors*.

The basic idea is to evaluate this from right to left (or in terms of the tree, bottom up) *storing results* as we progress and *re-using them* when necessary.

$\Pr(l1|A)$ depends on the value of A. We store it as a table $\mathbf{F}_{L1}(A)$. Similarly for $\Pr(l2|A)$.

$$\mathbf{F}_{L1}(A)=\begin{pmatrix}0.99\\0.08\end{pmatrix}\quad\mathbf{F}_{L2}(A)=\begin{pmatrix}0.6\\0.001\end{pmatrix}$$

as $\Pr(l1|a)=0.99$, $\Pr(l1|\neg a)=0.08$ and so on.

Performing exact inference: variable elimination

Similarly for $\Pr(A|C,G)$, which is dependent on A, C and G

$\mathbf{F}_A(A,C,G)=$

A	C	G	$\mathbf{F}_A(A,C,G)$
$\top$	$\top$	$\top$	0.98
$\top$	$\top$	$\bot$	0.96
$\top$	$\bot$	$\top$	0.2
$\top$	$\bot$	$\bot$	0.08
$\bot$	$\top$	$\top$	0.02
$\bot$	$\top$	$\bot$	0.04
$\bot$	$\bot$	$\top$	0.8
$\bot$	$\bot$	$\bot$	0.92

Can we write $\Pr(A|C,G)\Pr(l1|A)\Pr(l2|A)$ as

$\mathbf{F}_A(A,C,G)\mathbf{F}_{L1}(A)\mathbf{F}_{L2}(A)$

in a reasonable way?

Performing exact inference: variable elimination

Yes, provided *multiplication of factors* is defined correctly. Looking at

$$\Pr(C)\sum_{G}\Pr(G)\sum_{A}\Pr(A|C,G)\Pr(l1|A)\Pr(l2|A)$$

note that:

1. The values of the product
$$\Pr(A|C,G)\Pr(l1|A)\Pr(l2|A)$$
in the summation over A depend on the values of C and G external to it, and the values of A.
2. So
$$\mathbf{F}_A(A,C,G)\mathbf{F}_{L1}(A)\mathbf{F}_{L2}(A)$$
should be a table collecting values where correspondences between RVs are maintained.

This leads to a definition for *multiplication of factors* best given by example.

Performing exact inference: variable elimination

$$\mathbf{F}(A,B)\mathbf{F}(B,C)=\mathbf{F}(A,B,C)$$

where

A	B	$\mathbf{F}(A,B)$
$\top$	$\top$	0.3
$\top$	$\bot$	0.9
$\bot$	$\top$	0.4
$\bot$	$\bot$	0.1

B	C	$\mathbf{F}(B,C)$
$\top$	$\top$	0.1
$\top$	$\bot$	0.8
$\bot$	$\top$	0.8
$\bot$	$\bot$	0.3

A	B	C	$\mathbf{F}(A,B,C)$
$\top$	$\top$	$\top$	0.3×0.1
$\top$	$\top$	$\bot$	0.3×0.8
$\top$	$\bot$	$\top$	0.9×0.8
$\top$	$\bot$	$\bot$	0.9×0.3
$\bot$	$\top$	$\top$	0.4×0.1
$\bot$	$\top$	$\bot$	0.4×0.8
$\bot$	$\bot$	$\top$	0.1×0.8
$\bot$	$\bot$	$\bot$	0.1×0.3

This process gives us

$$\mathbf{F}_A(A,C,G)\mathbf{F}_{L1}(A)\mathbf{F}_{L2}(A) =$$

A	C	G	
$\top$	$\top$	$\top$	$0.98 \times 0.99 \times 0.6$
$\top$	$\top$	$\bot$	$0.96 \times 0.99 \times 0.6$
$\top$	$\bot$	$\top$	$0.2 \times 0.99 \times 0.6$
$\top$	$\bot$	$\bot$	$0.08 \times 0.99 \times 0.6$
$\bot$	$\top$	$\top$	$0.02 \times 0.08 \times 0.001$
$\bot$	$\top$	$\bot$	$0.04 \times 0.08 \times 0.001$
$\bot$	$\bot$	$\top$	$0.8 \times 0.08 \times 0.001$
$\bot$	$\bot$	$\bot$	$0.92 \times 0.08 \times 0.001$

How about

$$\mathbf{F}_{\overline{A},L1,L2}(C,G) = \sum_A \mathbf{F}_A(A,C,G)\mathbf{F}_{L1}(A)\mathbf{F}_{L2}(A)$$

To denote the fact that A has been summed out we place a bar over it in the notation.

$$\begin{aligned}\sum_A \mathbf{F}_A(A,C,G)\mathbf{F}_{L1}(A)\mathbf{F}_{L2}(A) =& \mathbf{F}_A(a,C,G)\mathbf{F}_{L1}(a)\mathbf{F}_{L2}(a) \\ &+ \mathbf{F}_A(\neg a,C,G)\mathbf{F}_{L1}(\neg a)\mathbf{F}_{L2}(\neg a)\end{aligned}$$

where

$$\mathbf{F}_A(a,C,G) =$$

C	G	
$\top$	$\top$	0.98
$\top$	$\bot$	0.96
$\bot$	$\top$	0.2
$\bot$	$\bot$	0.08

$\mathbf{F}_{L1}(a) = 0.99$ $\mathbf{F}_{L2}(a) = 0.6$

and similarly for $\mathbf{F}_A(\neg a,C,G)$, $\mathbf{F}_{L1}(\neg a)$ and $\mathbf{F}_{L2}(\neg a)$.

$$\mathbf{F}_A(a,C,G)\mathbf{F}_{L1}(a)\mathbf{F}_{L2}(a) =$$

C	G	
$\top$	$\top$	$0.98 \times 0.99 \times 0.6$
$\top$	$\bot$	$0.96 \times 0.99 \times 0.6$
$\bot$	$\top$	$0.2 \times 0.99 \times 0.6$
$\bot$	$\bot$	$0.08 \times 0.99 \times 0.6$

$$\mathbf{F}_A(\neg a,C,G)\mathbf{F}_{L1}(\neg a)\mathbf{F}_{L2}(\neg a) =$$

C	G	
$\top$	$\top$	$0.02 \times 0.08 \times 0.001$
$\top$	$\bot$	$0.04 \times 0.08 \times 0.001$
$\bot$	$\top$	$0.8 \times 0.08 \times 0.001$
$\bot$	$\bot$	$0.92 \times 0.08 \times 0.001$

$$\mathbf{F}_{\overline{A},L1,L2}(C,G) =$$

C	G	
$\top$	$\top$	$(0.98 \times 0.99 \times 0.6) + (0.02 \times 0.08 \times 0.001)$
$\top$	$\bot$	$(0.96 \times 0.99 \times 0.6) + (0.04 \times 0.08 \times 0.001)$
$\bot$	$\top$	$(0.2 \times 0.99 \times 0.6) + (0.8 \times 0.08 \times 0.001)$
$\bot$	$\bot$	$(0.08 \times 0.99 \times 0.6) + (0.92 \times 0.08 \times 0.001)$

Now, say for example we have $\neg c, g$. Then doing the calculation explicitly would give

$$\begin{aligned}&\sum_A \mathrm{Pr}\,(A|\neg c, g)\mathrm{Pr}\,(l1|A))\,\mathrm{Pr}\,(l2|A) \\ &\qquad = \mathrm{Pr}\,(a|\neg c, g)\,\mathrm{Pr}\,(l1|a)\,\mathrm{Pr}\,(l2|a) + \mathrm{Pr}\,(\neg a|\neg c, g)\,\mathrm{Pr}\,(l1|\neg a)\,\mathrm{Pr}\,(l2|\neg a) \\ &\qquad = (0.2 \times 0.99 \times 0.6) + (0.8 \times 0.08 \times 0.001)\end{aligned}$$

which matches!

Continuing in this manner form

$$\mathbf{F}_{G,\overline{A},L1,L2}(C,G) = \mathbf{F}_G(G)\mathbf{F}_{\overline{A},L1,L2}(C,G)$$

sum out G to obtain $\mathbf{F}_{\overline{G},\overline{A},L1,L2}(C) = \sum_G \mathbf{F}_G(G)\mathbf{F}_{\overline{A},L1,L2}(C,G)$, form

$$\mathbf{F}_{C,\overline{G},\overline{A},L1,L2} = \mathbf{F}_C(C)\mathbf{F}_{\overline{G},\overline{A},L1,L2}(C)$$

and normalise.

Performing exact inference: variable elimination

What's the computational complexity now?

- For Bayesian networks with *suitable structure* we can perform inference in *linear time and space*.
- However in the worst case it is still *#P-hard*.

Consequently, we may need to resort to *approximate inference*.

Approximate inference for Bayesian networks

Markov chain Monte Carlo (MCMC) methods also provide a method for performing *approximate inference* in *Bayesian networks*.

Say a system can be in a state $\mathbf{S}$ and moves from state to state in discrete time steps according to a probabilistic transition

$$\Pr\left(\mathbf{S} \to \mathbf{S}'\right).$$

Let $\pi_t(\mathbf{S})$ be the probability distribution for the state after t steps, so

$$\pi_{t+1}(\mathbf{S}') = \sum_{\mathbf{s}} \Pr\left(\mathbf{s} \to \mathbf{S}'\right) \pi_t(\mathbf{s}).$$

If at some point we obtain $\pi_{t+1}(\mathbf{s}) = \pi_t(\mathbf{s})$ for all $\mathbf{s}$ then we have reached a *stationary distribution* π. In this case

$$\forall \mathbf{s}' \pi(\mathbf{s}') = \sum_{\mathbf{s}} \Pr\left(\mathbf{s} \to \mathbf{s}'\right) \pi(\mathbf{s}).$$

There is exactly one stationary distribution for a given $\Pr\left(\mathbf{S} \to \mathbf{S}'\right)$ provided the latter obeys some simple conditions.

Approximate inference for Bayesian networks

The condition of *detailed balance*

$$\forall \mathbf{s}, \mathbf{s}' \pi(\mathbf{s}) \Pr\left(\mathbf{s} \to \mathbf{s}'\right) = \pi(\mathbf{s}') \Pr\left(\mathbf{s}' \to \mathbf{s}\right)$$

is sufficient to provide a π that is a stationary distribution. To see this simply sum:

$$\begin{aligned}
\sum_{\mathbf{s}} \pi(\mathbf{s}) \Pr\left(\mathbf{s} \to \mathbf{s}'\right) &= \sum_{\mathbf{s}} \pi(\mathbf{s}') \Pr\left(\mathbf{s}' \to \mathbf{s}\right) \\
&= \pi(\mathbf{s}') \underbrace{\sum_{\mathbf{s}} \Pr\left(\mathbf{s}' \to \mathbf{s}\right)}_{=1} \\
&= \pi(\mathbf{s}')
\end{aligned}$$

If all this is looking a little familiar, it's because we now have another excellent application for the material in *Mathematical Methods for Computer Science*.

That course used the alternative term *local balance*.

Approximate inference for Bayesian networks

Recalling once again the basic equation for performing probabilistic inference

$$\Pr\left(\mathbf{Q} | o_1, o_2, \ldots, o_m\right) \propto \sum_{\mathbf{L}} \Pr\left(\mathbf{Q}, \mathbf{L}, o_1, o_2, \ldots, o_m\right)$$

where

- $\mathbf{Q}$ is the query.
- $o_1, o_2, \ldots, o_m$ are the observations.
- $\mathbf{L}$ are the latent variables.
- $1/Z$ normalises the distribution.
- The query, observations and latent variables are a partition of the set $\mathbf{V} = \{V_1, V_2, \ldots, V_n\}$ of all variables.

We are going to consider obtaining samples from the distribution $\Pr\left(\mathbf{Q}, \mathbf{L} | o_1, o_2, \ldots, o_m\right)$.

The observations are fixed. Let the *state* of our system be a specific set of values for *a query variable and the latent variables*

$$\mathbf{S} = (S_1, S_2, \ldots, S_{l+1}) = (Q, L_1, L_2, \ldots, L_l)$$

and define $\overline{\mathbf{S}}_i$ to be the state vector *with* S_i *removed*

$$\overline{\mathbf{S}}_i = (S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_{n+1}).$$

To move from $\mathbf{s}$ to $\mathbf{s}'$ we replace one of its elements, say s_i, with a new value s_i' sampled according to

$$s_i' \sim \Pr\left(S_i | \overline{\mathbf{s}}_i, o_1, \ldots, o_m\right)$$

This has detailed balance, and has $\Pr\left(Q, \mathbf{L} | o_1, \ldots, o_m\right)$ as its stationary distribution.

It is known as *Gibbs sampling*.

To see that $\Pr\left(Q, \mathbf{L} | \mathbf{o}\right)$ is the stationary distribution we just demonstrate *detailed balance*:

$$\begin{aligned}
\pi(\mathbf{s})\Pr\left(\mathbf{s} \to \mathbf{s}'\right) &= \Pr\left(\mathbf{s}|\mathbf{o}\right)\Pr\left(s_i'|\overline{\mathbf{s}}_i, \mathbf{o}\right) \\
&= \Pr\left(s_i, \overline{\mathbf{s}}_i|\mathbf{o}\right)\Pr\left(s_i'|\overline{\mathbf{s}}_i, \mathbf{o}\right) \\
&= \Pr\left(s_i|\overline{\mathbf{s}}_i, \mathbf{o}\right)\Pr\left(\overline{\mathbf{s}}_i|\mathbf{o}\right)\Pr\left(s_i'|\overline{\mathbf{s}}_i, \mathbf{o}\right) \\
&= \Pr\left(s_i|\overline{\mathbf{s}}_i, \mathbf{o}\right)\Pr\left(s_i', \overline{\mathbf{s}}_i|\mathbf{o}\right) \\
&= \Pr\left(\mathbf{s}' \to \mathbf{s}\right)\pi(\mathbf{s}').
\end{aligned}$$

As a further simplification we can exploit *conditional independence*.

For example, sampling from $\Pr\left(S_i|\overline{\mathbf{s}}_i, \mathbf{o}\right)$ may be equivalent to sampling S_i conditional on some smaller set.

So:

- We successively sample the query variable and the unobserved variables, conditional on the remaining variables.
- This gives us a sequence $\mathbf{s}_1, \mathbf{s}_2, \ldots$ sampled according to $\Pr\left(Q, \mathbf{L}|\mathbf{o}\right)$.

Finally, note that as

$$\Pr\left(Q|\mathbf{o}\right) = \sum_{\mathbf{l}} \Pr\left(Q, \mathbf{l}|\mathbf{o}\right)$$

we can just ignore the values obtained for the unobserved variables. This gives us $q_1, q_2, \ldots$ with

$$q_i \sim \Pr\left(Q|\mathbf{o}\right).$$

To see that the final step works, consider what happens when we estimate the expected value of some function of Q.

$$\begin{aligned}
\mathbb{E}[f(Q)|\mathbf{o}] &= \sum_q f(q)\Pr\left(q|\mathbf{o}\right) \\
&= \sum_q f(q) \sum_{\mathbf{l}} \Pr\left(q, \mathbf{l}|\mathbf{o}\right) \\
&= \sum_q \sum_{\mathbf{l}} f(q)\Pr\left(q, \mathbf{l}|\mathbf{o}\right)
\end{aligned}$$

so sampling using $\Pr\left(q, \mathbf{l}|\mathbf{o}\right)$ and ignoring the values for $\mathbf{l}$ obtained works exactly as required.

Markov random fields

Markov random fields (MRFs) (sometimes called *undirected graphical models* or *Markov networks*) provide an *alternative approach* to representing a *probability distribution* while expressing *conditional independence assumptions*.

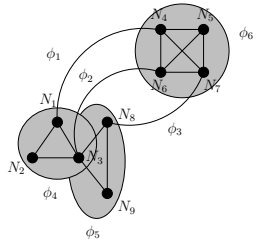
We now have:

1. An *undirected graph* $G = (N, E)$.
2. G has *a node* N_i *for each RV*.
3. For each *maximal clique* c in G there is a *clique potential* $\phi_c(N_c) > 0$ where N_c is the set of nodes in c.
4. The probability distribution expressed by G is

$$\Pr(N) \propto \prod_c \phi_c(N_c).$$

Markov random fields

Example: 3 maximal cliques of size 2, 2 of size 3 and 1 of size 4.



$$\Pr(N_1, \ldots, N_9) \propto \phi_1(N_1, N_4) \times \phi_2(N_3, N_6) \times \phi_3(N_7, N_8) \times \phi_4(N_1, N_2, N_3) \times \phi_5(N_3, N_8, N_9) \times \phi_6(N_4, N_5, N_6, N_7).$$

Markov random fields—conditional independence

The *test for conditional independence* is now simple: if X, Y and Z are disjoint subsets of the RVs then:

1. *Remove* the nodes in Z *and any attached edges* from the graph.
2. If there are *no paths from any variable in* X to *any variable in* Y then

$$X \perp Y | Z.$$

Final things to note:

1. MRFs have their *own algorithms for inference*.
2. They are an *alternative* to *BNs* for representing a probability distribution.
3. There are *trade-offs* that might make a BN or MRF *more or less favourable*.
4. For example: *potentials offer flexibility* because *they don't have to represent conditional distributions…*
5. …BUT you have to *normalize* the distribution you're representing.