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*.

\[
\text{The world: } V = \{V_1, V_2, \ldots, V_n\}
\]

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

\[
\Pr(V) = \Pr(V_1, V_2, \ldots, V_n)
\]
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*

$$\Pr(\mathbf{Q} | o_1, o_2, \ldots, o_m)$$
The latent variables $\mathbf{L}$ are all the RVs not in the sets $\mathbf{Q}$ or $\mathbf{O}$.

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

Query $\mathbf{Q} = \{Q_1, Q_2, \ldots, Q_k\}$

Observed $\mathbf{o} = \{o_1, o_2, \ldots, o_m\}$

Latent variables $\mathbf{L} = \{L_1, L_2, \ldots\}$

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

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

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

Knowledge base
General knowledge representation and inference: the BIG PICTURE

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

The world: \( V = \{V_1, V_2, \ldots, V_n\} \)

Query: \( Q = \{Q_1, Q_2, \ldots, Q_k\} \)

Observed: \( o = \{o_1, o_2, \ldots, o_m\} \)

Latent variables: \( L = \{L_1, L_2, \ldots\} \)

For example, if we now receive a new observation \( O' = o' \) then

\[
\text{Pr} (Q|o', o_1, o_2, \ldots, o_m) = \frac{1}{Z} \text{Pr} (o'|Q, o_1, o_2, \ldots, o_m) \text{Pr} (Q|o_1, o_2, \ldots, o_m)
\]

\( Z \) is a normalization constant.
Simple eh?

HAH!!! No chance...

Even if all your RVs are just Boolean:

- For \( n \) RVs knowing the knowledge base \( \Pr(V) \) 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.
How can we get around this?

1. You can be clever about representing $\Pr(V)$ to avoid storing all $O(2^n)$ numbers.

2. You can take that a step further and *exploit the structure of* $\Pr(V)$ 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*. (Although this is covered in other courses.)
- 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.

Again, this will need to be covered in another course, as we don’t have time...
Books

Books recommended for the course:

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


and supplement it with one of the following:


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 \( x \).

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

\[
    h_{\mathbf{w}}(\mathbf{x}) = \Pr (Y = c_i | \mathbf{x}).
\]
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 $s = [(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)]$.

Fit a polynomial

$$h_w(x) = w_0 + w_1x + w_2x^2 + \cdots + w_dx^d$$

by choosing the weights $w_i$ to minimize

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

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:

\[ P_n = \sum_{i=0}^{n} w_i x_i \]

\[ z = \sigma(a) = h_w(x) \]

The trick is the same: select the weights \( w_i \) to minimize some measure of error \( E(w) \) 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

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

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

$$w_{t+1} = w_t - \lambda \frac{\partial E(w)}{\partial w} \bigg|_{w_t}$$
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:

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

where $\mathbf{x}_i^{(j)}$ is the $j$th 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 $h_w(x)$.
- The trick remains the same: minimize an error $E(w)$.
- We do that by gradient descent
  \[ w_{t+1} = w_t - \lambda \left. \frac{\partial E(w)}{\partial w} \right|_{w_t} \]
- This can be achieved using backpropagation.
- Backpropagation is just a method for computing $\partial E(w)/\partial w$. 
Backpropagation

I want to emphasize the last three statements:

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

It’s needed because we’re doing gradient descent

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

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 $x_1, x_2, \ldots, x_m$ 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 $x'_1, \ldots, x'_{m'}$. 

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

$$\mathbf{x}^T = [x_1 \ x_2 \ \cdots \ x_n]$$

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 $\mathbf{X}_{i*}$ and $\mathbf{X}_{*j}$ the $i$th row and $j$th column of $\mathbf{X}$ respectively.
Matrix notation

If we have $m$ vectors $x_1, x_2, \ldots, x_m$ then the $j$th element of the $i$th vector is $x_i^{(j)}$. We may also form the matrix

$$ X = \begin{bmatrix} x_1^T \\
                    x_2^T \\
             \vdots \\
                   x_m^T \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(n)} \\
                                      x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(n)} \\
                                             \vdots & \vdots & \ddots & \vdots \\
                                      x_m^{(1)} & x_m^{(2)} & \cdots & x_m^{(n)} \end{bmatrix} $$

Similarly we can write

$$ X^T = [x_1 \ x_2 \ \cdots \ x_m] $$

The identity matrix is as usual

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

The inverse of $X$ is $X^{-1}$ and its determinant is $|X|$. 
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 $X$ taking values in $\mathbb{R}^n$ and refer to a specific value $x \in \mathbb{R}^n$.

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

- **Inputs** are in $n$ dimensions and are denoted by
  \[ x^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \]

  Each element $x_i$ is a feature.

- A training sequence has $m$ elements. The $m$ inputs are $x_1, \ldots, x_m$ and can be collected into the matrix
  \[ X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_m^T \end{bmatrix} \]

- The **labels** in the training sequence are denoted by
  \[ y^T = \begin{bmatrix} y_1 & y_2 & \cdots & y_m \end{bmatrix} \]

  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

$$s^T = [(x_1, y_1) \ (x_2, y_2) \ \ldots \ (x_m, y_m)].$$

This is the *training sequence*. 
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 $p(X, Y)$.

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

$$s^T = [(x_1, y_1) \ (x_2, y_2) \ \cdots \ (x_m, y_m)]$$

represents $m$ iid samples from the relevant distribution.

As the examples are iid we can write

$$p(s) = \prod_{i=1}^{m} p(x_i, y_i).$$
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* $h_w$ that we *want to identify* by choosing $w$.

I chose $h_w$ to be a polynomial with parameters $w$ — this is the dashed blue line.

So in fact the unknown function is $h_w(x)$, emphasizing that $w$ *determines a specific function* $f$.

Remember: you don’t know what $w$ is: *you need to identify it by analysing* $s$. 
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 $\mu$ and variance $\sigma^2$. 
Example: simple regression

To make s:

For the $i$th example:

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

Combining steps 2 and 3 gives you $p(y_i|x_i, w)$.

$$
p(y_i|x_i, w) = \mathcal{N}(h_w(x_i), \sigma^2)
= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_i - h_w(x_i))^2\right).
$$
The likelihood function

The quantity \( p(y_i|x_i, w) \) is important: it is known as the *likelihood*. You will sometimes see it re-arranged and written as the *likelihood function*

\[
L(w|x_i, y_i) = p(y_i|x_i, w).
\]

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 imagine \( w \) is *fixed* (but hidden!) from the outset and extend the likelihood to the whole data set \( s \)…
The likelihood function

The *likelihood* for the full data set is:

\[
p(s|w) = \prod_{i=1}^{m} p(x_i, y_i|w) \\
= \prod_{i=1}^{m} p(y_i|x_i, w)p(x_i|w) \\
= \prod_{i=1}^{m} p(y_i|x_i, w)p(x_i)
\]

The last step involves the reasonable assumption that \(x_i\) itself never depends on \(w\).
Maximizing likelihood

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

This immediately suggests a way of choosing $w$: 

Choose 

$$w_{\text{opt}} = \arg\max_w p(s|w).$$

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

How would we solve this maximization problem?
Maximizing likelihood

This is surprisingly easy:

$$w_{opt} = \arg\max_w p(s|w)$$

$$= \arg\max_w \left( \prod_{i=1}^m p(y_i|x_i, w)p(x_i) \right)$$

$$= \arg\max_w \left( \sum_{i=1}^m \log p(y_i|x_i, w) + \sum_{i=1}^m \log p(x_i) \right)$$

$$= \arg\max_w \sum_{i=1}^m \log p(y_i|x_i, w)$$

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:

$$w_{\text{opt}} = \arg\max_w \left[ \sum_{i=1}^m \log \sqrt{2\pi\sigma^2} - \frac{1}{2\sigma^2} \sum_{i=1}^m (y_i - h_w(x_i))^2 \right]$$

$$= \arg\min_w \frac{1}{2\sigma^2} \sum_{i=1}^m (y_i - h_w(x_i))^2$$

So we’ve just shown that:

To choose $w$ 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

\[ E(w) = \sum_{i=1}^{m} (y_i - h_w(x_i))^2 \]

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 \textit{fixed in advance} but instead make it an RV as well?

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

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

with mean vector \( \boldsymbol{\mu} \) and covariance matrix \( \Sigma \).
Maximizing the posterior

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

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

Then

$$w_{\text{opt}} = \arg \max_w p(w|s)$$

$$= \arg \max_w \frac{p(s|w)p(w)}{p(s)}$$

$$= \arg \max_w \left[ \log p(s|w) + \log p(w) \right]$$

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

$$w_{\text{opt}} = \arg \min_w \left[ \frac{1}{2\sigma^2} \sum_{i=1}^{m} ((y_i - h_w(x_i))^2) + \frac{\lambda}{2} ||w||^2 \right].$$
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* $h_w$.
- The idea was to use $\lambda$ 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* $w$ 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 $p(y|x, w)$ 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 \( \Pr(Y|x, w) \). It has two non-negative values, and

\[
\Pr(Y = 1|x, w) = 1 - \Pr(Y = 0|x, w).
\]

So you can’t just add noise to the underlying \( h_w \).

**Fix:** define the likelihood as

\[
\Pr(Y = 1|x, w) = \sigma_\theta(h_w(x))
\]

and use something like

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

to impose the above property.
The likelihood for classification problems

Sigmoid function for values of $\theta$ from 1 to 5

Target and data for probabilistic classification
The likelihood for classification problems

Logistic $\sigma_\theta(z)$ applied to the output of a linear function

$P_r(x \text{ is in } C_1)$
The likelihood for classification problems

So: if we’re given a training sequence $s$, what is the probability that it was generated using some $w$?

For an example $(x, y)$

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

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

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

If we assume that the examples are iid then the probability of seeing the labels in a training sequence $s$ is straightforward.
The likelihood for classification problems

The likelihood is now

\[ p(s|w) = \prod_{i=1}^{m} p(y_i|x_i, w)p(x_i) \]

\[ = \prod_{i=1}^{m} [\sigma_\theta(h_w(x_i))]^{y_i} [1 - \sigma_\theta(h_w(x_i))]^{(1-y_i)} p(x_i) \]

where the first line comes straight from slide 43.

Note that:

- Whereas previously we had the noise variance \( \sigma^2 \) we now have the parameter \( \theta \). Both serve a similar purpose.

- From this expression we can directly derive maximum-likelihood and MAP learning algorithms for classifiers.
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...*
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 $x \in \mathbb{R}^d$.
- 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 $x$ to class $c_1’* and so on. We may have further actions, for example the action *‘I don’t know how to classify $x’*.

There is also a *loss* $\lambda_{ij}$ 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 $\lambda_{ij}$.
Bayesian decision theory

The ability to specify 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 healthy 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 $p(X|C)$ for seeing $X$ 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 $p(X|C)$.

\[
p(X, C) = p(X|C)\Pr(C) = \Pr(C|X)p(X)
\]

As usual Bayes rule tells us that

\[
\Pr(C|X) = \frac{1}{Z}p(X|C)\Pr(C)
\]

where

\[
Z = p(X) = \sum_{i=1}^{K} p(X|c_i)\Pr(c_i).
\]
Bayesian decision theory

Say *nature shows us* $x$ and we *take action* $a_i$.

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

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

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

Note that this particular $x$ is *fixed*. 
Bayesian decision theory

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

The average loss, or risk, is

$$R = \mathbb{E}_{(x,c) \sim p(x,c)} [\lambda(D(x), c)]$$
$$= \mathbb{E}_x \mathbb{E}_{c \sim \text{Pr}(C|x)} [\lambda(D(x), c)|x]$$
$$= \mathbb{E}_x [R(D(x)|x)]$$
$$= \int R(D(x)|x)p(x)dx.$$

Here we have used the standard result from probability theory that

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

(See the supplementary notes for a proof.)
Bayesian decision theory

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

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

This $D$ provides us with the *minimum possible risk*, or Bayes risk $R^*$. The rule specified is called the *Bayes decision rule*.
Example: minimum error rate classification

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

What loss should we consider in these circumstances?

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

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

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

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

(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

\[
R = \mathbb{E}_{(x,c) \sim p(x,c)} [\lambda(D(x), C)]
= \Pr (D(x) \text{ chooses the wrong class})
\]

so the Bayes decision rule minimises the probability of error.
Example: minimum error rate classification

What is the Bayes decision rule in this case?

\[
R(a_i|\mathbf{x}) = \sum_{j=1}^{K} \lambda_{ij} \Pr(c_j|\mathbf{x})
\]

\[
= \sum_{i \neq j} \Pr(c_j|\mathbf{x})
\]

\[
= 1 - \Pr(c_i|\mathbf{x})
\]

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

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

But what about the *training sequence* $s$?

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 classifier.)
- 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* $x$ *and* $s$ then you find that…
- …to minimize error probability you should maximize $\Pr(C|x, s)$. 
Bayesian supervised learning

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

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

$$
\begin{align*}
\Pr (C | x, s) &= \sum_h \Pr (C, h | x, s) \\
&= \sum_h \Pr (C | h, x, s) \Pr (h | x, s) \\
&= \sum_h \Pr (C | h, x) \Pr (h | s) .
\end{align*}
$$

Here we have re-introduced $h$ using marginalisation.
Bayesian supervised learning

So our classification should be

\[
C = \arg\max_{C \in \{c_1, \ldots, c_K\}} \sum_{h} \Pr(C|h, x) \Pr(h|s)
\]

Of course, when dealing with hypotheses defined by weights \(w\) the sum becomes an integral

\[
C' = \arg\max_{C \in \{c_1, \ldots, c_K\}} \int_{\mathbb{R}^W} \Pr(C|w, x) p(w|s) \, dw
\]

where \(W\) is the number of weights. The key point:

- You can also write these equations in the form
  \[
  C' = \arg\max_{C \in \{c_1, \ldots, c_K\}} \mathbb{E}_{h \sim \Pr(h|s)} [\Pr(C|h, x)]
  \]

- 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

\[
C = \arg\max_{C \in \{c_1, \ldots, c_K\}} \mathbb{E}_{h \sim \Pr(h|s)} \left[ \Pr(C|h, x) \right]
\]

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

\[
C = \arg\max_{C \in \{c_1, \ldots, c_K\}} \mathbb{E}_{h \sim \Pr(h | s)} [\Pr (C | h, x)]
\]

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:

![Linearly separable data](image1)

![Simple, but not linearly separable, data](image2)

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 $x_1 x_2$. 
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:

[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

\[ h_w(x) = \sigma \left( w_0 + \sum_{i=1}^{n} w_i x_i \right) \]

Or \( h_w(x) = \sigma(w^T x) \) if we add an extra element having constant value 1 to \( x \).

Make it nonlinear by introducing basis functions \( \phi_i \):

\[ \Phi^T(x) = [\phi_1(x) \ \phi_2(x) \ \cdots \ \phi_k(x)] \]

\[ h_w(x) = \sigma \left( w_0 + \sum_{i=1}^{k} w_i \phi_i(x) \right) \]

or assuming there’s a basis function \( \phi(x) = 1 \)

\[ h_w(x) = \sigma(w^T \Phi(x)). \]
Linear regression

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

$$s^T = [(x_1, y_1) \ (x_2, y_2) \ \cdots \ (x_m, y_m)].$$

I want to minimize

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

Last year we would have found the gradient of $E(w)$ and used gradient descent

$$w_{t+1} = w_t - \lambda \frac{\partial E(w)}{\partial w} \bigg|_{w_t}.$$

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

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

\[
\frac{\partial \mathbf{a}^T \mathbf{x}}{\partial \mathbf{x}} = \left[ \frac{\partial \mathbf{a}^T \mathbf{x}}{\partial x_1} \frac{\partial \mathbf{a}^T \mathbf{x}}{\partial x_2} \ldots \frac{\partial \mathbf{a}^T \mathbf{x}}{\partial x_n} \right]^T = \mathbf{a}
\]

because for each element \( x_j \)

\[
\frac{\partial \mathbf{a}^T \mathbf{x}}{\partial x_j} = \frac{\partial}{\partial x_j} \left( a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \right) = a_j
\]

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

\[ \Phi = \begin{bmatrix} \Phi^T(x_1) \\ \Phi^T(x_2) \\ \vdots \\ \Phi^T(x_m) \end{bmatrix} \]

so

\[
E(w) = \frac{1}{2}(y - \Phi w)^T(y - \Phi w)
\]

\[
= \frac{1}{2} (y^T y - 2 y^T \Phi w + w^T \Phi^T \Phi w)
\]

and

\[
\frac{\partial E(w)}{\partial w} = \Phi^T \Phi w - \Phi^T y
\]
So the optimum solution is obtained by solving
\[ \Phi^T \Phi w = \Phi^T y \]
giving
\[ w_{opt} = (\Phi^T \Phi)^{-1} \Phi^T y \]

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

$$E(w) = \frac{1}{2} \sum_{i=1}^{m} (y_i - h_w(x_i))^2 + \frac{\lambda}{2} \|w\|^2.$$  

It is an exercise to show that the solution is:

$$w_{opt} = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T y$$

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 $\lambda$:

![Graphs showing target and data for different values of $\lambda$.]

How can we choose $\lambda$? 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 $\sigma$, typically the *sigmoid function*, so

$$h_w(x) = \sigma_\theta(w^T \Phi(x)).$$

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

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

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

$$E(w) = -\sum_{i=1}^{m} y_i \log \sigma_\theta(w^T \Phi(x_i)) + (1 - y_i) \log(1 - \sigma_\theta(w^T \Phi(x_i))).$$
Iterative re-weighted least squares

Introducing the extra nonlinearity means we can no longer minimize

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

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 \( f : \mathbb{R} \rightarrow \mathbb{R} \), 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 $E : \mathbb{R}^n \rightarrow \mathbb{R}$ iterate as follows:

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

Here the Hessian is the matrix of second derivatives of $E(w)$

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

All we need to do now is to work out the derivatives...
Iterative re-weighted least squares

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

Simplifying slightly we use \( \theta = 1 \) and define \( z_i = \sigma(w^T \Phi(x_i)) \). So

\[
\frac{\partial E(w)}{\partial w_k} = - \left[ \sum_{i=1}^{m} y_i \frac{1}{z_i} \frac{\partial z_i}{\partial w_k} + (1 - y_i) \frac{-1}{1 - z_i} \frac{\partial z_i}{\partial w_k} \right]
\]

\[
= \sum_{i=1}^{m} \frac{\partial z_i}{\partial w_k} \left( \frac{1 - y_i}{1 - z_i} - \frac{y_i}{z_i} \right)
\]

\[
= \sum_{i=1}^{m} \frac{\partial z_i}{\partial w_k} \frac{z_i - y_i}{z_i(1 - z_i)}.
\]
Iterative re-weighted least squares

So

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

Thus using the fact that

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

we have

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

and therefore

$$\frac{\partial E(w)}{\partial w} = \Phi^T(z - y).$$
Iterative re-weighted least squares

It is an exercise to show that

\[ H_{ij}(w) = \sum_{k=1}^{m} z_k (1 - z_k) \phi_i(x_k) \phi_j(x_k) \]

and therefore

\[ H(w) = \Phi^T Z \Phi \]

where \( Z \) is a diagonal matrix with diagonal elements \( z_k (1 - z_k) \).

This gives us the \textit{iterative re-weighted least squares algorithm} (IRLS)

\[ w_{t+1} = w_t - [\Phi^T Z \Phi]^{-1} \Phi^T (z - y). \]
Iterative re-weighted least squares
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 $\gamma$ is the *margin*, and this is the *maximum margin classifier.*
The maximum margin classifier

If you completed the exercises for Artificial Intelligence I then you’ll know that linear classifiers have a very simple geometry. For

\[ f(x) = w^T x + b \]

For \( x' \) on one side of the line \( f(x) = 0 \) we have \( f(x') > 0 \) and on the other side \( f(x') < 0 \).
The maximum margin classifier

Problems:

• Given the usual training data $s$, 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:

Constrained optimization

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

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

Ultimately we will need to be able to solve problems of the form: find $x_{opt}$ such that

$$x_{opt} = \arg\min_x f(x)$$

under the constraints

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

and

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

For example:

Minimize the function

\[ f(x, y) = - (2x + y^2 + xy) \]

subject to the constraint

\[ g(x, y) = x + 2y - 1 = 0. \]
Constrained optimization

*Step 1:* introduce the *Lagrange multiplier* $\lambda$ 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

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

So for our example we need

$$2 + y + \lambda = 0$$
$$2y + x + 2\lambda = 0$$
$$x + 2y - 1 = 0$$

where the last is just the constraint.
Constrained optimization

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.
Constrained optimization

How about the full problem? Find

$$x_{\text{opt}} = \arg\min_x f(x) \text{ such that } g_i(x) = 0 \text{ for } i = 1, 2, \ldots, n$$

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

The Lagrangian is now

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

and the relevant necessary conditions are more numerous.
Constrained optimization

The necessary conditions now require that when \( x' \) is a solution \( \exists \lambda', \alpha' \) such that

1. \[
\frac{\partial L(x', \lambda', \alpha')}{\partial x} = 0.
\]

2. The equality and inequality constraints are satisfied at \( x' \).

3. \( \alpha' \geq 0 \).

4. \( \alpha'_j h_j(x') = 0 \) 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.
Constrained optimization

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
   \[
   \tilde{L}(\alpha) = \inf_x L(x, \alpha).
   \]

2. The *dual optimization problem* is
   \[
   \max_{\alpha} \tilde{L}(\alpha) \text{ such that } \alpha \geq 0.
   \]

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*.

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:

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

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 \( w \).
3. We now classify using a “hard” threshold \( \text{sgn} \), rather than the “soft” threshold \( \sigma \).
The maximum margin classifier

Consider the geometry again. **Step 1:**

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

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

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

\[
\frac{|f_{w,w_0}(x')|}{||w||}.
\]
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 \((x_i, y_i)\) the quantity \(y_i f_{w,w_0}(x_i)\) will be positive if the resulting classification is correct...

- ...the aim is to solve:

\[
(w, w_0) = \arg \max_{w,w_0} \left[ \min_i \frac{y_i f_{w,w_0}(x_i)}{||w||} \right].
\]
The maximum margin classifier

YUK!!!(With bells on…)

The maximum margin classifier

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

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

$$\iff cw_0 + cw^T \Phi(x) = 0.$$ 

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

Version 1:

$$(w, w_0, \gamma) = \text{argmax} \gamma_{w,w_0,\gamma}$$

subject to the constraints

$$y_i f_{w,w_0}(x_i) \geq \gamma, i = 1, 2, \ldots, m$$

$$||w||^2 = 1.$$
The maximum margin classifier

Solution, version 2: still, convert to a constrained optimization, but instead of fixing $||w||$: 

Fix $\min\{y_i f_{w,w_0}(x_i)\}$ to be anything you like!

Version 2: 

$$(w, w_0) = \arg\min_{w,w_0} \frac{1}{2}||w||^2$$

subject to the constraints 

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

(This works because maximizing $\gamma$ now corresponds to minimizing $||w||$.)
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 $\frac{1}{2}||w||^2$

such that

$y_if_{w_0}(x_i) \geq 1$ for $i = 1, \ldots, m$.

Referring back, this means the Lagrangian is

$L(w, w_0, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i (y_if_{w_0}(x_i) - 1)$

and a necessary condition for a solution is that

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

107
The maximum margin classifier

Working these out is easy:

\[
\frac{\partial L(w, w_0, \alpha)}{\partial w} = \frac{\partial}{\partial w} \left( \frac{1}{2} ||w||^2 - \sum_{i=1}^{m} \alpha_i (y_i f_{w, w_0}(x_i) - 1) \right)
\]

\[
= w - \sum_{i=1}^{m} \alpha_i y_i \frac{\partial}{\partial w} (w^T \Phi(x_i) + w_0)
\]

\[
= w - \sum_{i=1}^{m} \alpha_i y_i \Phi(x_i)
\]

and

\[
\frac{\partial L(w, w_0, \alpha)}{\partial w_0} = -\frac{\partial}{\partial w_0} \left( \sum_{i=1}^{m} \alpha_i y_i f_{w, w_0}(x_i) \right)
\]

\[
= -\frac{\partial}{\partial w_0} \left( \sum_{i=1}^{m} \alpha_i y_i (w^T \Phi(x_i) + w_0) \right)
\]

\[
= -\sum_{i=1}^{m} \alpha_i y_i.
\]
The maximum margin classifier

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

1. The weight vector can be expressed as

$$ w = \sum_{i=1}^{m} \alpha_i y_i \Phi(x_i) $$

with $\alpha \geq 0$. 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

$$ \alpha_i [y_i f_{w,w_0}(x_i) - 1] = 0. $$
The maximum margin classifier

The fact that for each example

$$\alpha_i [y_i f_{w,w_0}(x_i) - 1] = 0$$

means that:

Either $y_i f_{w,w_0}(x_i) = 1$ or $\alpha_i = 0$.

This means that examples fall into two groups.

1. Those for which $y_i f_{w,w_0}(x_i) = 1$.
   As the constraint used to maximize the margin was $y_i f_{w,w_0}(x_i) \geq 1$ 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 $y_i f_{w,w_0}(x_i) \neq 1$.
   These are non-support vectors and in this case it must be that $\alpha_i = 0$. 
The maximum margin classifier

Support vectors:

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

Remember that

\[ w = \sum_{i=1}^{m} \alpha_i y_i \Phi(x_i) \]

so the weight vector \( w \) only depends on the support vectors.

**ALSO:** the dual parameters \( \alpha \) can be used as an alternative set of weights. The overall classifier is

\[
h_{w, w_0}(x) = \text{sgn} \left( w_0 + w^T \Phi(x) \right) \\
= \text{sgn} \left( w_0 + \sum_{i=1}^{m} \alpha_i y_i \Phi^T(x_i) \Phi(x) \right) \\
= \text{sgn} \left( w_0 + \sum_{i=1}^{m} \alpha_i y_i K(x_i, x) \right)
\]

where \( K(x_i, x) = \Phi^T(x_i) \Phi(x) \) is called the kernel.
The maximum margin classifier

Remember where this process started:

The kernel is computing

\[ K(x, x') = \Phi^T(x)\Phi(x') \]

\[ = \sum_{i=1}^{k} \phi_i(x)\phi_i(x') \]

This is generally called an \textit{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:

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

$$= \text{sgn} \left( w_0 + \sum_{i=1}^{k} w_i \phi_i(x) \right)$$

$$= \text{sgn} \left( w_0 + \sum_{i=1}^{m} \alpha_i y_i \Phi^T(x_i) \Phi(x) \right)$$

$$= \text{sgn} \left( w_0 + \sum_{i=1}^{m} \alpha_i y_i K(x_i, x) \right)$$

What if $K(x, x')$ 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 $\alpha$.

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 $\sigma^2$ 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

   \[ L(w, w_0, \alpha) = \frac{1}{2} ||w||^2 - \sum_i \alpha_i (y_i f_{w, w_0}(x_i) - 1). \]

2. The weight vector is

   \[ w = \sum_i \alpha_i y_i \Phi(x_i). \]

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

\[ \tilde{L}(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j K(x_i, x_j) \]

such that \( \alpha \geq 0. \)
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 $\xi_i$, one for each example.

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

The constrained optimization problem was:

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

The constrained optimization problem is now modified to:

\[
\text{argmin}_{\mathbf{w}, w_0, \xi} \left( \frac{1}{2} \| \mathbf{w} \|^2 + C \sum_{i=1}^{m} \xi_i \right) \text{ such that } y_i f_{\mathbf{w}, w_0}(\mathbf{x}_i) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \text{ for } i = 1, \ldots, m.
\]

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:

Support Vector Machines

Spirals data, no noise

Spirals data, $\sigma^2 = 0.001$

RBF kernel, $C = 10$, $\sigma^2 = 0.01$

Polynomial kernel, $C = 10$, degree = 5
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 debateable 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.

\[ h_\theta(x) \]

It should take an attribute vector

\[ x^T = [x_1 \ x_2 \ \cdots \ x_n] \]

and label it.

We now denote a classifier by \( h_\theta(x) \) where \( \theta^T = (w \ p) \) denotes any weights \( w \) and (hyper)parameters \( p \).

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 $s$ of $m$ examples and a test set $s'$ 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 $h_\theta$ obtained using only $s$. How do we use $s'$ to assess our method’s performance?

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

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

where

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

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 $s$ of size $m$ containing $m^+$ positive examples and $m^-$ negative examples...
Unbalanced data

Define

1. The true positives

\[ P^+ = \{(x, +1) \in s | h(x) = +1\}, \quad \text{and} \quad p^+ = |P^+| \]

2. The false positives

\[ P^- = \{(x, -1) \in s | h(x) = +1\}, \quad \text{and} \quad p^- = |P^-| \]

3. The true negatives

\[ N^+ = \{(x, -1) \in s | h(x) = -1\}, \quad \text{and} \quad n^+ = |N^+| \]

4. The false negatives

\[ N^- = \{(x, +1) \in s | h(x) = -1\}, \quad \text{and} \quad n^- = |N^-| \]

Thus \( \hat{\text{er}}_s(h) = (p^+ + n^+)/m \).

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)

\[
MCC = \frac{p^+n^+ - p^-n^-}{\sqrt{(p^+ + p^-)(n^+ + n^-)(p^+ + n^-)(n^+ + p^-)}}
\]

2. F1 score

\[
F1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}
\]

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

\begin{align*}
C = 0.01, \quad \sigma^2 = 0.01 \\
C = 0.1, \quad \sigma^2 = 0.01 \\
C = 10, \quad \sigma^2 = 0.01 \\
C = 1000, \quad \sigma^2 = 0.01
\end{align*}
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 $s'$ 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 $s$ and with the hyperparameter set to $p_i$.

2. Assess the resulting $h_\theta$ by computing a suitable measure (for example accuracy, MCC or F1) using $v$.

Finally, select the best estimated value for $p$, use it to train a new $h_\theta$ using $s$ and $v$ combined, and assess its actual performance using $s'$.
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 divide our complete set into training set \( s \) and testing set \( s' \) as before.
But now instead of further subdividing \( s \) just once we divide it into \( n \) folds \( s^{(i)} \) each having \( m/n \) examples.

Typically \( n = 10 \) although other values are also used, for example if \( n = m \) we have leave-one-out crossvalidation.
Crossvalidation

Let $s_{-i}$ denote the set obtained from $s$ by removing $s^{(i)}$.

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

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

Then, 

$$\frac{1}{n} \sum_{i=1}^{n} \hat{e}_r_{s^{(i)}}(L_{s_{-i}, p})$$

is the $n$-fold crossvalidation error estimate.

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

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

Using crossvalidation to optimize the hyperparameters $C$ and $\sigma^2$. 
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*. 

---

**Machine Learning Commandments**

---

---
Comparing classifiers

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

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

Can I claim that the Discriminatron 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 *Introduction to Probability*:

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

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

then as $n \to \infty$

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

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

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

Re-arranging 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 $\sigma^2$ but it can be estimated using

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

Alternatively, when $X$ takes only values 0 or 1

$$\sigma^2 = \mathbb{E} [(X - \mu)^2] = \mathbb{E} [X^2] - \mu^2 = \mu(1 - \mu) \simeq \hat{X}_n(1 - \hat{X}_n).$$
Assessing a single classifier

The actual probability of error for a classifier \( h \) is

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

and we are estimating \( \text{er}(h) \) using the accuracy

\[
\hat{\text{er}}_s(h) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}[h(x_i) \neq y_i]
\]

for a test set \( s \).

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

\[
X_i = \mathbb{I}[h(x_i) \neq y_i].
\]
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

\[
\text{er}(h) = \hat{\text{er}}_s(h) \pm 1.96 \sqrt{\frac{\hat{\text{er}}_s(h)(1 - \hat{\text{er}}_s(h))}{m}}.
\]

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

\[
\text{er}(h) = 0.18 \pm 1.96 \sqrt{\frac{0.18(1 - 0.18)}{100}}
\]

\[
= 0.18 \pm 0.075.
\]

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

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

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 $\sigma_Y^2$ and mean $\mu_Y$.
2. The random variable $Y - \mu_Y$ has variance $\sigma_Y^2$ and mean 0.
3. It is a straightforward exercise to show that dividing a random variable having variance $\sigma^2$ by $\sigma$ 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 $h_1$ (*AIMLBlockChain Classifier*) and $h_2$ (*DeepHype Discriminator*) and I want to know something about the quantity

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

I estimate $d$ using

$$\hat{d} = \hat{\text{er}}_{s_1}(h_1) - \hat{\text{er}}_{s_2}(h_2)$$

where $s_1$ and $s_2$ 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*

\[
\mathbb{E} [\hat{\text{er}}_{s_1}(h_1) - \hat{\text{er}}_{s_2}(h_2)] = d.
\]
Comparing classifiers

Also notice:

1. The two parts of the estimate $\hat{er}_{s_1}(h_1)$ and $\hat{er}_{s_2}(h_2)$ 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 $\hat{er}_{s_1}(h_1)$ and $\hat{er}_{s_2}(h_2)$.

3. Adding Gaussians gives another Gaussian.

4. We can calculate a confidence interval for our estimate.

With probability 0.95

$$d = \hat{d} \pm 1.96 \sqrt{\frac{\hat{er}_{s_1}(h_1)(1 - \hat{er}_{s_1}(h_1))}{m_1} + \frac{\hat{er}_{s_2}(h_2)(1 - \hat{er}_{s_2}(h_2))}{m_2}}.$$ 

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

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

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

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 $s$ to hypothesis $h$.
- So we really want to know about the quantity

$$d = \mathbb{E}_{s \in S^m} [\text{er}(L_1(s)) - \text{er}(L_2(s))].$$

- 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 $s$ available and we can only estimate errors $\text{er}(h)$—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

\[
\text{s}^{(1)} \quad \text{s}^{(2)} \quad \ldots \quad \text{s}^{(n)}
\]

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

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

is the \( n \)-fold crossvalidation error estimate. Now we estimate \( d \) using
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 \( p \)

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

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

\[
\sigma_d = \sqrt{\frac{1}{n(n-1)} \sum_{i=1}^{n} (d_i - \hat{d})^2}
\]

where

\[
d_i = \hat{e}_{r_s(i)}(L_1(s_{-i})) - \hat{e}_{r_s(i)}(L_2(s_{-i})).
\]
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 $h_w(x)$. (In fact this can be *pretty much any* parameterized function we like.)
- A training sequence $s^T = [(x_1, y_1) \ldots (x_m, y_m)]$, split into
  $$y = (y_1 \ y_2 \ \cdots \ y_m)$$
  and
  $$X = (x_1 \ x_2 \ \cdots \ x_m).$$
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:

\[ \Pr(C|x, s) = \int \Pr(C|w, x) p(w|s) \, dw. \]

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

\[ p(Y|x, s) = \int p(Y|w, x) p(w|s) \, dw \]

- \( s \) is the training set.
- \( x \) is a new example to be classified.
- \( Y \) is the RV representing the prediction for \( x \).
The Bayesian approach to neural networks

It turns out that *if you try to incorporate* the density \( p(x) \) 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 \( p(Y|w, x) \).
3. However, this seems to be uniformly disliked—writing \( p(Y|w) \) for an expression that still depends on \( x \) seems confusing.
4. Solution: write \( p(Y|w; x) \) instead. Note the *semi-colon*!

So we’re actually going to look at

\[
p(Y|y; x, X) = \int p(Y|w; x) p(w|y; X) \, dw
\]

*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

\[ p(w | y; X) \propto p(y | w; X)p(w). \]

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

The posterior density \( p(w | y; X) \) becomes more localised.
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 $p(Y|w; x)$ is.
2. **STEP 2**: remind ourselves what $p(w|y; X)$ 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

\[ y = h_w(x) + \epsilon \]

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

\[
p(Y|w; x) = \frac{1}{\sqrt{2\pi\sigma^2_n}} \exp \left( -\frac{1}{2\sigma^2_n} (Y - h_w(x))^2 \right).
\]

Here, the subscript in \( \sigma^2_n \) reminds us that it’s the variance of the noise.

Traditionally this is re-written using the hyperparameter

\[
\beta = \frac{1}{\sigma^2_n}
\]

so the likelihood is

\[
p(Y|w; x) \propto \exp \left( -\frac{\beta}{2} (Y - h_w(x))^2 \right).
\]
**The Bayesian approach to neural networks**

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

\[
p(w|y; X) \propto p(y|w; X)p(w)
\]

and as before, the likelihood is

\[
p(y|w; X) \propto \exp \left( -\frac{\beta}{2} \sum_{i=1}^{m} (y_i - h_w(x_i))^2 \right)
\]

\[= \exp \left( -\beta E(w) \right)
\]

and using a Gaussian prior with mean 0 and covariance \(\Sigma = \sigma^2 I\) gives

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

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

\[
p(w|y; X) = \frac{1}{Z(\alpha, \beta)} \exp \left( - \left( \frac{\alpha||w||^2}{2} + \beta E(w) \right) \right).
\]
What’s going on? Turning prior into posterior

Considering the central part of $p(w|y; X)$:

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

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 $w_{ML}$ is a good approximation to $w_{MAP}$. 
The Bayesian approach to neural networks

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

\[
I \propto \int \exp \left( -\frac{\beta}{2} (Y - h_w(x))^2 \right) \exp \left( - \left( \frac{\alpha ||w||^2}{2} + \beta E(w) \right) \right) \, dw.
\]

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

\[ \int F(w)p(w|y; X) \, dw \]

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 $p(w|y; X)$.
2. We can use Monte Carlo methods.

We’ll be taking a look at both possibilities.
Method 1: approximation to \( p(w|y; X) \)

\[
I \propto \int \exp \left( \frac{-\beta}{2} (Y - h_w(x))^2 \right) \exp \left( -\left( \frac{\alpha ||w||^2}{2} + \beta E(w) \right) \right) \, dw.
\]

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

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

at the *maximum a posteriori* weights \( w_{\text{MAP}} \).

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 \( f : \mathbb{R} \rightarrow \mathbb{R} \) is

\[
f(x) \approx f(x_0) + \frac{1}{1!}(x - x_0)f'(x_0) + \frac{1}{2!}(x - x_0)^2f''(x_0) + \cdots + \frac{1}{k!}(x - x_0)^k f^k(x_0).
\]

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

\[
\exp(-f(x))
\]

where

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

This has a *form similar to* \( S(w) \), but in one dimension.
Reminder: Taylor expansion

The functions of interest look like this:

![Graph of f(x) and exp(-f(x))]

By replacing $-f(x)$ with its Taylor expansion about its maximum, which is at $x_{\text{max}} = 2.1437$

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

\[
\begin{align*}
    f(x) & \approx f(x_0) + \frac{1}{1!}(x - x_0)^T \nabla f(x)|_{x_0} \\
    & \quad + \frac{1}{2!}(x - x_0)^T \nabla^2 f(x)|_{x_0} (x - x_0)
\end{align*}
\]

where \( \nabla \) denotes *gradient*

\[
\nabla f(x) = \left( \frac{\partial f(x)}{\partial x_1} \quad \frac{\partial f(x)}{\partial x_2} \quad \ldots \quad \frac{\partial f(x)}{\partial x_n} \right)
\]

and \( \nabla^2 f(x) \) is the matrix with elements

\[
M_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}.
\]

*(Looks complicated, but it’s just the obvious extension of the 1-dimensional case.)*
Method 1: approximation to $p(w|y; X)$

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

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

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

This is because

$$A = \nabla \nabla \left( \frac{\alpha||w||^2}{2} + \beta E(w) \right)|_{w_{\text{MAP}}} = \alpha I + \beta \nabla \nabla E(w_{\text{MAP}}).$$
Method 1: approximation to $p(w|y; X)$

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

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

It’s also likely to be straightforward to compute $\nabla \nabla E(w)$:

The quantity $\nabla \nabla E(w)$ 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 $x$, here is a useful standard integral:

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

$$\int_{\mathbb{R}^n} \exp \left( -\frac{1}{2} \left( x^T A x + x^T b + c \right) \right) dx$$

$$= (2\pi)^{n/2} |A|^{-1/2} \exp \left( -\frac{1}{2} \left( c - \frac{b^T A^{-1} b}{4} \right) \right).$$

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 $p(w|y; X)$

Defining

$$\Delta w = w - w_{\text{MAP}}$$

we now have an approximation

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

Using the *BIG INTEGRAL*

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

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 $p(w|y; X)$

$$I \propto \int \exp \left( -\frac{\beta}{2} (Y - h_w(x))^2 \right) \exp \left( -\frac{1}{2} \Delta w^T A \Delta w \right) \, dw.$$  

There is still no solution! We need another approximation...

We can introduce a linear approximation\(^2\) of $h_w(x)$ at $w_{\text{MAP}}$:

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

where $g = \nabla h_w(x)|_{w_{\text{MAP}}}$.  

(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 $p(w|y; X)$ is narrow, which depends on $A$.  

180
Method 1: second approximation

This leads to

\[
p(Y|y; x, X) \propto \int \exp \left( -\frac{\beta}{2} (Y - h_{\text{wMAP}}(x) - g^T \Delta w)^2 - \frac{1}{2} \Delta w^T A \Delta w \right) \, dw.
\]

SUCCESS!!!

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

\[
p(Y|y; x, X) \sim \frac{1}{\sqrt{2\pi \sigma_Y^2}} \exp \left( -\frac{(Y - h_{\text{wMAP}}(x))^2}{2\sigma_Y^2} \right)
\]

where

\[
\sigma_Y^2 = \frac{1}{\beta} + g^T A^{-1} g.
\]
Method 1: final expression

Hooray! But what does it mean?

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

\[
p(Y|y; x, X) \text{ peaks at } h_{\text{wMAP}}(x).
\]

That is, the MAP solution.

The variance \( \sigma^2_Y \) can be interpreted as a measure of certainty:

The first term of \( \sigma^2_Y \) is \( 1/\beta \) and corresponds to the noise.
The second term of \( \sigma^2_Y \) is \( g^T A^{-1} g \) and corresponds to the width of \( p(w|y; X). \)
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_.

183
Method II: Markov chain Monte Carlo (MCMC) methods

The second solution to the problem of performing integrals

\[ I = \int F(w)p(w|y; X)dw \]

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

\[ I \approx \frac{1}{N} \sum_{i=1}^{N} F(w_i) \]

where the \( w_i \) have distribution \( p(w|y; X) \). Unfortunately, generating \( w_i \) with a given distribution can be non-trivial.
MCMC methods

A simple technique is to introduce a random walk, so

\[ w_{i+1} = w_i + \epsilon \]

where \( \epsilon \) is zero mean spherical Gaussian and has small variance. Obviously the sequence \( w_i \) 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 \( p(w_{i+1}|y; X) > p(w_i|y; X) \) then accept the step.
2. Else accept the step with probability \( \frac{p(w_{i+1}|y; X)}{p(w_i|y; X)} \).

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

A (very) brief introduction to how to learn hyperparameters

So far in our coverage of the Bayesian approach to neural networks, the hyperparameters $\alpha$ and $\beta$ were assumed to be known and fixed.

- But this is not a good assumption because...
- ...$\alpha$ corresponds to the width of the prior and $\beta$ 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 $x$ and $X$ as leaving them in starts to make everything cluttered.
The Bayesian approach to neural networks

The prior and likelihood depend on $\alpha$ and $\beta$ respectively so we now make this clear and write

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

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:

$$p(\alpha, \beta|y) = \frac{p(y|\alpha, \beta)p(\alpha, \beta)}{p(y)}.$$
Hierarchical Bayes and the evidence

If we know \( p(\alpha, \beta | y) \) then a straightforward approach is to use the values for \( \alpha \) and \( \beta \) that maximise it:

\[
\arg\max_{\alpha, \beta} p(\alpha, \beta | y).
\]

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

\[ p(y | \alpha, \beta). \]

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

The quantity

\[ p(y|\alpha, \beta) \]

is called the evidence or marginal likelihood.

When we re-wrote our earlier equation for the posterior density of the weights, making \( \alpha \) and \( \beta \) explicit, we found

\[
p(w|y, \alpha, \beta) = \frac{p(y|w, \beta)p(w|\alpha)}{p(y|\alpha, \beta)}.
\]

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.
There is an alternative approach to Bayesian regression and classification:

The fundamental idea is to *not* think in terms of weights $w$ *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 $x$ and $X$ to keep the notation simple.

We have seen that inference can be performed by:

1. Computing the posterior density $p(w|y)$ of the parameters given the observed labels.
2. Computing the Bayes-optimal prediction

$$p(Y|y) = \int p(Y|w)p(w|y) dw$$

which is the expected value of the likelihood for a new point $x$.
3. Choosing any hyperparameters $p$ using the evidence $p(y|p)$.

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

$$p(Y|y) = \int p(Y|f)p(f|y) \, df$$

in any sensible way?
Can we change the equation for prediction to

\[ p(Y|y) = \int p(Y|f)p(f|y) \, df \]

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 $p(w)$.

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** $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$.

We then write $F \sim \text{GP}(m, k)$ to denote that $F$ is a GP.

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

\[
m(x) = \mathbb{E}_{f \sim F} [f(x)] \\
k(x_1, x_2) = \mathbb{E}_{f \sim F} [(f(x_1) - m(x_1))(f(x_2) - m(x_2))]
\]
GP priors

Squared exponential prior, $l = 1/2$

γ-exponential prior, $l = \gamma = 1/2$

Rational quadratic prior, $l = \alpha = 3$

Neural network prior, $\sigma_1 = 15, \sigma_2 = 10$
Covariance functions

Polynomial:
\[ k(x_1, x_2) = (c + x_1^T x_2)^k \]

Exponential:
\[ k(x_1, x_2) = \exp \left( -\frac{|x_1 - x_2|}{l} \right) \]

Squared exponential:
\[ k(x_1, x_2) = \exp \left( -\frac{|x_1 - x_2|^2}{2l^2} \right) \]

Gamma exponential:
\[ k(x_1, x_2) = \exp \left( -\left( \frac{|x_1 - x_2|}{l} \right)^\gamma \right) \]
Covariance functions

Rational quadratic:

\[ k(x_1, x_2) = \left( 1 + \frac{|x_1 - x_2|^2}{2\alpha l^2} \right)^{-\alpha} \]

Exponential:

\[ k(x_1, x_2) = \sin^{-1}\left( \frac{2(x'_1)^T \Sigma x'_2}{((1 + 2(x'_1)^T \Sigma x'_1)(1 + 2(x'_2)^T \Sigma x'_2))^{1/2}} \right) \]

where \((x')^T = [1 \ x^T]\).

As usual these have associated hyperparameters.
These have to be dealt with correctly as always.
Say we have some data

\[ y_i = f(x_i) \]

for \( i = 1, \ldots, m \) and \( f \sim \text{GP}(m, k) \). (Remember, the \( x_i \) are fixed, not RVs.)

Any finite set of points must be jointly Gaussian. So

\[ p(y) = \mathcal{N}(m, K) \]

where

\[ m^T = [m(x_1) \, \cdots \, m(x_m)] \]

and \( K \) is the Gram matrix \( K_{ij} = k(x_i, x_j) \).

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

Note 2: from now on we will assume \( m(x) = 0 \). (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

\[ y_i = f(x_i) + \epsilon_i. \]

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

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

**BUT:** in order to do prediction we actually need to involve a new point \( x' \), 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

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

where

$$\mathbf{K}' = \begin{bmatrix} k & \mathbf{k}^T \\ \mathbf{k} & \mathbf{K} + \sigma^2 \mathbf{I} \end{bmatrix}$$

$$\mathbf{k}^T = \begin{bmatrix} k(x', x_1) & \cdots & k(x', x_m) \end{bmatrix}$$

$$k = k(x', x') + \sigma^2.$$

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 $\sigma^2$ 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 \( x \sim \mathcal{N}(\mu, \Sigma) \)

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

Split \( x \) so

\[
x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}
\]

and correspondingly

\[
\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}
\quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}.
\]

What are \( p(x_1) \) and \( p(x_1|x_2) \)?
Gaussian density: marginals and conditionals

Define the precision matrix

\[
\Lambda = \Sigma^{-1} = \begin{bmatrix}
\Lambda_{11} & \Lambda_{12} \\
\Lambda_{21} & \Lambda_{22}
\end{bmatrix}.
\]

It is possible to show that

\[
p(x_1) = \mathcal{N}(\mu_1, \Sigma_{11})
\]

\[
p(x_1|x_2) = \mathcal{N}(\mu_1 - \Lambda_{11}^{-1}\Lambda_{12}(x_2 - \mu_2), \Lambda_{11}^{-1}).
\]
Inverting a block matrix

In the last slide, we see:

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

Re-writing $\Sigma$ as

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

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

$$\Lambda_{11} = A'$$
$$\Lambda_{12} = -A'BD^{-1}$$
$$\Lambda_{21} = -D^{-1}CA'$$
$$\Lambda_{22} = D^{-1} + D^{-1}CA'BBD^{-1}$$

where

$$A' = (A - BD^{-1}C)^{-1}. $$
GP regression

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

Because everything is Gaussian this turns out to be easy:

\[
p(y', y) = \mathcal{N}(0, K')
\]

\[
K' = \begin{bmatrix} k & k^T \\ k & L \end{bmatrix}
\]

\[
L = K + \sigma^2 I.
\]

From these we want to know \( p(y'|y) \).

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

\[
p(y'|y) = \mathcal{N}(k^T L^{-1} y, k^T L^{-1} k).
\]

\begin{itemize}
\item \textbf{Mean}
\item \textbf{Variance}
\end{itemize}
GP regression

Squared exponential prior, $l = 1/2$

γ-exponential prior, $l = \gamma = 1/2$

Rational quadratic prior, $l = a = 3$

Neural network prior, $\sigma_1 = 15, \sigma_2 = 10$
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 \( p \), which in this case are any parameters associated with \( k(x_1, x_2) \) along with \( \sigma^2 \), then we’ve just computed

\[
p(y'|y, p) = \frac{p(y', y|p)}{p(y|p)}.
\]

The denominator is the marginal likelihood, and we computed it above on slide 201:

\[
p(y|p) = \mathcal{N}(0, L) = \frac{1}{\sqrt{(2\pi)^m |L|}} \exp \left( -\frac{1}{2}y^T L^{-1}y \right).
\]
Learning the hyperparameters

As usual this looks nicer if we consider its log

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

This is a rare beast:

1. It’s a sensible formula that tells you how good a set \( p \) 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.
Part IV

Unsupervised learning
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 $\{x_i\}$ of $m$ points, guess that there are $K$ clusters. Here $K = 3$.

Chose at random $K$ centre points $c_j$ for the clusters. Then iterate as follows:

1. Divide $\{x_i\}$ into $K$ clusters, so each point is associated with the closest centre:
   \[ x_i \in C_j \iff \forall k \, \|x_i - c_j\| \leq \|x_i - c_k\|. \]
   Call these clusters $C_1, \ldots, C_K$.

2. Update the cluster centres to be the average of the associated points:
   \[ c_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i. \]
The $K$-means algorithm

Actual data for 3 clusters

1 iteration

2 iterations

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

$$p(x|\theta) = \sum_{k=1}^{K} \pi_k p(x|\mu_k, \Sigma_k)$$

where $\theta = \{\pi, \mu_1, \Sigma_1, \ldots, \mu_K, \Sigma_K\}$ and typically $p(x|\mu, \Sigma) = \mathcal{N}(\mu, \Sigma)$. 
This leads to a log-likelihood for \( m \) points of

\[
\log p(X|\theta) = \log \prod_{i=1}^{m} p(x_i|\theta)
\]

\[
= \sum_{i=1}^{m} \log p(x_i|\theta)
\]

\[
= \sum_{i=1}^{m} \log \left( \sum_{k=1}^{K} \pi_k p(x_i|\mu_k, \Sigma_k) \right)
\]

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 $x_i$ introduce the latent variable $z_i$ where

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

and

$$z_i^{(j)} = \begin{cases} 1 & \text{if } x_i \text{ was generated by cluster } j \\ 0 & \text{otherwise} \end{cases}$$
Clustering as maximum-likelihood

Having introduced the $z_i$ we can use the marginalization trick and write

$$\log p(X|\theta) = \log \sum_Z p(X, Z|\theta)$$

$$= \log \sum_Z p(X|Z, \theta)p(Z|\theta)$$

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

*Why is this?*

First, if I know which cluster generated $x_i$ then its probability is just that for the corresponding Gaussian

$$p(x_i|z, \theta) = \prod_{k=1}^K \left[p(x_i|\mu_k, \Sigma_k)^{z_i(k)}\right]$$

and similarly

$$p(z_i|\theta) = \prod_{k=1}^K \left[\pi_k^{z_i(k)}\right]$$
In other words, if you treat the $z_i$ as *observed* rather than *latent* then you can write

$$p(x_i, z_i | \theta) = \prod_{k=1}^{K} \left[ p(x_i | \mu_k, \Sigma_k \pi_k)^{z_i^{(k)}} \right]$$

$$\log p(X, Z | \theta) = \log \prod_{i=1}^{m} p(x_i, z_i | \theta)$$

$$= \log \prod_{i=1}^{m} \prod_{k=1}^{K} \left[ p(x_i | \mu_k, \Sigma_k \pi_k)^{z_i^{(k)}} \right]$$
Clustering as maximum-likelihood

Consequently

\[
\log p(X, Z|\theta) = \sum_{i=1}^{m} \sum_{k=1}^{K} z_i^{(k)} \left( \log p(x_i|\mu_k, \Sigma_k) + \log \pi_k \right)
\]

What have we achieved so far?

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

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 $q(Z)$ be *any* distribution on the *latent variables*. Write

$$
\sum_Z q(Z) \log \frac{p(X,Z|\theta)}{q(Z)} = \sum_Z q(Z) \log \frac{p(Z|X,\theta)p(X|\theta)}{q(Z)} 
$$

$$
= \sum_Z q(Z) \left( \log \frac{p(Z|X,\theta)}{q(Z)} + \log p(X|\theta) \right) 
$$

$$
= -D_{KL}[q(Z)||p(Z|X,\theta)] + \sum_Z q(Z) \log p(X|\theta) 
$$

$$
= -D_{KL}[q(Z)||p(Z|X,\theta)] + \log p(X|\theta). 
$$

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

---

221
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

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

It has the important properties that:

1. It is non-negative
   $$D_{KL}(p||q) \geq 0.$$

2. It is 0 precisely when the distributions are equal
   $$D_{KL}[p||q] = 0 \text{ if and only if } p = q.$$
The EM algorithm

If we also define

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

then we can re-arrange the last expression to get

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

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

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

The EM algorithm works as follows:

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

Let’s look at the two steps separately.

Say we have $\theta_t$ at time $t$ in the iteration.

For the **E step**, we have $\theta_t$ fixed and

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

so this is easy!

1. As $\theta_t$ is fixed, so is $\log p(X|\theta_t)$.
2. So to maximize $L[q, \theta_t]$ 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

$$q_{t+1}(Z) = p(Z|X, \theta_t).$$
The EM algorithm

For the \textit{M step} we have

$$L[q, \theta] = \sum_Z q(Z) \log p(X, Z|\theta) - \sum_Z q(Z) \log q(Z)$$

where the second term (the entropy of \(q(Z)\)) doesn’t depend on \(\theta\).

We fix \(q_{t+1}(Z) = p(Z|X, \theta_t)\). We now choose \(\theta_{t+1}\) as

\[
\theta_{t+1} = \arg\max_{\theta} \sum_Z p(Z|X, \theta_t) \log p(X, Z|\theta_t) \\
= \arg\max_{\theta} \mathbb{E}_Z [\log p(X, Z|\theta_t)]
\]
The EM algorithm

We saw earlier that

\[
\log p(X, Z|\theta_t) = \sum_{i=1}^{m} \sum_{k=1}^{K} z_i^{(k)} \left( \log p(x_i|\mu_k, \Sigma_k) + \log \pi_k \right)
\]

where \( \theta \) collects all the parameters

\[
\theta_t = \{ \pi, \mu_1, \Sigma_1, \ldots, \mu_K, \Sigma_K \}.
\]

Note that the parameters \( \pi, \mu_k \) and \( \Sigma_k \) all have an implicit time \( t \) attached, but we avoid writing it to keep the notation manageable.

So: this step looks a little trickier: we need to maximize the expected value of this expression for the distribution \( p(Z|X, \theta_t) \).
The EM algorithm

It’s not as bad as it looks:

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

Thus

$$\mathbb{E}_Z \left[z_i^{(k)}\right] = \sum_Z z_i^{(k)} p(Z|X, \theta_t)$$

$$= \sum_{z_1} \cdots \sum_{z_m} z_i^{(k)} p(z_1, \ldots, z_m|X, \theta_t)$$

$$= \sum_{z_i} z_i^{(k)} p(z_i|X, \theta_t) \text{ (marginalizing)}$$

$$= \sum_{z_i^{(k)} \in \{0,1\}} z_i^{(k)} p(z_i^{(k)}|X, \theta_t) \text{ (marginalizing again)}$$

$$= p(z_i^{(k)} = 1|X, \theta_t).$$
The EM algorithm

So

\[ \mathbb{E}_Z \left[ z_i^{(k)} \right] = p(z_i^{(k)} = 1 | X, \theta_t) \]

\[ = \frac{p(z_i^{(k)} = 1, x_i | \theta_t)}{p(x_i | \theta_t)} \text{ using conditional independence} \]

\[ = \frac{\pi_k p(x_i | \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j p(x_i | \mu_j, \Sigma_j)}. \]

As a shorthand, define

\[ \gamma_i^{(k)} = \frac{\pi_k p(x_i | \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j p(x_i | \mu_j, \Sigma_j)} \]

so the expression we’ve arrived at is

\[ \theta_{t+1} = \arg\max_{\theta} \sum_{i=1}^{m} \sum_{k=1}^{K} \gamma_i^{(k)} \left( \log p(x_i | \mu_k, \Sigma_k) + \log \pi_k \right). \]
The EM algorithm

The EM algorithm for a mixture model summarized:

• We want to find $\theta$ to maximize $\log p(X|\theta)$.
• But that’s not tractable.
• So we introduce an arbitrary distribution $q$ and obtain a lower bound
  
  $$L(q, \theta) \leq \log p(X|\theta).$$

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

\[
\theta_{t+1} = \arg\max_\theta \sum_{i=1}^m \sum_{k=1}^K \gamma_i^{(k)} \left( \log p(x_i|\mu_k, \Sigma_k) + \log \pi_k \right)
\]
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

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

are:

$$\pi'_j = \frac{\sum_{i=1}^{m} \gamma_{ij}}{m}$$

$$\mu'_j = \frac{\sum_{i=1}^{m} \gamma_{ij} x_i}{\sum_{i=1}^{m} \gamma_{ij}}$$

$$\Sigma'_j = \frac{\sum_{i=1}^{m} \gamma_{ij} (x_i - \mu'_j)(x_i - \mu'_j)^T}{\sum_{i=1}^{m} \gamma_{ij}}.$$
The EM algorithm for a mixture of Gaussians
Part V

Bayesian networks

Markov random fields

Copyright © Sean Holden 2022-23.
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 $L$ are all the RVs not in the sets $Q$ or $O$.

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

Query $Q = \{Q_1, Q_2, \ldots, Q_q\}$

Observed $o = \{o_1, o_2, \ldots, o_m\}$

Latent variables $L = \{L_1, L_2, \ldots, L_l\}$

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

$$\Pr(Q|o_1, o_2, \ldots, o_m) = \sum_L \Pr(Q, L|o_1, o_2, \ldots, o_m)$$

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

Knowledge base
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

$$
\Pr(Q|o', o_1, o_2, \ldots, o_m) = \frac{1}{Z} \Pr(o'|Q, o_1, o_2, \ldots, o_m) \Pr(Q|o_1, o_2, \ldots, o_m)
$$

After $O'$ observed

Before $O'$ 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 $Pr(V)$ 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*.
- \( \Pr(X | Y = y \land Z = z) \) might be very different.

Conditional probability is *not* analogous to *logical implication*. 
Implication and conditional probability

In general, it is difficult to relate \textit{implication} to \textit{conditional probability}.

\[
\Pr(A \rightarrow B) = \Pr(\neg A \lor B)
\]

\[
\Pr(A|B) = \frac{\Pr(A \land B)}{\Pr(B)}
\]

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

With implication,

\[
\Pr(\text{fish} \rightarrow \neg \text{swim}) = \Pr(\neg \text{fish} \lor \neg \text{swim}) = \text{LARGE!}
\]

With conditional probability,

\[
\Pr(\neg \text{swim}|\text{fish}) = \frac{\Pr(\neg \text{swim} \land \text{fish})}{\Pr(\text{fish})} = \text{SMALL!}
\]
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 $\Pr(V)$. 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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>0.014</td>
<td>0.028</td>
<td>0.042</td>
<td>0.057</td>
<td>0.071</td>
<td>0.086</td>
</tr>
<tr>
<td>$T$</td>
<td>0.033</td>
<td>0.067</td>
<td>0.1</td>
<td>0.133</td>
<td>0.167</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Here $\Pr(\text{Coin} = H) = 0.3$ and the die has probability $i/21$ for the $i$th outcome.
Exploiting independence

*BUT*: if we assume the outcomes are independent then

\[ \Pr(Coin, Dice) = \Pr(Coin) \Pr(Dice) \]

Where \( \Pr(Coin) \) has two numbers and \( \Pr(Dice) \) has six.

So instead of 12 numbers we only need 8.
Exploiting independence

A slightly more complex example:

<table>
<thead>
<tr>
<th></th>
<th>CP</th>
<th>¬CP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SB</td>
<td>¬SB</td>
</tr>
<tr>
<td>HD</td>
<td>0.024</td>
<td>0.006</td>
</tr>
<tr>
<td>¬HD</td>
<td>0.0019</td>
<td>0.0076</td>
</tr>
</tbody>
</table>

- 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:

\[ TC = \{\text{Oxford, Cambridge, Draw}\}. \]
Exploiting independence

Now

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

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

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

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

\[ \Pr(CP, SB|HD) = \Pr(CP|HD) \Pr(SB|HD) \]

which simplifies matters.

\[ \text{Conditional independence: } A \perp B|C \]

- \(A\) 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!

\[ \Pr(HD|CP, SB) \propto \Pr(CP|HD) \Pr(SB|HD) \Pr(HD) \]
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

<table>
<thead>
<tr>
<th>Pr(Climber)</th>
<th>Pr(Goose)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes: 0.05</td>
<td>Yes: 0.2</td>
</tr>
<tr>
<td>No: 0.95</td>
<td>No: 0.8</td>
</tr>
</tbody>
</table>

| Pr(L1|A)      | Pr(L2|A)      |
|-----------|-------------|
| a         | 0.99        |
| ¬a        | 0.08        |
| a         | 0.6         |
| ¬a        | 0.001       |

| Pr(A|C,G) |
|---------|
| C | G | Pr(A|C,G) |
| Y | Y | 0.98     |
| N | N | 0.08     |
| Y | N | 0.96     |
| N | Y | 0.2      |
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

\[ \Pr(N_i | \text{parents}(N_i)) . \]

- 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 \( \Pr (N_i | \text{parents}(N_i)) \) 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

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

For example

\[
\Pr (\neg C, \neg G, A, L_1, L_2) = \Pr (L_1 | A) \Pr (L_2 | A) \Pr (A | \neg C, \neg G) \Pr (\neg C) \Pr (\neg G) = 0.99 \times 0.6 \times 0.08 \times 0.95 \times 0.8.
\]
Semantics

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

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

Repeating this gives

\[
\Pr(N_1, \ldots, N_n) = \Pr(N_n|N_{n-1}, \ldots, N_1) \Pr(N_{n-1}|N_{n-2}, \ldots, N_1) \cdots \Pr(N_1) = \prod_{i=1}^{n} \Pr(N_i|N_{i-1}, \ldots, N_1).
\]

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 it’s 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…
Path $p$ is *blocked* with respect to $Z$ if:

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

   ![Diagram](image1)

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

   ![Diagram](image2)

   (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 descendants is in $Z$:

   ![Diagram](image3)
Semantics: what’s REALLY going on here?

Finally:

1. $X$ and $Y$ are \emph{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 \emph{d-separated by $Z$} then $X \perp Y \mid Z$. 
More complex nodes

How do we represent

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

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.

\[ \text{Pr}(\text{Speed of car}|\text{Throttle position, Tuned engine}) \]

where SC and TP are continuous and TE is Boolean.

- For a specific setting of \( \text{ET} = \text{true} \) it might be the case that SC increases with TP, but that some uncertainty is involved

\[ \text{Pr} (\text{SC}|\text{TP, et}) = N(g_{\text{et}} \text{TP} + c_{\text{et}}, \sigma_{\text{et}}^2). \]

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

\[ \text{Pr} (\text{SC}|\text{TP, et}) = N(g_{\text{et}} \text{TP} + c_{\text{et}}, \sigma_{\text{et}}^2). \]

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

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

We could for example use the probit distribution

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

where

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

and $N$ is the Gaussian density with zero mean and variance 1.
More complex nodes

The probit distribution

\( \Phi(x) \)

Pr(GRC = true|size) with \( t = 100 \) and different values of \( s \)
Basic inference

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

\[ \Pr(Q|o_1, o_2, \ldots, o_m) = \frac{1}{Z} \sum_L \Pr(Q, L, o_1, o_2, \ldots, o_m) \]

where

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

As the BN fully describes the full joint distribution

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

it can be used to perform inference in the obvious way

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

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

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

\[
\text{Pr}(C, G, A, L_1, L_2) = \text{Pr}(C) \text{Pr}(G) \text{Pr}(A|C, G) \text{Pr}(L_1|A) \text{Pr}(L_2|A).
\]
Performing exact inference

Consider the computation of the query \( \Pr (C|l_1, l_2) \)

We have

\[
\Pr (C|l_1, l_2) \propto \sum_{A} \sum_{G} \Pr (C) \Pr (G) \Pr (A|C, G) \Pr (l_1|A) \Pr (l_2|A).
\]

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

\[
\Pr(C|l1, l2) \propto \sum_A \sum_G \Pr(C) \Pr(G) \Pr(A|C, G) \Pr(l1|A) \Pr(l2|A)
\]

\[
= \frac{1}{Z} \Pr(C') \sum_A \Pr(l1|A) \Pr(l2|A) \sum_G \Pr(G) \Pr(A|C, G)
\]

\[
= \frac{1}{Z} \Pr(C') \sum_G \Pr(G) \sum_A \Pr(A|C, G) \Pr(l1|A) \Pr(l2|A).
\]

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:

\[
\Pr(C') \sum_{G} \Pr(G) \sum_{A} \Pr(A|C', G) \Pr(l_1|A) \Pr(l_2|A)
\]

where \(C, G, A, L_1, L_2\) 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.

\[
\Pr(l_1|A) \text{ depends on the value of } A. \text{ We store it as a table } F_{L_1}(A). \text{ Similarly for } \Pr(l_2|A).
\]

\[
F_{L_1}(A) = \begin{pmatrix} 0.99 \\ 0.08 \end{pmatrix} \quad F_{L_2}(A) = \begin{pmatrix} 0.6 \\ 0.001 \end{pmatrix}
\]

as \(\Pr(l_1|a) = 0.99, \ Pr(l_1|\neg a) = 0.08\) and so on.
Performing exact inference: variable elimination

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

\[
\begin{array}{ccc|c}
A & C & G & F_A(A, C, G) \\
\hline
\top & \top & \top & 0.98 \\
\top & \top & \bot & 0.96 \\
\top & \bot & \top & 0.2 \\
\top & \bot & \bot & 0.08 \\
\bot & \top & \top & 0.02 \\
\bot & \top & \bot & 0.04 \\
\bot & \bot & \top & 0.8 \\
\bot & \bot & \bot & 0.92 \\
\end{array}
\]

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

\[
F_A(A, C, G) F_{l1}(A) F_{l2}(A)
\]

in a reasonable way?
Performing exact inference: variable elimination

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

\[
\text{Pr}(C') \sum_G \text{Pr}(G) \sum_A \text{Pr}(A|C, G) \text{Pr}(l1|A) \text{Pr}(l2|A)
\]

note that:

1. The values of the product

\[
\text{Pr}(A|C, G) \text{Pr}(l1|A) \text{Pr}(l2|A)
\]

in the summation over \(A\) depend on the values of \(C\) and \(G\) external to it, and the values of \(A\).

2. So

\[
F_A(A, C, G)F_{L1}(A)F_{L2}(A)
\]

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

\[ F(A, B)F(B, C) = F(A, B, C) \]

where

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>0.3</td>
<td>T</td>
<td>0.1</td>
<td>T</td>
<td>0.3 \times 0.1</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>⊥</td>
<td>0.9</td>
<td>⊥</td>
<td>0.8</td>
<td>⊥</td>
<td>0.3 \times 0.8</td>
<td></td>
</tr>
<tr>
<td>⊥</td>
<td>T</td>
<td>0.4</td>
<td>⊥</td>
<td>0.8</td>
<td>⊥</td>
<td>0.9 \times 0.8</td>
<td></td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>0.1</td>
<td>⊥</td>
<td>0.3</td>
<td>⊥</td>
<td>0.9 \times 0.3</td>
<td></td>
</tr>
</tbody>
</table>

268
Performing exact inference: variable elimination

This process gives us

\[ F_A(A, C, G)F_{L1}(A)F_{L2}(A) = \]

<table>
<thead>
<tr>
<th>( A )</th>
<th>( C )</th>
<th>( G )</th>
<th>Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
<td>0.98 ( \times ) 0.99 ( \times ) 0.6</td>
</tr>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( \bot )</td>
<td>0.96 ( \times ) 0.99 ( \times ) 0.6</td>
</tr>
<tr>
<td>( \top )</td>
<td>( \bot )</td>
<td>( \top )</td>
<td>0.2 ( \times ) 0.99 ( \times ) 0.6</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( \top )</td>
<td>( \bot )</td>
<td>0.08 ( \times ) 0.99 ( \times ) 0.6</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>0.02 ( \times ) 0.08 ( \times ) 0.001</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>0.04 ( \times ) 0.08 ( \times ) 0.001</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \top )</td>
<td>0.8 ( \times ) 0.08 ( \times ) 0.001</td>
</tr>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>0.92 ( \times ) 0.08 ( \times ) 0.001</td>
</tr>
</tbody>
</table>
Performing exact inference: variable elimination

How about

$$F_{A,L_1,L_2}(C, G) = \sum_A F_A(A, C, G)F_{L_1}(A)F_{L_2}(A)?$$

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

$$\sum_A F_A(A, C, G)F_{L_1}(A)F_{L_2}(A) = F_A(a, C, G)F_{L_1}(a)F_{L_2}(a)$$

$$+ F_A(\neg a, C, G)F_{L_1}(\neg a)F_{L_2}(\neg a)$$

where

$$F_A(a, C, G) = \begin{array}{c|cc} C & G \\ \hline \top & \top & 0.98 \\ \top & \bot & 0.96 \\ \bot & \top & 0.2 \\ \bot & \bot & 0.08 \end{array} \quad F_{L_1}(a) = 0.99 \quad F_{L_2}(a) = 0.6$$

and similarly for $F_A(\neg a, C, G)$, $F_{L_1}(\neg a)$ and $F_{L_2}(\neg a)$. 
Performing exact inference: variable elimination

\[ F_A(a, C, G) F_{L1}(a) F_{L2}(a) = \]

\[
\begin{array}{c|ccc}
C & G & \text{Value} \\
\hline
\top & \top & 0.98 \times 0.99 \times 0.6 \\
\top & \bot & 0.96 \times 0.99 \times 0.6 \\
\bot & \top & 0.2 \times 0.99 \times 0.6 \\
\bot & \bot & 0.08 \times 0.99 \times 0.6 \\
\end{array}
\]

\[ F_A(\neg a, C, G) F_{L1}(\neg a) F_{L2}(\neg a) = \]

\[
\begin{array}{c|ccc}
C & G & \text{Value} \\
\hline
\top & \top & 0.02 \times 0.08 \times 0.001 \\
\top & \bot & 0.04 \times 0.08 \times 0.001 \\
\bot & \top & 0.8 \times 0.08 \times 0.001 \\
\bot & \bot & 0.92 \times 0.08 \times 0.001 \\
\end{array}
\]

\[ F_{\neg A, L1, L2}(C, G) = \]

\[
\begin{array}{c|ccc}
C & G & \text{Value} \\
\hline
\top & \top & (0.98 \times 0.99 \times 0.6) + (0.02 \times 0.08 \times 0.001) \\
\top & \bot & (0.96 \times 0.99 \times 0.6) + (0.04 \times 0.08 \times 0.001) \\
\bot & \top & (0.2 \times 0.99 \times 0.6) + (0.8 \times 0.08 \times 0.001) \\
\bot & \bot & (0.08 \times 0.99 \times 0.6) + (0.92 \times 0.08 \times 0.001) \\
\end{array}
\]
Performing exact inference: variable elimination

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

$$\sum_A \Pr (A|\neg c, g) \Pr (l1|A) \Pr (l2|A)$$

$$= \Pr (a|\neg c, g) \Pr (l1|a) \Pr (l2|a) + \Pr (\neg a|\neg c, g) \Pr (l1|\neg a) \Pr (l2|\neg a)$$

$$= (0.2 \times 0.99 \times 0.6) + (0.8 \times 0.08 \times 0.001)$$

which matches!

Continuing in this manner form

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

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

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

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 $S$ and moves from state to state in discrete time steps according to a probabilistic transition

$$\Pr (S \rightarrow S').$$

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

$$\pi_{t+1}(S') = \sum_s \Pr (s \rightarrow S') \pi_t(s).$$

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

$$\forall s' \pi(s') = \sum_s \Pr (s \rightarrow s') \pi(s).$$

There is exactly one stationary distribution for a given $\Pr (S \rightarrow S')$ provided the latter obeys some simple conditions.
The condition of *detailed balance*

\[ \forall s, s' \pi(s) \Pr(s \rightarrow s') = \pi(s') \Pr(s' \rightarrow s) \]

is sufficient to provide a \( \pi \) that is a stationary distribution. To see this simply sum:

\[
\sum_s \pi(s) \Pr(s \rightarrow s') = \sum_s \pi(s') \Pr(s' \rightarrow s) = \pi(s') \sum_s \Pr(s' \rightarrow s) = \pi(s').
\]

If all this is looking a little familiar, it’s because we now have another excellent application for the material in the Part IB course *Data Science*.
Recalling once again the basic equation for performing probabilistic inference

\[
\text{Pr}(Q|o_1, o_2, \ldots, o_m) \propto \sum_{L} \text{Pr}(Q, L, o_1, o_2, \ldots, o_m)
\]

where

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

We are going to consider obtaining samples from the distribution

\[
\text{Pr}(Q, L|o_1, o_2, \ldots, o_m).
\]
The observations are fixed. Let the *state* of our system be a specific set of values for *a query variable and the latent variables*

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

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

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

To move from \( s \) to \( s' \) we replace one of its elements, say \( s_i \), with a new value \( s'_i \) sampled according to

\[ s'_i \sim \Pr (S_i|\overline{S}_i, o_1, \ldots, o_m) \]

This has detailed balance, and has \( \Pr (Q, L|o_1, \ldots, o_m) \) as its stationary distribution.

It is known as *Gibbs sampling*. 
Approximate inference for Bayesian networks

To see that \( \Pr(Q, L|o) \) is the stationary distribution we just demonstrate *detailed balance*:

\[
\pi(s) \Pr(s \rightarrow s') = \Pr(s|o) \Pr(s'|\overline{s_i}, o)
\]

\[
= \Pr(s_i, \overline{s_i}|o) \Pr(s'|\overline{s_i}, o)
\]

\[
= \Pr(s_i|\overline{s_i}, o) \Pr(\overline{s_i}|o) \Pr(s'|\overline{s_i}, o)
\]

\[
= \Pr(s_i|\overline{s_i}, o) \Pr(s'_i, \overline{s_i}|o)
\]

\[
= \Pr(s' \rightarrow s) \pi(s').
\]

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

For example, sampling from \( \Pr(S_i|\overline{s_i}, o) \) 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 $s_1, s_2, \ldots$ sampled according to $Pr(Q, L|o)$.

Finally, note that as

$$Pr(Q|o) = \sum_l Pr(Q, l|o)$$

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

$$q_i \sim Pr(Q|o).$$
Approximate inference for Bayesian networks

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

$$
\mathbb{E}[f(Q)|o] = \sum_q f(q) \Pr(q|o)
$$

$$
= \sum_q f(q) \sum_l \Pr(q,l|o)
$$

$$
= \sum_q \sum_l f(q) \Pr(q,l|o)
$$

so sampling using $\Pr(q,l|o)$ and ignoring the values for $l$ 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.