Machine Learning and Bayesian Inference

Dr Sean Holden

Computer Laboratory, Room FC06

Telephone extension 63725

Email: sbh11@cl.cam.ac.uk

www.cl.cam.ac.uk/∼sbh11/

Part VI

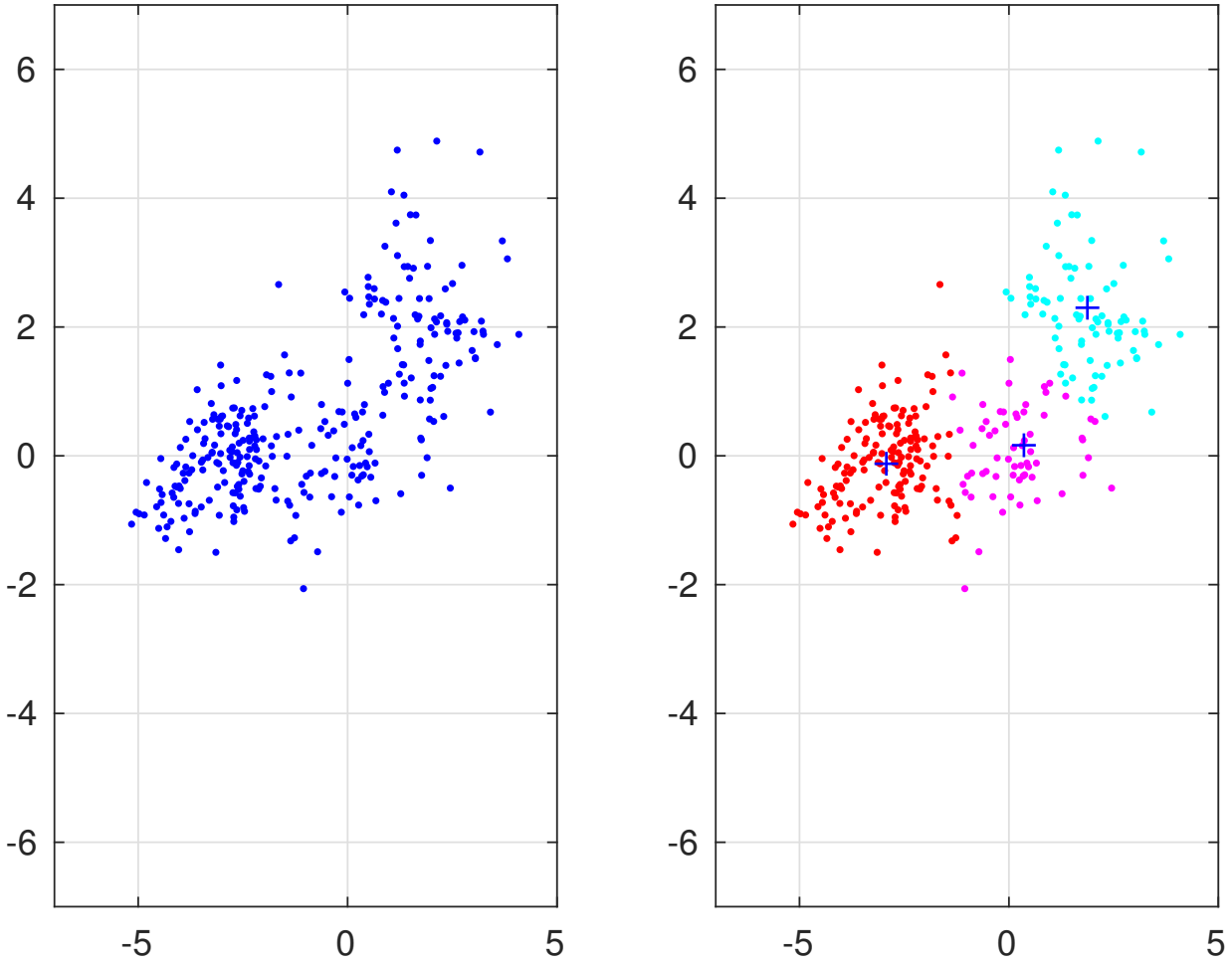
In a nutshell...

Unsupervised learning

Reinforcement learning

Unsupervised learning

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



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

The K-means algorithm

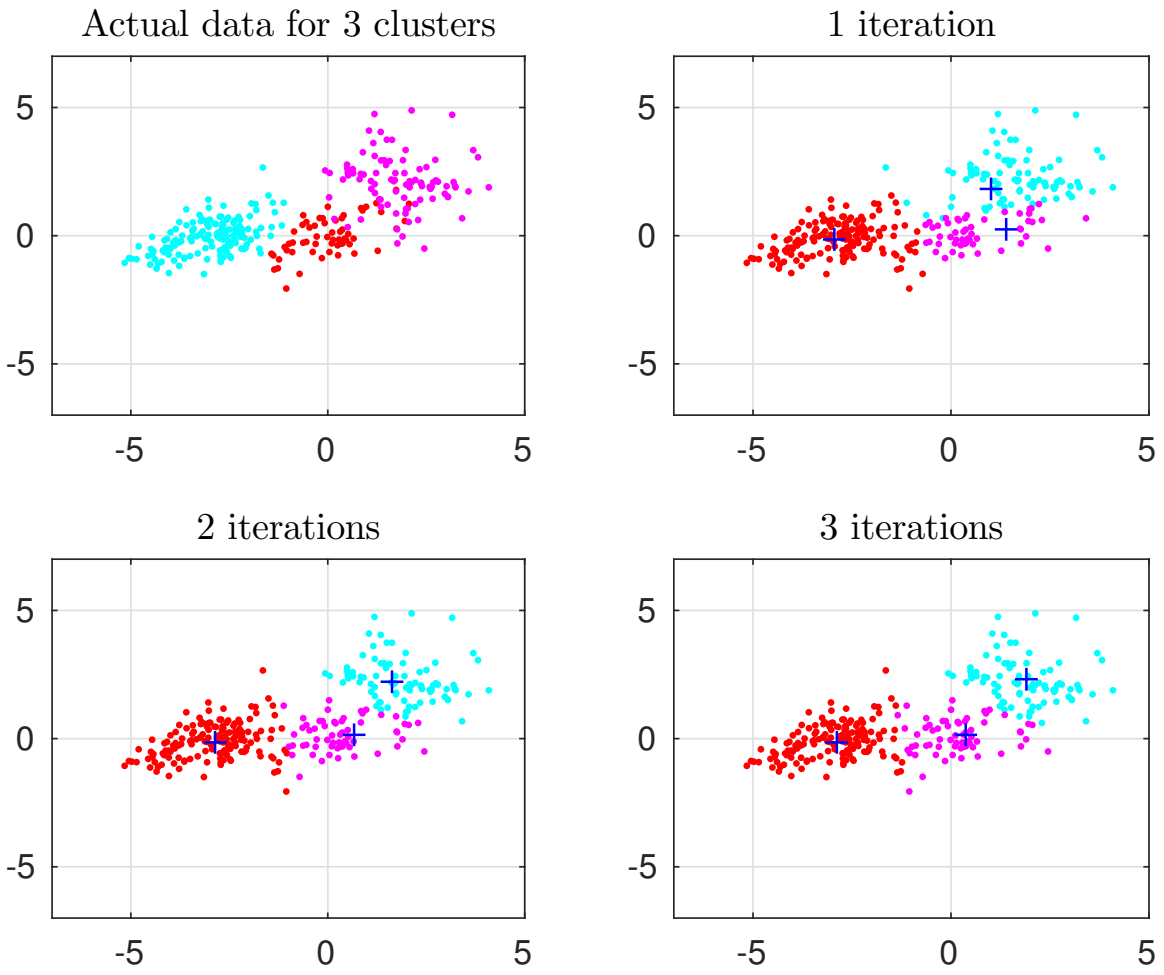
The example on the last slide was obtained using the classical *K-means algorithm*.

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

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

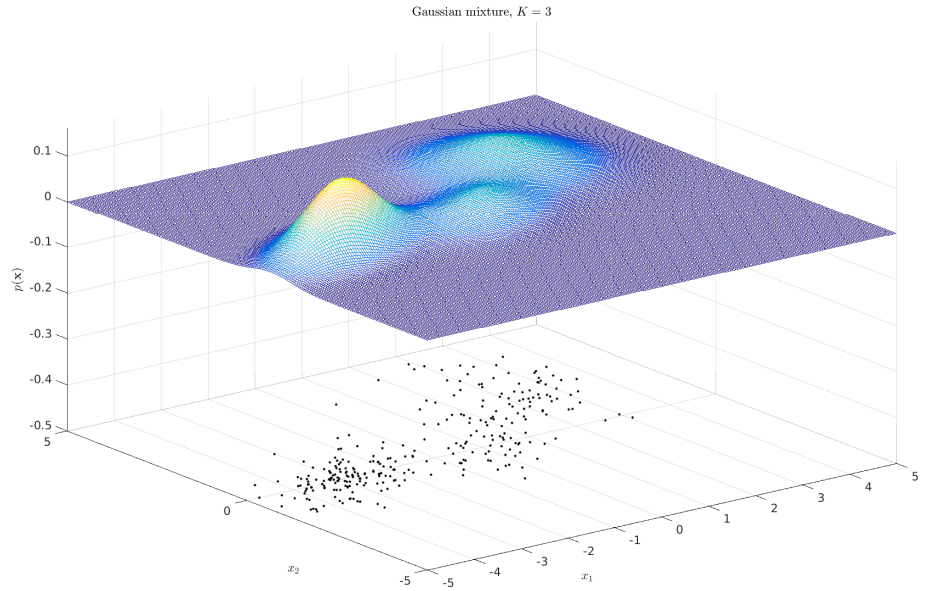
1. Divide $\{\mathbf{x}_i\}$ into K clusters, so *each point is associated with the closest centre*:
$$\mathbf{x}_i \in C_j \iff \forall k \; ||\mathbf{x}_i - \mathbf{c}_j|| \leq ||\mathbf{x}_i - \mathbf{c}_k||.$$
Call these clusters $C_1, \ldots, C_K$.
2. Update the cluster centres to be the *average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

The K-means algorithm



Clustering as maximum-likelihood

The modern approach is once again *probabilistic*.

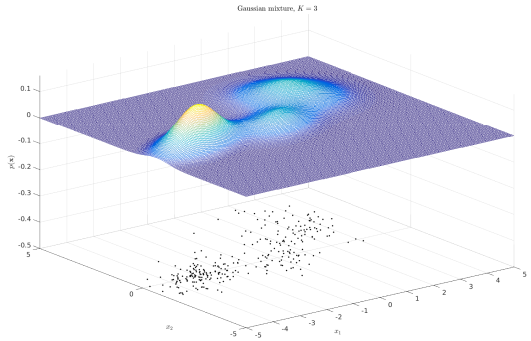


Data from K clusters can be modelled probabilistically as

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

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

Clustering as maximum-likelihood



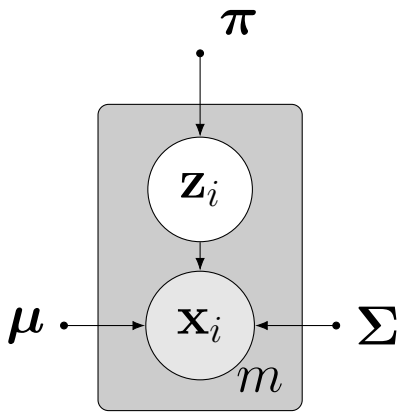
This leads to a log-likelihood for m points of

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

This tends to be *hard to maximise directly* to choose $\boldsymbol{\theta}$. (You can find stationary points but they depend on one-another.)

Clustering as maximum-likelihood

We can however introduce some *latent variables*.



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

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

and

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

Clustering as maximum-likelihood

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

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

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

Why is this?

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

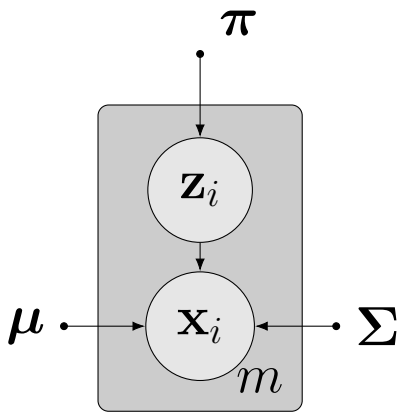
$$p(\mathbf{x}_i|\mathbf{z}_i, \boldsymbol{\theta}) = \prod_{k=1}^{K} \left[p(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right]^{z_i^{(k)}}$$

and similarly

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

Clustering as maximum-likelihood

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



then you can write

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

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

Clustering as maximum-likelihood

Consequently

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

What have we achieved so far?

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

But how do we link them together?

The EM algorithm

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

Here we apply it to unsupervised learning, but it can also be applied to learning *Hidden Markov Models (HMMs)* and many other things

Let $q(\mathbf{Z})$ be *any* distribution on the *latent variables*. Write

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

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

The Kullback-Leibler (KL) distance

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

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

It has the important properties that:

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

The EM algorithm

If we also define

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

then we can re-arrange the last expression to get

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

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

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

The EM algorithm works as follows:

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

The EM algorithm

Let's look at the two steps separately.

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

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

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

so this is easy!

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

So in the E step we just choose

$$q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}_t).$$

The EM algorithm

The *M step* is a little more involved, but we end up with

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

and

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

and *this maximization is tractable*. (Though you will need a *Langrange multiplier*...)

The EM algorithm

The EM algorithm for a mixture model summarized:

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

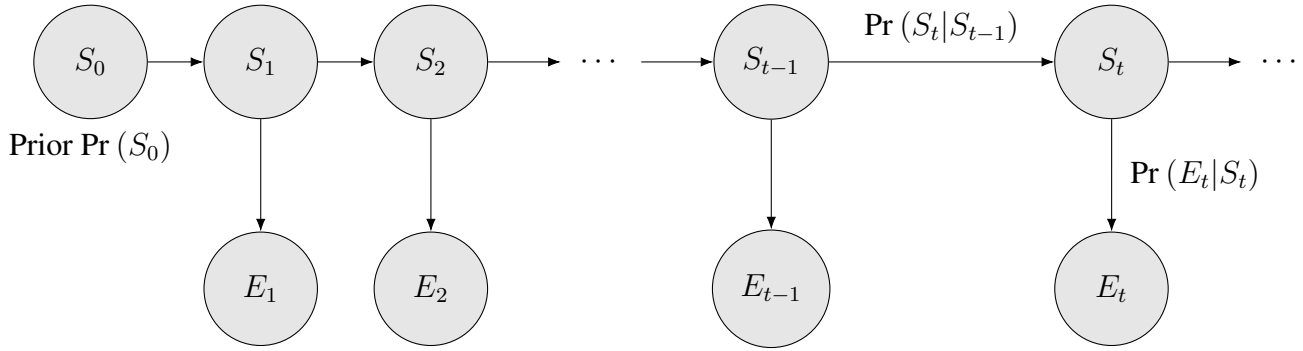
$$L(q, \boldsymbol{\theta}) \leq \log p(\mathbf{X}|\boldsymbol{\theta}).$$

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

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

Reinforcement learning and HMMs

Hidden Markov Models are appropriate when our agent models the world as follows

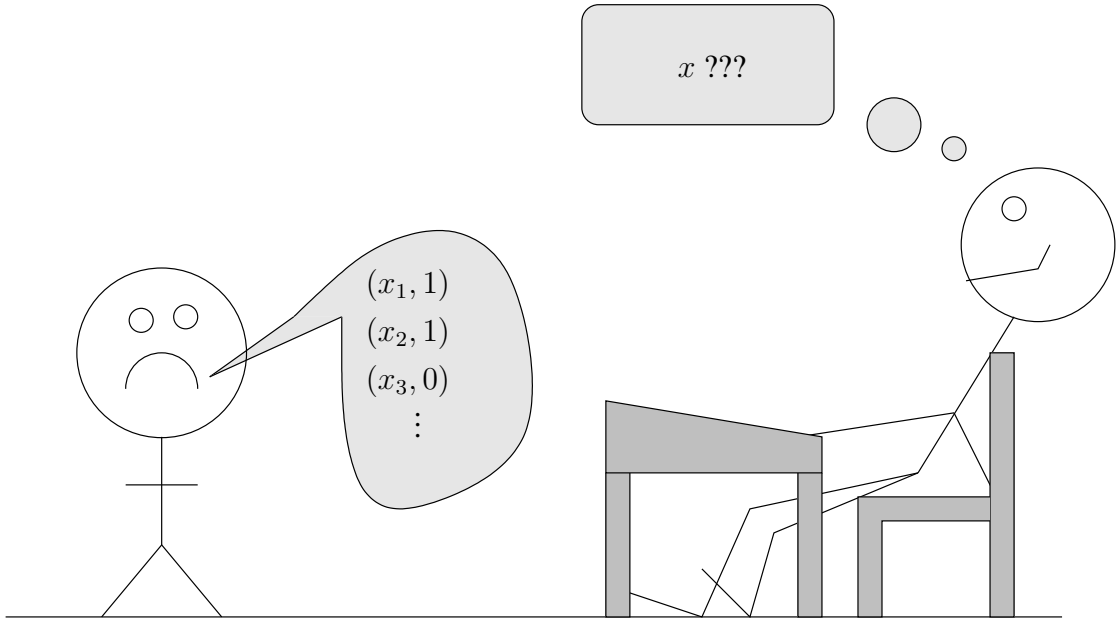


and *only wants* to infer information about the *state* of the world on the basis of observing the available *evidence*.

This might be criticised as un-necessarily restricted, although it is very effective *for the right kind of problem*.

Reinforcement learning and supervised learning

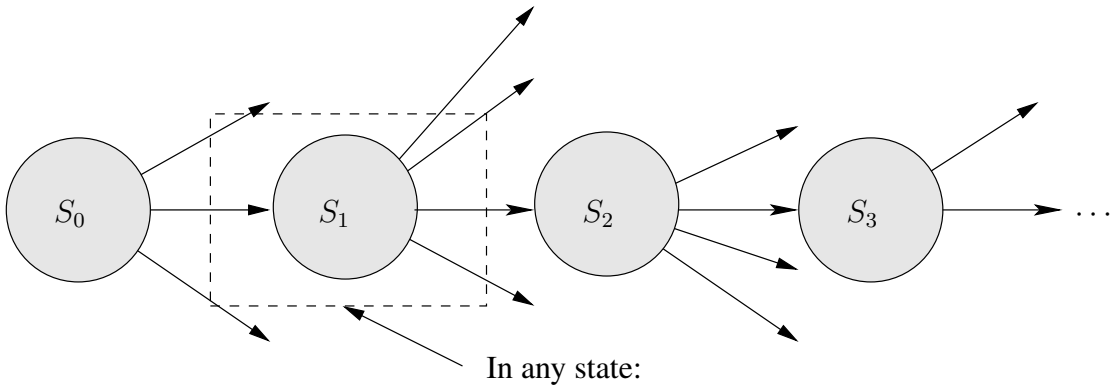
Supervised learners learn from *specifically labelled chunks of information*:



This might also be criticised as un-necessarily restricted: there are other ways to learn.

Reinforcement learning: the basic case

Modelling the world in a *more realistic way*:



The agent can *perform actions* in order to *change the world's state*.

If the agent performs an action in a particular state, then it *gains a corresponding reward*.

Deterministic Markov Decision Processes

Formally, we have a set of states

$$S = \{s_1, s_2, \ldots, s_n\}$$

and in each state we can perform one of a set of actions

$$A = \{a_1, a_2, \ldots, a_m\}.$$

We also have a function

$$\mathcal{S} : S \times A \to S$$

such that $\mathcal{S}(s, a)$ is the new state resulting from performing action a in state s, and a function

$$\mathcal{R} : S \times A \to \mathbb{R}$$

such that $\mathcal{R}(s, a)$ is the *reward* obtained by executing action a in state s.

Deterministic Markov Decision Processes

From the point of view of the agent, there is a matter of considerable importance:

> The agent does not have access to the functions $\mathcal{S}$ and $\mathcal{R}$.

It therefore has to *learn* a *policy*, which is a function

$$p : S \to A$$

such that $p(s)$ provides the action a that should be executed in state s.

What might the agent use as its *criterion for learning a policy*?

Measuring the quality of a policy

Say we start in a state at time t, denoted s_t, and we follow a policy p. At each future step in time we get a reward. Denote the rewards r_t, r_{t+1}, ... and so on.

A common measure of the quality of a policy p is the *discounted cumulative reward*

$$V^p(s_t) = \sum_{i=0}^{\infty} \epsilon^i r_{t+i}$$
$$= r_t + \epsilon r_{t+1} + \epsilon^2 r_{t+2} + \cdots$$

where $0 \leq \epsilon \leq 1$ is a constant, which defines a *trade-off* for how much we value *immediate rewards* against *future rewards*.

The intuition for this measure is that, on the whole, we should like our agent *to prefer rewards gained quickly*.

Two important issues

Note that in this kind of problem we need to address two particularly relevant issues:

- The *temporal credit assignment* problem: that is, how do we decide *which specific actions* are important in *obtaining a reward*?
- The *exploration/exploitation* problem. How do we decide between *exploiting* the knowledge we already have, and *exploring* the environment in order to possibly obtain new (and more useful) knowledge?

We will see later how to deal with these.

The optimal policy

Ultimately, our learner's aim is to learn the *optimal policy*

$$p_{\text{opt}}(s) = \underset{p}{\operatorname{argmax}}\, V^p(s)$$

for some initial state s. Define the optimal discounted cumulative reward $V_{\text{opt}}(s) = V^{p_{\text{opt}}}(s)$. How might we go about *learning the optimal policy*?

The only information we have during learning is the individual rewards obtained from the environment.

We could try to learn $V_{\text{opt}}(s)$ directly, so that states can be compared:

$$\text{Consider } s \text{ as better than } s' \text{ if } V_{\text{opt}}(s) > V_{\text{opt}}(s').$$

However we actually want to compare *actions*, not *states*. Learning $V_{\text{opt}}(s)$ might help as

$$p_{\text{opt}}(s) = \underset{a}{\operatorname{argmax}}\, [\mathcal{R}(s,a) + \epsilon V_{\text{opt}}(\mathcal{S}(s,a))]$$

but *only if we know* $\mathcal{S}$ and $\mathcal{R}$.

As we are interested in the case where these functions are *not* known, we need something slightly different.

The $\mathcal{Q}$ function

The trick is to define the following function:

$$\mathcal{Q}(s,a) = \mathcal{R}(s,a) + \epsilon V_{\text{opt}}(\mathcal{S}(s,a)).$$

This function specifies the discounted cumulative reward obtained if you do action a in state s *and then follow the optimal policy*.

As

$$p_{\text{opt}}(s) = \underset{a}{\operatorname{argmax}}\, \mathcal{Q}(s,a)$$

then provided one can learn $\mathcal{Q}$ *it is not necessary to have knowledge of* $\mathcal{S}$ *and* $\mathcal{R}$ *to obtain the optimal policy*.

The $\mathcal{Q}$ function

Note also that

$$V_{\text{opt}}(s) = \max_{\alpha} \mathcal{Q}(s, \alpha)$$

and so

$$\mathcal{Q}(s, a) = \mathcal{R}(s, a) + \epsilon \max_{\alpha} \mathcal{Q}(\mathcal{S}(s, a), \alpha)$$

which suggests a simple learning algorithm.

Let Q' be *our learner's estimate* of what the exact $\mathcal{Q}$ function is.

That is, in the current scenario Q' is *a table containing the estimated values of* $\mathcal{Q}(s, a)$ for all pairs (s, a).

$\mathcal{Q}$-learning

Start with all entries in Q' set to 0. (In fact random entries will do.)

Repeat the following:

1. Look at the current state s and choose an action a. (We will see how to do this in a moment.)
2. Do the action a and obtain some reward $\mathcal{R}(s, a)$.
3. Observe the new state $\mathcal{S}(s, a)$.
4. Perform the update

$$Q'(s, a) = \mathcal{R}(s, a) + \epsilon \max_{\alpha} Q'(\mathcal{S}(s, a), \alpha).$$

Note that this can be done in *episodes*. For example, in learning to play games, we can play multiple games, each being a single episode.

The procedure *converges under some simple conditions*.

Choosing actions to perform

We have not yet answered the question of how to *choose actions* to perform during learning.

One approach is to choose actions *based on our current estimate* Q'. For instance

$$\text{action chosen in current state } s = \operatorname*{argmax}_a Q'(s, a).$$

However we have already noted the *trade-off between exploration and exploitation*. It makes more sense to:

- *Explore* during the early stages of training.
- *Exploit* during the later stages of training.

(This also turns out to be sensible to guarantee convergence.)

Choosing actions to perform

One way in which to choose actions that incorporates these requirements is to introduce a constant λ and choose actions *probabilistically* according to

$$\Pr\left(\text{action } a \middle| \text{state } s\right) = \frac{\lambda^{Q'(s,a)}}{\sum_a \lambda^{Q'(s,a)}}.$$

Note that:

- If λ is *small* this promotes *exploration*.
- If λ is *large* this promotes *exploitation*.

We can vary λ as training progresses.

There are two further simple ways in which the process can be improved:

1. If training is episodic, we can store the rewards obtained during an episode and update *backwards* at the end.

   This allows better updating at the expense of requiring more memory.
2. We can remember information about rewards and occasionally *re-use* it by re-training.

Nondeterministic MDPs

The $\mathcal{Q}$-learning algorithm generalises easily to a more realistic situation, where the outcomes of actions are *probabilistic*.

Instead of the functions $\mathcal{S}$ and $\mathcal{R}$ we have *probability distributions*

$$\Pr\left(\text{new state}\middle|\text{current state, action}\right)$$

and

$$\Pr\left(\text{reward}\middle|\text{current state, action}\right).$$

and we now use $\mathcal{S}(s,a)$ and $\mathcal{R}(s,a)$ to denote the corresponding random variables.

We now have

$$V^p = \mathbb{E}\left(\sum_{i=0}^{\infty} \epsilon^i r_{t+i}\right)$$

and the best policy p_{opt} maximises V^p.

$\mathcal{Q}$-learning for nondeterministic MDPs

We now have

$$\begin{aligned}\mathcal{Q}(s,a) &= \mathbb{E}(\mathcal{R}(s,a)) + \epsilon \sum_{\sigma} \Pr(\sigma|s,a)\, V^{\text{opt}}(\sigma) \\ &= \mathbb{E}(\mathcal{R}(s,a)) + \epsilon \sum_{\sigma} \Pr(\sigma|s,a) \max_{\alpha} \mathcal{Q}(\sigma,\alpha)\end{aligned}$$

and the rule for learning becomes

$$Q'_{n+1} = (1-\theta_{n+1})Q'_n(s,a) + \theta_{n+1}\left[\mathcal{R}(s,a) + \max_{\alpha} Q'_n(\mathcal{S}(s,a),\alpha)\right]$$

with

$$\theta_{n+1} = \frac{1}{1+v_{n+1}(s,a)}$$

where $v_{n+1}(s,a)$ is the number of times the pair s and a has been visited so far.

Alternative representation for the Q' table

But there's always a catch...

We have to store the table for Q':

- Even for quite straightforward problems it is HUGE!!! - certainly big enough that it can't be stored.
- A standard approach to this problem is, for example, to represent it as a *neural network*.
- One way might be to make s and a the inputs to the network and train it to produce $Q'(s, a)$ as its output.

This, of course, introduces its own problems, although it has been used very successfully in practice.