UNIT V GRAPHICAL MODELS

Markov Chain Monte Carlo Methods – Sampling – Proposal Distribution – Markov Chain Monte Carlo – Graphical Models – Bayesian Networks – Markov Random Fields – Hidden Markov Models – Tracking Methods.

Markov Chain Monte Carlo Methods

Markov Chain Monte Carlo sampling provides a class of algorithms for systematic random sampling from high-dimensional probability distributions. Unlike Monte Carlo sampling methods that are able to draw independent samples from the distribution, Markov Chain Monte Carlo methods draw samples where the next sample is dependent on the existing sample, called a Markov Chain. This allows the algorithms to narrow in on the quantity that is being approximated from the distribution, even with a large number of random variables.

What Is Markov Chain Monte Carlo?

The solution to sampling probability distributions in high-dimensions is to use Markov Chain Monte Carlo, or MCMC for short.

The most popular method for sampling from high-dimensional distributions is Markov chain Monte Carlo or MCMC.

Monte Carlo

Monte Carlo is a technique for randomly sampling a probability distribution and approximating a desired quantity.

Markov Chain

Markov chain is a systematic method for generating a sequence of random variables where the current value is probabilistically dependent on the value of the prior variable. Specifically, selecting the next variable is only dependent upon the last variable in the chain.

Example

Consider a board game that involves rolling dice, such as snakes and ladders (or chutes and ladders). The roll of a die has a uniform probability distribution across 6 stages (integers 1 to 6). You have a position on the board, but your next position on the board is only based on the current position and the random roll of the dice. Your specific positions on the board form a Markov chain.

Another example of a Markov chain is a random walk in one dimension, where the possible moves are 1, -1, chosen with equal probability, and the next point on the number line in the walk is only dependent upon the current position and the randomly chosen move.

Combining these two methods, Markov Chain and Monte Carlo, allows random sampling of high-dimensional probability distributions that honors the probabilistic dependence between samples by constructing a Markov Chain that comprise the Monte Carlo sample.

SAMPLING

Markov chain Monte Carlo (**MCMC**) methods comprise a class of algorithms for sampling from a probability distribution. We have produced samples from probability distributions in almost all of the algorithms.

For example, for initialisation of weights. In many cases, the probability distribution we have used has been the uniform one on [0, 1), and we have done it using the np.random.rand() function in NumPy, although we have also seen sampling from Gaussian distributions using np.random.normal().

Random Numbers

- The basis of all of these sampling methods is in the generation of random numbers, and this is something that computers are not really capable of doing.
- There are plenty of algorithms that produce pseudo-random numbers, the simplest of which is the linear congruential generator. This is a very simple function that is defined by a recurrence relation (i.e put one number in to get the second number, and then feed that back in to get the third, and then repeat the cycle):

$$x_{n+1} = (ax_n + c) \mod m,$$

  where a, c, and m are parameters that have to be chosen carefully.
- The initial input x0 (which is known as the seed), are integers, and so are all of the outputs. The modulus function means that the largest number that can be produced is m, and so there are at most m numbers that can be produced by the algorithm.
- Once one number appears a second time, the whole pattern will repeat again since the equation only uses the current output as input. The length of the sequence between repeats is the period, and it should obviously be as long as possible, since it is the most obvious non-randomness in the algorithm.
- The industry-standard algorithm for generating random samples is the Mersenne Twister, which is based on Mersenne prime numbers. It is the random number generator used in NumPy.

Gaussian Random Numbers

- The Mersenne twister produces uniform random numbers. However, often we might want to produce samples from other distributions, e.g., Gaussian.
- The usual method of doing this is the Box–Muller scheme, which uses a pair of uniformly randomly distributed numbers in order to make two independent Gaussian-distributed numbers with zero mean and unit variance.
- Suppose that we had two independent zero mean, unit variance normals. Then their product is:

$$f(x,y) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \frac{1}{\sqrt{2\pi}} e^{-y^2/2} = \frac{1}{2\pi} e^{-(x^2+y^2)/2}.$$

If we use polar coordinates instead (so $x = r \sin(\theta)$ and $y = r \cos(\theta)$) then we would have $r^2 = x^2 + y^2$ and $\theta = \tan^{-1}(y/x)$. Both of these are uniformly distributed random variables ($0 \leq r \leq 1$ and $0 \leq \theta < 2\pi$). In other words, $\theta = 2\pi U_1$ where U_1 is a uniformly distributed random variable. Now we just need a similar expression for r.

We can write that:

$$P(r \leq R) = \int_{r'=0}^{R} \int_{\theta=0}^{2\pi} \frac{1}{2\pi} e^{-r'^2} r' dr' d\theta = \int_{r'=0}^{r} e^{-r'^2} r' dr'.$$

If we use the change of variables $\frac{1}{2} r' 2 = s$ (so that $r' dr' = ds$) then:

$$P(r \leq R) = \int_{s=0}^{r^2/2} e^{-} s ds = 1 - e^{-r^2/2}.$$

So to sample r we just need to solve $1 - e^{(-r^2/2)} = 1 - U_2$ where U_2 is another uniformly distributed random variable, and which has solution $r = \sqrt{-2\ln(U_2)}$. So one algorithm to generate the Gaussian variables is:

The Box–Muller Scheme

- Pick two uniformly distributed random numbers $0 \leq U_1, U_2 \leq 1$
- Set $\theta = 2\pi U_1$ and $r = \sqrt{(} - 2\ln(U_2))$
- Then $x = r\cos(\theta)$ and $y = r\sin(\theta)$ are independent Gaussian-distributed variables with zero mean and unit variance

An alternative approach to computing these random variables is to pick the two uniform random values and scale them to lie between -1 and 1, and to interpret them as describing a point in the plane. If this point is outside the unit circle (so if the variables are U_1 and U_2 as above then if $w^2 = U_2^1 + U_2^2 > 1$) then it is discarded, and another point picked until it is within the circle. Then the transformation $x = U_1\left(\frac{-2lnw^2}{w^2}\right)^{1/2}$ and similarly for y with U_2 also provides the variables.

There is a more efficient algorithm for computing Gaussian-distributed random numbers known as the Ziggurat algorithm.

MONTE CARLO OR BUST

The Monte Carlo principle states that if you take independent and identically distributed (i.e., well-behaved) samples $x^{(i)}$ from an unknown high-dimensional distribution p(x), then as the number of samples gets larger the sample distribution will converge to the true distribution.

Written mathematically, this says:

$$\begin{aligned} p_N(\mathbf{x}) &= \frac{1}{N}\sum_{i=1}^{N}\delta(\mathbf{x}^(i) = \mathbf{x}) \\ &\rightarrow \lim_{N\rightarrow\infty} p_N(\mathbf{x}) = p(\mathbf{x}), \end{aligned}$$

where $\delta(x_i = x)$ is the Dirac delta function that is 0 everywhere except at the point x_i and has $\int \delta x dx = 1$. This can be used to compute the expectation as well (where f(x) is some function and x has discrete values, and the superscript $\cdot^{(i)}$ represents the index of the sample):

$$\begin{aligned} E_N(f) &= \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}^{(i)}) \\ &\rightarrow \lim_{N\rightarrow\infty} E_N(f) = \sum_{\mathbf{x}} f(\mathbf{x})p(\mathbf{x}). \end{aligned}$$

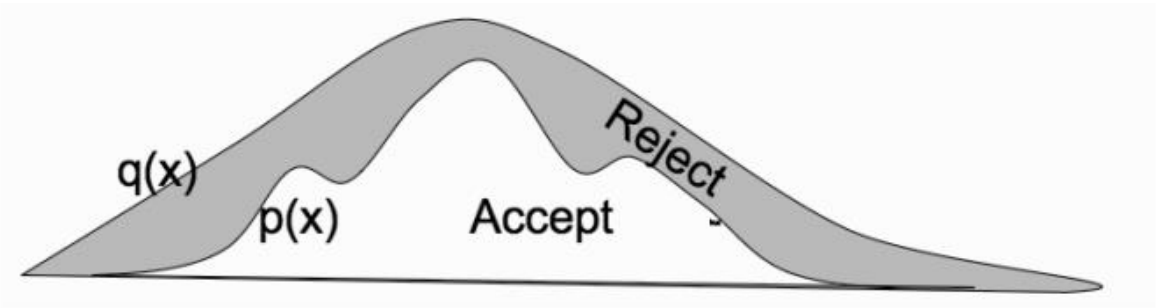
Proposal Distribution

Assume that we can evaluate some related distribution $\tilde{p}(x)$ for a given x, where:

$$p(\mathbf{x}) = \frac{1}{Z_p}\tilde{p}(\mathbf{x}),$$

where Zp is some normalisation constant.

We make the decision of whether or not to accept the sample by picking a uniformly distributed random number u between 0 and $M_q(x^*)$. If this random number is less than $\tilde{p}(x^*)$, then we accept x^*, otherwise we reject it. The reason why this works is known as the envelope principle: the pair (x^*, u) is uniformly distributed under $M_q(x^*)$, and the rejection part throws away samples that don't match the uniform distribution on $p(x^*)$, so $M_q(x)$ forms an envelope on p(x).



we sample from $M_q(x)$ and reject any sample that lies in the grey area. The smaller M is, the more samples we get to keep, but we need to ensure that $\tilde{p}(x) \leq M_q(x)$. This method is known as rejection sampling, and the algorithm can be written as:

The Rejection Sampling Algorithm

- Sample $\mathbf{x}^*$ from $q(\mathbf{x})$ (e.g., using the Box–Muller scheme if $q(\mathbf{x})$ is Gaussian)
- Sample u from uniform$(0, \mathbf{x}^*)$
- If $u < p(\mathbf{x}^*)/Mq(\mathbf{x}^*)$:
  - add $\mathbf{x}^*$ to the set of samples
- Else:
  - reject $\mathbf{x}$ and pick another sample

Suppose that we want to compute the expectation of a function f(x) for a continuous random variable x distributed according to unknown distribution p(x). Starting from the expression of the expectation that we wrote out earlier, we can introduce another distribution q(x):

$$\begin{aligned} E(f) &= \int p(\mathbf{x})\, f(x)\, d\mathbf{x} \\ &= \int p(\mathbf{x}) f(\mathbf{x}) \frac{q(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x} \\ &\approx \frac{1}{N} \sum_{i=1}^{N} \frac{p\left(\mathbf{x}^{(i)}\right)}{q\left(\mathbf{x}^{(i)}\right)} f\left(x^{(i)}\right), \end{aligned}$$

where we have used the fact that q(x) is the density of a random variable, and so if we perform $\int qxdx$ over all values of x, then it must equal 1. The ratio $w(x^{(i)}) = p(x^{(i)})/q(x^{(i)})$ is called the importance weight, and it corrects for sampling from the grey region without having to reject samples. While this can be used to estimate the expectation directly, the real benefit of computing the importance weights is that they can be used in order to resample the data. This leads to an algorithm known descriptively as Sampling Importance-Resampling. In the words of the advert, it 'does exactly what it says on the tin':

The Sampling-Importance-Resampling Algorithm

- Produce N samples $\mathbf{x}^{(i)}$, $i = 1 \ldots N$ from $q(\mathbf{x})$
- Compute normalised importance weights

$$w^{(i)} = \frac{p(\mathbf{x}^{(i)})/q(\mathbf{x}^{(i)})}{\sum_j p(\mathbf{x}^{(j)})/q(\mathbf{x}^{(j)})}$$

- Resample from the distribution $\{\mathbf{x}^{(i)}\}$ with probabilities given by the weights $w^{(i)}$

MARKOV CHAIN MONTE CARLO

Markov Chains

- A Markov chain is a chain with the Markov property, i.e., the probability at time t depends only on the state at t – 1.
- The set of possible states are linked together by transition probabilities that say how likely it is that you move from the current state to each of the others, and they are

generally written as a matrix T. They might be constant, or functions of some other variables, but here we will assume that they are constant.

- Given a chain, we can perform a random walk on the chain by choosing a start state and randomly choosing each successive state according to the transition probabilities.
- The link to sampling that we need is that if the transition probabilities reflect the distribution that we wish to sample from, then a random walk will explore that distribution.
- One problem with this is that random walks are very inefficient at exploring space, since they move back towards the start as often as they move away, which means the distance they move from the start scales as $\sqrt{t}$, where t is the number of samples. We therefore want to explore more efficiently than just using a random walk.

We also want the distribution $p(\mathbf{x})$ to be invariant to the Markov chain, which means that the transition probabilities don't change the distribution:

$$p(\mathbf{x}) = \sum_{\mathbf{y}} T(\mathbf{y}, \mathbf{x}) p(\mathbf{y}).$$

Finding the transition probabilities to make this true requires that we can move backwards and forwards along the chain with equal probability, so that the chain is reversible This says that the probability of being in an unlikely state s (sampling datapoint $\mathbf{x}$), but heading for a likely state s' (datapoint $\mathbf{x}'$) should be the same as being in the likely state s' and heading for the unlikely state s, so that:

$$p(\mathbf{x}) T(\mathbf{x}, \mathbf{x}') = p(\mathbf{x}') T(\mathbf{x}', \mathbf{x}).$$

This is known as the detailed balance condition and the fact that it leaves the distribution $p(\mathbf{x})$ alone is fairly obvious with a little calculation. If the chain satisfies the detailed balance condition, then it must be ergodic, since $\sum_{\mathbf{y}} T(\mathbf{x}, \mathbf{y}) = 1$, since you must have come from some state, and so:

$$\sum_{\mathbf{y}} p(\mathbf{y}) T(\mathbf{y}, \mathbf{x}) = p(\mathbf{x}),$$

which means that $p(\mathbf{x})$ must be an invariant distribution of T. So if we can work out how to construct a Markov chain with detailed balance we can sample from it in order to sample from our distribution. This is known as Markov Chain Monte Carlo (MCMC) sampling, and the most popular algorithm that is used for MCMC is the Metropolis–Hastings algorithm after the two people who were directly involved in its creation.

The Metropolis–Hastings Algorithm

The idea of Metropolis–Hastings is similar to that of rejection sampling: we take a sample x^* and choose whether or not to keep it. Except, unlike rejection sampling, rather than picking another sample if we reject the current one, instead we add another copy of the previous accepted sample. Here, the probability of keeping the sample is $u(x^*|x^{(i-1)})$:

$$u(\mathbf{x}^*|\mathbf{x}^{(i)}) = \min\left(1, \frac{\tilde{p}(\mathbf{x}^*)q(\mathbf{x}^{(i)}|\mathbf{x}^*)}{\tilde{p}(\mathbf{x}^{(i)})q(\mathbf{x}^*|\mathbf{x}^{(i)})}\right).$$

The Metropolis–Hastings Algorithm

- Given an initial value x_0
- Repeat
  - sample $\mathbf{x}^*$ from $q(\mathbf{x}_i|\mathbf{x}_{i-1})$
  - sample u from the uniform distribution
  - if $u <$ Equation (15.14):
    - set $\mathbf{x}[i+1] = \mathbf{x}^*$
  - otherwise:
    - set $\mathbf{x}[i+1] = \mathbf{x}[i]$
- Until you have enough samples

Gibbs Sampling

Gibbs sampling or a **Gibbs sampler** is a Markov chain Monte Carlo (MCMC) algorithm for obtaining a sequence of observations which are approximated from a specified multivariate probability distribution, when direct sampling is difficult. This sequence can be used to approximate the joint distribution (e.g., to generate a histogram of the distribution); to approximate the marginal distribution of one of the variables, or some subset of the variables (for example, the unknown parameters or latent variables); or to compute an integral (such as the expected value of one of the variables). Typically, some of the variables correspond to observations whose values are known, and hence do not need to be sampled.

A set of probabilities from a network that looks like:

$$p(\mathbf{x}) = \prod_j p(x_j|x_{\alpha j}),$$

where $x_{\alpha j}$ is the parents of x_j.

Given that we know $p(x_j|x_{\alpha j}) \prod_{k\in\beta(j)} p(x_k|x_{\alpha(k)})$ (which is $p(x_j|x_{-j})$), maybe we should try using it as the proposal distribution, giving:

$$q(x^*|x^{(i)}) = \begin{cases} p\left(x^*_j, x^{(i)}_j\right) & \text{if } x^*_j = x^{(i)}_j \\ 0 & \text{otherwise.} \end{cases}$$

If we then use Metropolis–Hastings, we find that the acceptance probability P_a is:

$$P_a = \min\left\{1, \frac{p(x^*)p(x^{(i)}_j|x^{(i)}_{-j})}{p(x^{(i)})p(x^*_j|x^*_{-j})}\right\},$$

and looking carefully at this and expanding out the conditional probabilities we get:

$$P_a = \min\left\{1, \frac{p(x^*)p(x^{(i)}_j, x^{(i)}_{-j})p(x^{(i)}_{-j})}{p(x^{(i)})p(x^*_j, x^*_{-j})p(x^*_{-j})}\right\}.$$

Since $p(x^*_j, x^*_{-j}) = p(x^*)$, and similarily for $p^{(i)}$, we only have to worry about $\frac{p(x^{(i)})}{p(x^*_{-j})}$. From the definition of the proposal distribution we know that $x*_{-j} = x^{(i)}_{-j}$, and so the computation is actually $\min 1, 1 = 1$. So we always accept the proposal, which makes things much simpler.

The total algorithm is given by choosing each variable and sampling from its conditional distribution.

The total algorithm is given by choosing each variable and sampling from its conditional distribution.

The Gibbs Sampler

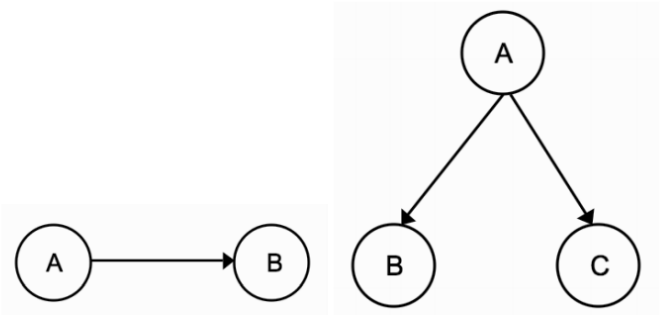
- For each variable x_j:
  - initialise $x^{(0)}_j$
- Repeat
  - for each variable x_j:
    - sample $x^{(i+1)}_1$ from $p(x_1|x^{(i)}_2, \ldots x^{(i)}_n)$
    - sample $x^{(i+1)}_2$ from $p(x_2|x^{(i+1)}_1, x^{(i)}_3, \ldots x^{(i)}_n)$

    * ...
    * sample $x_n^{(i+1)}$ from $p(x_n|x_1^{(i+1)}, \ldots x_{n-1}^{(i+1)})$
- Until you have enough samples

Graphical Models

The graphs used in graphical models are the exact ones that are taught in basic algorithms classes: a set of nodes, together with links between them, which can be either directed (i.e., have arrows on them so that you can only go one way along them) or not.

There are two basic types of graphical models, depending upon whether or not the edges are directed.



BAYESIAN NETWORKS

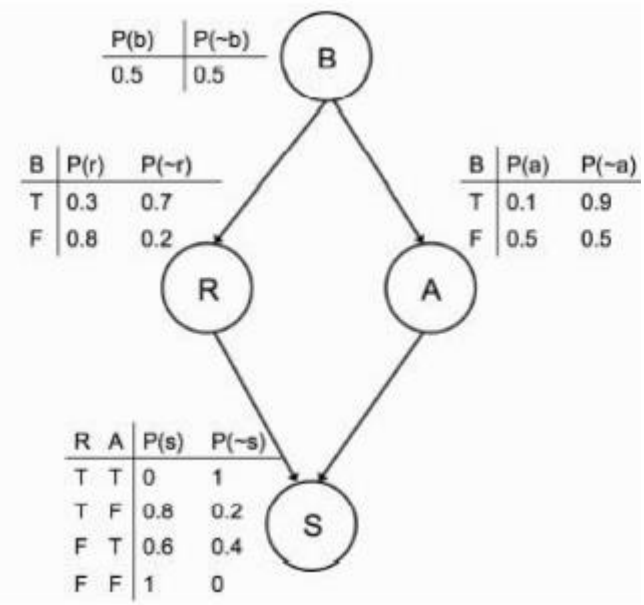
Bayesian networks are a widely-used class of probabilistic graphical models. They consist of two parts: a structure and parameters. The structure is a directed acyclic graph (DAG) that expresses conditional independencies and dependencies among random variables associated with nodes.

The parameters consist of conditional probability distributions associated with each node. A Bayesian network is a compact, flexible and interpretable representation of a joint probability distribution.

It is also an useful tool in knowledge discovery as directed acyclic graphs allow representing causal relations between variables. Typically, a Bayesian network is learned from data.

Example

The sample graphical model. ‘B’ denotes a node stating whether the exam was boring, ‘R’ whether or not you revised, ‘A’ whether or not you attended lectures, and ‘S’ whether or not you will be scared before the exam.



It table shows that whether or not you will be scared before an exam based on whether or not the course was boring ('B'), which was the key factor you used to decide whether or not to attend lectures ('A') and revise ('R'). We can use it to perform inference in order to decide the likelihood of you being scared before the exam ('S').

There are two kinds of inferences, depending on whether the observations that are made come from the top of the graph or the bottom.

If a set of observations is used to predict an unknown outcome, then we are doing top-down inference or prediction, whereas if the outcome is known, but the causes are hidden, then we are doing bottom-up inference or diagnosis.

In order to compute the probability of being scared, we need to compute P(b, r, a, s), where the lower-case letters indicate particular values that the upper-case variables can take.

In the graphical model we can read the conditional probabilities from the graph—if there is no direct link, then variables are conditionally independent given a node that is already included, so those variables are not needed.

$$\begin{aligned} P(s) &= \sum_{b,r,a} P(b,r,a,s) \\ &= \sum_{b,r,a} P(b) \times P(r|b) \times P(a|b) \times P(s|r,a) \\ &= \sum_{b} P(b) \times \sum_{r,a} P(r|b) \times P(a|b) \times P(s|r,a). \end{aligned}$$

$$\begin{aligned} P(s) &= 0.3 \times 0.1 \times 0 + 0.3 \times 0.9 \times 0.8 + 0.7 \times 0.1 \times 0.6 + 0.7 \times 0.9 \times 1 \\ &= 0.328. \end{aligned}$$

$$
\begin{aligned}
P(r|s) &= \frac{P(s|r)P(r)}{P(s)} \\
&= \frac{\sum_{b,a} P(b,a,r,s)}{P(s)} \\
&= \frac{0.5 \cdot (0.3 \cdot 0.1 \cdot 0 + 0.3 \cdot 0.9 \cdot 0.8) + 0.5 \cdot (0.8 \cdot 0.5 \cdot 0 + 0.8 \cdot 0.5 \cdot 0.8)}{P(s)} \\
&= \frac{0.268}{0.684} = 0.3918. \\
P(a|s) &= \frac{P(s|a)P(a)}{P(s)} \\
&= \frac{0.144}{0.684} = 0.2105.
\end{aligned}
$$

This use of Bayes' rule is the reason why this type of graphical model is known as a Bayesian network.

The Variable Elimination Algorithm

- Create the λ tables:
  - for each variable v:
    - make a new table
    - for all possible true assignments x of the parent variables:
      - add rows for $P(v|x)$ and $1 - P(v|x)$ to the table
    - add this table to the set of tables

- Eliminate known variables v:
  - for each table:
    - remove rows where v is incorrect
    - remove column for v from table
- Eliminate other variables (where x is the variable to keep):
  - for each variable v to be eliminated:
    - create a new table t'
    - for each table t containing v:
      - $v_{\text{true},t} = v_{\text{true},t} \times P(v|x)$
      - $v_{\text{false},t} = v_{\text{false},t} \times P(\neg v|x)$
    - $v_{\text{true},t'} = \sum_t (v_{\text{true},t})$
    - $v_{\text{false},t'} = \sum_t (v_{\text{false},t})$
  - replace tables t with the new one t'
- Calculate conditional probability:
  - for each table:
    - $x_{\text{true}} = x_{\text{true}} \times P(x)$
    - $x_{\text{false}} = x_{\text{false}} \times P(\neg x)$
    - probability is $x_{\text{true}}/(x_{\text{true}} + x_{\text{false}})$

Approximate Inference

There are two other methods of doing approximate inference -loopy belief propagation and mean field approximation.

The basic idea of using MCMC methods in Bayesian networks is to sample from the hidden variables, and then (depending upon the MCMC algorithm employed) weight the samples by their likelihoods. Creating the samples is very easy: for prediction, we start at the top of the graph and sample from each of the known probability distributions.

In this sampling method, we have to work through the graph from top to bottom and select rows from the conditional probability table that match the previous case. This is not what we would do if we were constructing the table by hand. Suppose that you wanted to know how many courses you did not attend the lectures for because the course was boring. You would simply look back through your courses and count the number of boring courses where you

didn’t go to lectures, ignoring all the interesting courses. We can use exactly this idea if we use rejection sampling.

The method samples from the unconditional distribution and simply rejects any samples that don’t have the correct prior probability. It means that we can sample from each distribution independently, and then throw away any samples that don’t match the other variables. This is obviously computationally easier, but we might have to reject a lot of samples.

The solution to this problem is to work out what evidence we already have and use this evidence to assign likelihoods to the other variables that are sampled.

Gibbs sampling will find us the maxima of our probability distribution given enough samples.

The probabilities in the network are:

$$p(x) = \prod_j p(x_j|x_{\alpha j}),$$

where $x_{\alpha j}$ are the parent nodes of x_j . In a Bayesian network, any given variable is independent of any node that is not their child,

$$p(x_j|x_{-j}) = p(x_j|x_{\alpha j}) \prod_{k\in\beta(j)} p(x_k|x_{\alpha(k)}),$$

where $\beta(j)$ is the set of children of node x_j and x_{-j} signifies all values of x_i except x_j . For any node we only need to consider its parents, its children, and the other parents of the children, as shown in Figure. This set is known as the Markov blanket of a node.

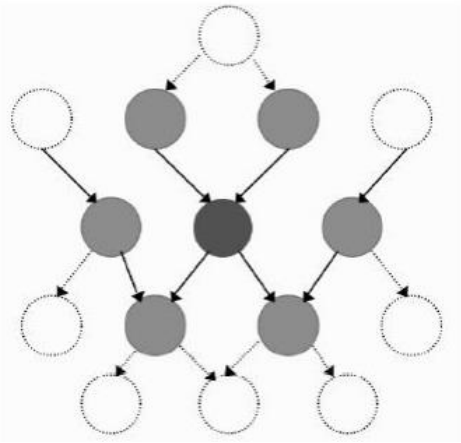


Figure: The Markov blanket of a node is the set of nodes (shaded light grey) that are either parents or children of the node, or other parents of its children (shaded dark grey).

Making Bayesian Networks

If the structure and conditional probability tables are given for the Bayesian network, then we can perform inference on it by using Gibbs sampling or, if the network is simple enough, exactly. However, this raises the important question about where the Bayesian network itself comes from. Constructing Bayesian networks by hand is obviously very boring to do, and unless it is based on real data, then it is subjective.

So why is it so difficult to construct Bayesian networks?

First, we have already seen that the problem of exact inference on Bayesian networks was NP-hard, which is why we had to use approximate inference. Now let's think about the structure of the graph a little. If there are N nodes (i.e., N random variables in the graph), then how many different graphs are there?

For just three nodes ('A', 'B', 'C') we can leave the three unconnected, connect 'A' to 'B' and leave 'C' alone, connect 'B' to 'A' and leave 'C' alone (remember that the links are directional) and lots of variations of that, so that there are seven possible graphs before we have even connected all three nodes to each other.

For ten nodes there are $O(10^{18})$ possible graphs, so we are not going to be searching over all of them. Further, we might want our algorithm to be able to include latent variables, i.e., hidden nodes, which might be a sensible thing to do in terms of explaining the data, but it does make the problem of search even worse.

The idea is to choose the probability distributions to maximise the likelihood of the training data. If there are no hidden nodes, then it is possible to compute the likelihood directly:

$$\begin{aligned} L &= \frac{1}{M} \log \prod_{m=1}^{N} P(D_m|G) \\ &= \frac{1}{M} \sum_{n=1}^{N} \sum_{m=1}^{M} \log P(X_n|\text{parents}(X_n), D_m), \end{aligned}$$

where M is the number of training data examples D_m, and X_n is one of the N nodes in graph G.

MARKOV RANDOM FIELDS

Bayesian networks are inherently asymmetric, since each edge had an arrow on it. If we remove this constraint, then there is no longer any idea of children and parent nodes. It also

makes the idea of conditional independence that we saw for the Bayesian network easier: two nodes in a Markov Random Field (MRF) are conditionally independent of each other, given a third node, if there is no path between the two nodes that doesn't pass through the third node. This is actually a variation on the Markov property, which is how the networks got their name: the state of a particular node is a function only of the states of its immediate neighbours, since all other nodes are conditionally independent given its neighbours.

A Markov random field (MRF) is a undirected, connected graph

- each node represents a random variable
    - • open circles indicate non-observed random variables
    - • filled circles indicate observed random variables
    - • dots indicate given constants
- links indicate an explicitly modelled stochastic dependence

There is now a simple iterative update algorithm, which is to start with noisy image I and ideal I', and update I so that at each step the energy calculation is lower. So you pick one pixel $I_{xi,xj}$ for some values of x_i , x_j at a time, and compute the energies with this pixel being set to -1 and 1, picking the lower one. In probabilistic terms, we are making the probability p(I, I') higher. The algorithm then moves on to another pixel, either choosing a random pixel at each step or moving through them in some pre-determined order, running through the set of pixels until their values stop changing.

The Markov Random Field Image Denoising Algorithm

- Given a noisy image I and an original image I', together with parameters η, ζ:
- Loop over the pixels of image I:
    - compute the energies with the current pixel being -1 and 1
    - pick the one with lower energy and set its value in I accordingly

HIDDEN MARKOV MODELS (HMMS)

The Hidden Markov Model is one of the most popular graphical models. It is used in speech processing and in a lot of statistical work. The HMM generally works on a set of temporal data. At each clock tick the system moves into a new state, which can be the same as the previous one.

The HMM is the simplest dynamic Bayesian network, a Bayesian network that deals with sequential (often time-series) data.

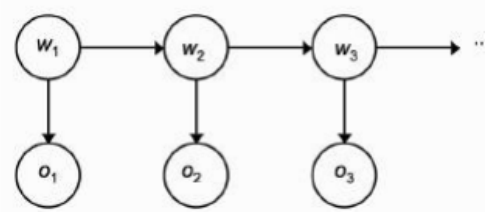
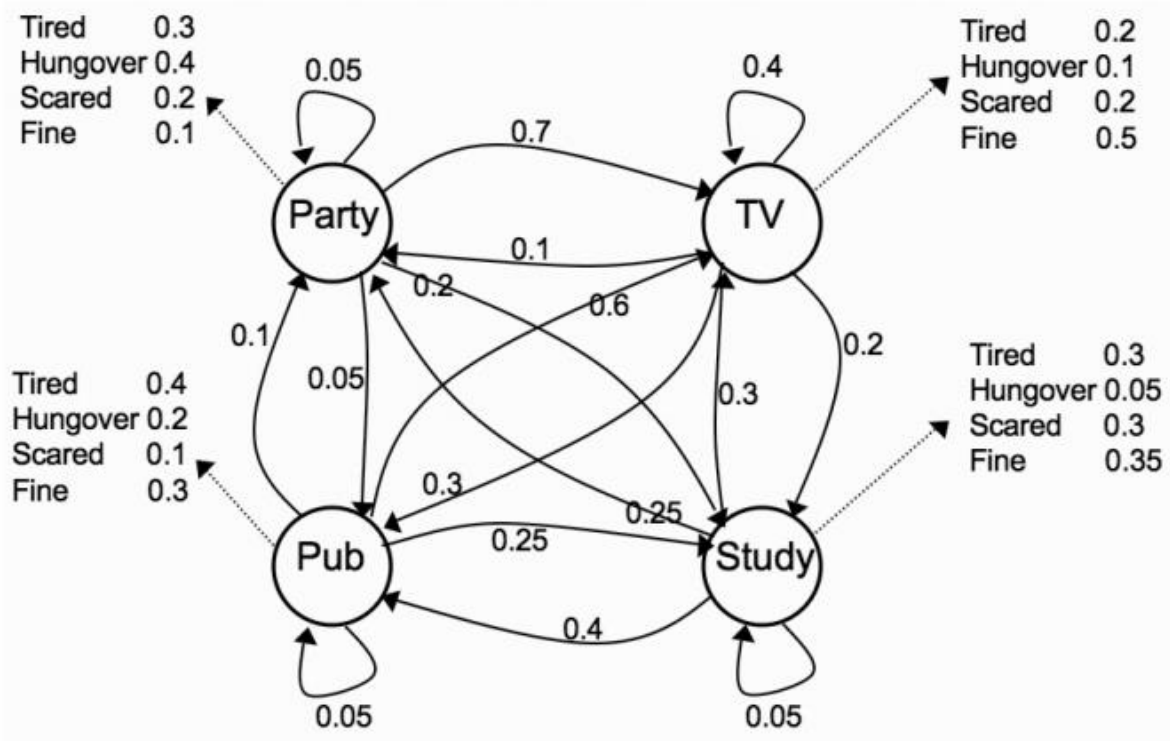


Fig: The Hidden Markov Model is an example of a dynamic Bayesian network. The figure shows the first three states and the related observations unrolled as time progresses.

Example:

The HMM itself is made up of the transition probabilities $a_{i,j}$ and the observation probabilities $b_j(o_k)$, and the probability of starting in each of the states, π_i . So these are the things that need to specify, starting with the transition probabilities (which are also shown in Figure):



	Previous night			
	TV	Pub	Party	Study
TV	0.4	0.6	0.7	0.3
Pub	0.3	0.05	0.05	0.4
Party	0.1	0.1	0.05	0.25
Study	0.2	0.25	0.2	0.05

the observation probabilities:

	TV	Pub	Party	Study
Tired	0.2	0.4	0.3	0.3
Hungover	0.1	0.2	0.4	0.05
Scared	0.2	0.1	0.2	0.3
Fine	0.5	0.3	0.1	0.35

The Forward Algorithm

Suppose the following observations: O = (tired, tired, fine, hungover, hungover, scared, hungover, fine),the probability that my observations O = {o(1), . . . , o(T)} come from the model can be computed using simple conditional probability.

$$P(O) = \sum_{r=1}^{R} P(O|\Omega_r)P(\Omega_r).$$

The r index here describes a possible sequence of states, so Ω_1 is one sequence, Ω_2 another, and so on. Considering Markov property,

$$P(\Omega_r) = \prod_{t=1}^{T} P(\omega_j(t)|\omega_i(t-1)) = \prod_{t=1}^{T} a_{i,j},$$

and

$$P(O|\Omega_r) = \prod_{t=1}^{T} P(o_k(t)|\omega_j(t)) = \prod_{t=1}^{T} b_j(o_k).$$

So Equation P(O) can be written as:

$$
\begin{aligned}
P(O) &= \sum_{r=1}^{R} \prod_{t=1}^{T} P(o_k(t)|\omega_j(t)) P(\omega_j(t)|\omega_i(t-1)) \\
&= \sum_{r=1}^{R} \prod_{t=1}^{T} b_j(o_k) a_{i,j}.
\end{aligned}
$$

Since the probability of each state only depends on the data at the current and previous timestep (o(t), ω(t), ω(t−1)) we can build up our computation of P(O) one timestep at a time. This is known as the forward trellis. To construct the trellis we introduce a new variable $\alpha_i(t)$ that describes the probability that at time t the state is ω_i and that the first (t − 1) steps all matched the observations o(t):

$$
\alpha_j(t) = \begin{cases} 0 & t = 0, j \neq \text{initial state} \\ 1 & t = 0, j = \text{initial state} \\ \sum_i \alpha_i(t-1) a_{i,j} b_j(o_t) & \text{otherwise.} \end{cases}
$$

where b_j (o_t) means the particular emission probability of output o_t.

$a_{i,j}$ is the transition probability of going from state I to state j, so if there are N states, then it is of size N ×N; $b_i(o)$ is the transition probability of emitting observation o in state i, so it is of size N ×O, where O is the number of different observations that there are (four in the example).

It will be useful to introduce four more variables, all of which are probabilities that are conditioned on the observation sequence and the model:

• $\alpha_{i,t}$, which is the probability of getting the observation sequence up to time t and being in state i at time t (size N × t),

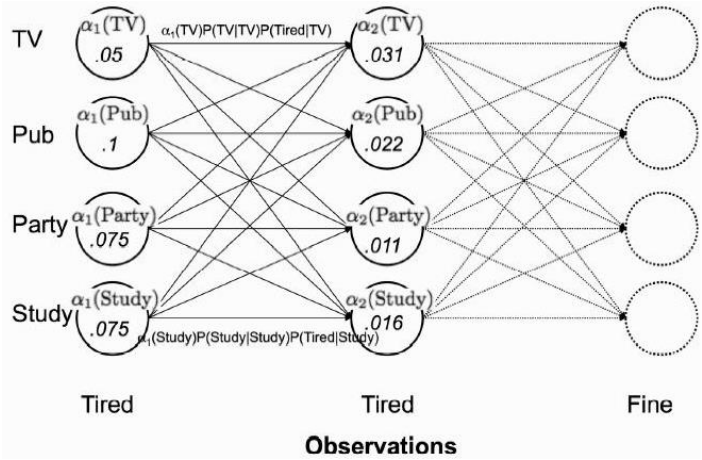
• $\beta_{i,t}$, which is the probability of the sequence from t+ 1 to the end given that the state is i at time t, • $\delta_{i,t}$, which is the highest probability of any path that reaches state i at time t,

• $\xi_{i,j,t}$, which is the probability of being in state i at time t, state j at time t + 1, and so is an N × N × T matrix.

Since $\alpha_{i,t}$ is the probability of getting the observation sequence up to time t and being in state i at time t conditioned on the model and the observations, the probability of the whole observation sequence given the model is just $\sum_{i=1}^{N} \alpha_{i,T}$.

The HMM Forward Algorithm

- Initialise with $\alpha_{i,0} = \pi_i b_i(o_0)$
- For each observation in order o_t, $t = 1, \ldots, T$
  - for each of the N_s possible states s:
    - $\alpha_{s,t+1} = b_s(o_{t+1}) \left(\sum_{i=1}^{N} (\alpha_{i,t} a_{i,s}) \right)$



The Viterbi Algorithm

- Used to solve is the decoding problem of working out the hidden states.

The HMM Viterbi Algorithm

- Start by initialising $\delta_{i,0}$ by $\pi_i bi(o_0)$ for each state i, $\phi_0 = 0$
  - run forward in time t:
    - for each possible state s:
      - $\delta_{s,t} = \max_i(\delta_{i,t-1} a_{i,s}) b_s(o_t)$
      - $\phi_{s,t} = \arg\max_i(\delta_{i,t-1} a_{i,s})$
  - set q_T^*, the most likely end hidden state to be $q_T^* = \arg\max_i \delta_{i,T}$
  - run backwards in time computing:
    - $q_{t-1}^* = \phi_{q_t^*,t}$

TRACKING METHODS

Two methods of performing tracking - the Kalman filter and the particle filter.

The Kalman Filter

The Kalman filter is a recursive estimator. It makes an estimate of the next step, then computes an error term based on the value that was actually produced in the next step, and tries to correct

it. It then uses both of those to make the next prediction, and iterates this procedure. It can be seen as a simple cycle of predict-correct behaviour, where the error at each step is used to improve the estimate at the next iteration.

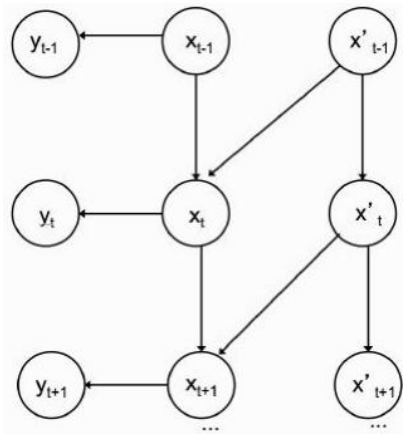


FIGURE : A representation of the Kalman filter with time derivatives (such as for tracking) as a graphical model.

Much of the jargon that is associated with the Kalman filter is familiar to us: the state, which is hidden, consists of the variables that we want to know, which we see through noisy observations over time. There is a transition model that tells us how states change from one to another, and an observation model (also called the sensor model here) that tells us how states lead to observations. The underlying idea is that there is some time-varying process that is generating a set of noisy outputs, where there are two sources of noise: process noise, which represents the fact that the process changes over time, but we don't know how, and observation (or measurement) noise, which is the errors that are made in the readings. Both are assumed to be independent of each other, and zero mean Gaussians. We write the process as a stochastic difference equation in x, which has n dimensions:

$$\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t + \mathbf{w}_t, \qquad (16.21)$$

where A_t is an n × n matrix that represents the non-driven part of the underlying process, B is an n × l matrix that represents the driving force, and u is the l-dimensional driving force. w is the process noise, which is assumed to be zero mean with standard deviation Q.

The observations that we make are m-dimensional

$$\mathbf{y}_t = \mathbf{H}\mathbf{x}_t + \mathbf{v}_t, \qquad (16.22)$$

where m × n matrix H describes how measurements of the state are measured, and v is the measurement noise, which is also assumed to be zero mean, but with standard deviation R.

The basic idea is to make a prediction and then correct it when the next observation is available, i.e., at the next timestep. We will use $\hat{x}$ and $\hat{y}$ as the estimates, so $\hat{y}_{t+1} = HA\hat{x}_{t+1}$ and so the error is $y_{t+1} - \hat{y}_{t+1}$; that is the difference between what was actually observed and what we predicted (without measurement noise). Since this is a probabilistic process with Gaussian distributions, we can also keep a predicted covariance matrix that goes with it: $\hat{\Sigma}_{t+1} = A\Sigma_t A^T + Q$ (which is $E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^T]$). The Kalman filter weights these error computations by how much trust the filter currently has in its predictions; these weights are known as the Kalman gain and are computed by:

$$\mathbf{K}_{t+1} = \hat{\mathbf{\Sigma}}_{t+1}\mathbf{H}^T \left(\mathbf{H}\hat{\mathbf{\Sigma}}_{t+1}\mathbf{H}^T + \mathbf{R}\right)^{-1}. \tag{16.27}$$

This equation comes from minimising the mean-square error.

Using it, the update for the estimate is:

$$\mathbf{x}_{t+1} = \hat{\mathbf{x}}_{t+1} + \mathbf{K}_{t+1}\left(\mathbf{z}_{t+1} - \mathbf{H}\hat{\mathbf{x}}_{t+1}\right), \tag{16.28}$$

All that is then required is to update the covariance estimate:

$$\mathbf{\Sigma}_{t+1} = (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})\hat{\mathbf{\Sigma}}_{t+1}, \tag{16.29}$$

where I is the identity matrix of the relevant size.

The Kalman Filter Algorithm

- Given an initial estimate $\mathbf{x}(0)$
- For each timestep:
  - **predict the next step**
    - predict state as $\hat{\mathbf{x}}_{t+1} = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t$
    - predict covariance as $\hat{\mathbf{\Sigma}}_{t+1} = \mathbf{A}\mathbf{\Sigma}_t\mathbf{A}^T + \mathbf{Q}$
  - **update the estimate**
    - compute the error in the estimate, $\boldsymbol{\epsilon} = \mathbf{y}_{t+1} - \mathbf{H}\mathbf{A}\mathbf{x}_{t+1}$
    - compute the Kalman gain using Equation (16.27)
    - update the state using Equation (16.28)
    - update the covariance using Equation (16.29)

The Particle Filter

The idea is to use sampling to keep track of the state of the probability distribution. This is known as sequential sampling, since we are using a set of samples for time t to estimate the process at time t + 1, and then resampling from there. One benefit of sampling methods is that we don't have to hold on to the Markov assumption. In tracking, prior history can be useful, which means that the Markov assumption is a bad one. The proposal distribution is generally written as $q(x_{t+1}|x_{0:t}, y_{0:t})$ to make this dependence clear, and the proposal distribution that is generally used in the estimated transition probabilities $p(\hat{x}_{t+1}|x_{0:t}, y_{01:t})$, since it is a simple distribution that is related to the process.

The Particle Filter Algorithm

- Sample $\mathbf{x}_0^{(i)}$ from $p(\mathbf{x}_0$ for $i = 1 \ldots N$
- For each timestep:
  - **importance sample**
  - for each datapoint:
    - sample $\hat{\mathbf{x}}_t^{(i)}$ from $q(\mathbf{x}_t^{(i)}|\mathbf{x}_{0:t-1}^{(i)}, \mathbf{y}_{1:t})$
    - add $\hat{\mathbf{x}}_t^{(i)}$ onto the list of samples to get $\mathbf{x}_{0:t}^{(i)}$ from $\mathbf{x}_{0:t-1}^{(i)}$
    - compute the importance weights:

$$w_t^{(i)} = w_{t-1}^{(i)} \frac{p(\mathbf{y}_t|\hat{\mathbf{x}}_t^{(i)})p(\mathbf{x}_t^{(i)}|\hat{\mathbf{x}}_{t-1}^{(i)})}{q(\mathbf{x}_t^{(i)}|\mathbf{x}_{0:t-1}^{(i)}, \mathbf{y}_{1:t})} \tag{16.38}$$

  - normalise the importance weights by dividing by their sum
  - **resample the particles**
    - retain particles according to their importance weights, so that there might be several copies of some particles, and none of others to get the same number of particles approximately sampled from $p(\mathbf{x}_{0:t}^{(i)}|\mathbf{y}_{1:t})$