Spherical Gaussian

The distribution can be represented by a circle centered at mean μ with radius σ.
We can generate points from the following spherical Gaussian Distribution, each with a **likelihood** of

$$P(x|\mu,\sigma^2) = N(x;\mu,\sigma^2 I) \quad = \frac{1}{(2\pi\sigma^2)^{d/2}} exp(-\frac{1}{2\sigma^2}||x-\mu||^2)$$

- x: Point
- μ: Mean
- σ: Variance
- d: Dimension

The probability of generating points away from the mean μ decreases as they get further away regardless of direction.

Overall Objective Function

We try to find the best Gaussian that fits the data with the criterion of maximum likelihood (ML).
The likelihood of the training data is calculated as follows

$$\begin{aligned}\ell(S_n|\mu,\sigma^2) &= \prod_{t=1}^{n} p(x^{(t)}|\mu,\sigma^2) \\ &= \sum_{t=1}^{n} \log p(x^{(t)}|\mu,\sigma^2) \\ &= \sum_{t=1}^{n} [-\frac{d}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}||x^{(t)}-\mu||^2] \\ &= -\frac{dn}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=1}^{n}||x^{(t)}-\mu||^2\end{aligned}$$

$$\hat{\mu} = \frac{1}{n}\sum_{t=1}^{n}\vec{x}^{(t)}$$
$$\sigma^2 = \frac{1}{dn}\sum_{t=1}^{n}||x^{(t)}-\mu||^2$$

Deriving maximum likelihood estiamtor $\hat{\mu}$

$$\frac{\partial \ell(S_n|\mu,\sigma^2)}{\partial \mu} = 0$$
$$\frac{\partial \ell(S_n|\mu,\sigma^2)}{\partial \mu} = -\frac{1}{2\sigma^2}\cdot 2\cdot\sum_{t=1}^{n}\vec{\mu}-\vec{x}^{(t)}$$
$$= \vec{0}$$

Deriving maximum likelihood estiamtor $\hat{\sigma^2}$

$$\frac{\partial \ell(S_n|\mu,\sigma^2)}{\partial \sigma^2} = 0$$
$$\frac{\partial \ell(S_n|\mu,\sigma^2)}{\partial \sigma^2} = -\frac{dn}{2}\cdot\frac{1}{\sigma^2}\cdot 2\pi - \frac{1}{2\sigma^4}\sum_{t=1}^{n}||x^{(t)}-\mu||^2$$
$$\frac{dn}{2}\cdot\frac{1}{\sigma^2} = \frac{1}{2\sigma^4}\sum_{t=1}^{n}||x^{(t)}-\mu||^2$$
$$dn = \frac{1}{\sigma^2}\sum_{t=1}^{n}||x^{(t)}-\mu||^2$$

Mixture of Gaussians

Assuming k clusters is optimal for describing the data,

$$P(x|\mu^{(i)},\sigma_i^2), \text{for } i = 1,...,k$$

So how do we generate data points from the mixture? We first sample index i to see which cluster we should use. In other words, we sample i from a multinomial distribution governed by $p_1,\ldots,p_k$, where $\sum_{i=1}^{k} p_i = 1$. Think of throwing a biased k-faced die. Larger p_i means that we generate more points from that clusters. Once we know the cluster, we can sample x from the corresponding Gaussian. More precisely, $i \sim Multinomial(p_1,\ldots,p_k) \qquad x \sim P(x|\mu^{(i)},\sigma_i^2)$

For easier representation, we let θ specify all parameters for the mixture model

$$\theta = \{\mu^{(1)},...\mu^{(k)},\sigma_1^2,...,\sigma_k^2,p_1,...p_k\} \qquad P(x|\theta) = \sum_{i=1}^{k} p_i P(x|\mu^{(i)},\sigma_i^2)$$

where $p_1...p_k$ specify the frequency of points we would expect to see in each cluster.

Labeled Case

If data points are already labeled (assigned to a single cluster), we could estimate the Gaussian models same as before, and even evaluate the cluster sizes based on the actual number of points.

Let $\delta(i|t)$ be an indicator that tells us whether $x^{(t)}$ should be assigned to cluster i.

- $\delta(i|t) = 1$, if $x^{(t)}$ is assigned to i
- $\delta(i|t) = 0$, otherwise

Maximum Likelihood Objective

$$\sum_{t=1}^{n}\Big[\underbrace{\sum_{i=1}^{k}\delta(i|t)\log\left(p_i\cdot p(x^{(t)}|\mu^{(i)},\sigma_i^2)\right)}_{\text{Only one term is non-zero, as specified by } \delta(i|t)}\Big] \quad \sum_{i=1}^{k}\Big[\underbrace{\sum_{t=1}^{n}\delta(i|t)\log\left(p_i\cdot p(x^{(t)}|\mu^{(i)},\sigma_i^2)\right)}_{\text{The objective for all points in } i\text{-th cluster}}\Big]$$

number of points assigned to cluster i: $\hat{n}_i = \sum_{t=1}^{n}\delta(i|t)$

fraction of points in cluster i: $\hat{p}_i = \frac{\hat{n}_i}{n}$

mean of points in cluster i: $\hat{\mu}^{(t)} = \frac{1}{n}\sum_{t=1}^{n}\delta(i|t)x^{(t)}$

mean squared spread in cluster i: $\hat{\sigma^2}_i = \frac{1}{d\hat{n}_i}\sum_{t=1}^{n}\delta(i|t)||x^{(t)}-\hat{\mu}^{(i)}||^2$

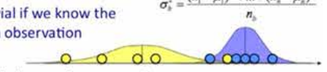
Unlabeled Case

Now $\delta(i|t)$ is not given, but we can apply the EM-algorithm to derive the respective $\delta(i|t)$.

Mixture models in 1-d

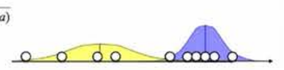
- Observations $x_1 \ldots x_n$
  - K=2 Gaussians with unknown μ, σ²
  - estimation trivial if we know the source of each observation

$$\mu_b = \frac{x_1+x_2+\ldots+x_{n_b}}{n_b}$$
$$\sigma_b^2 = \frac{(x_1-\mu_b)^2+\ldots+(x_{n_b}-\mu_b)^2}{n_b}$$



- What if we don't know the source?
- If we knew parameters of the Gaussians (μ, σ²)
  - can guess whether point is more likely to be a or b

$$P(b|x_i) = \frac{P(x_i|b)P(b)}{P(x_i|b)P(b)+P(x_i|a)P(a)}$$
$$P(x_i|b) = \frac{1}{\sqrt{2\pi\sigma_b^2}}\exp\left(-\frac{(x_i-\mu_b)^2}{2\sigma_b^2}\right)$$



Expectation Maximization (EM)

- Chicken and egg problem
  - need (μ_a, σ_a^2) and (μ_b, σ_b^2) to guess source of points
  - need to know source to estimate (μ_a, σ_a^2) and (μ_b, σ_b^2)
- EM algorithm
  - start with two randomly placed Gaussians (μ_a, σ_a^2), (μ_b, σ_b^2)
  - for each point: P(b|x_i) = does it look like it came from b?
  - adjust (μ_a, σ_a^2) and (μ_b, σ_b^2) to fit points assigned to them
  - iterate until convergence

EM Algorithm Process

We need to first initialize the mixture parameters. For example, we could initialize the means $\mu^{(1)},\ldots,\mu^{(k)}$ as in the k-means algorithm, and set the variances σ_i^2 all equal to the overall data variances:

$$\hat{\sigma}^2 = \frac{1}{dn}\sum_{t=1}^{n}\left\|x^{(t)}-\hat{\mu}\right\|^2$$

Since we have no information about the cluster sizes, we will set $p_i = 1/k, i = 1,\ldots,k$.

- **E-Step:** Softly assign points to clusters according to the posterior probabilities

$$p(i|t) = \frac{p_i P(x|\mu^{(i)},\sigma_i^2)}{P(x|\theta)} = \frac{p_i P(x|\mu^{(i)},\sigma_i^2)}{\sum_{j=1}^{k} p_j P(x|\mu^{(j)},\sigma_j^2)}$$

Here $\sum_{i=1}^{k} p(i|t) = 1$. These are exactly analogous to (but soft versions of) $\delta(i|t)$ in the labeled case. Each point $x^{(t)}$ is assigned to cluster i with weight $p(i|t)$. The larger this weight, the more strongly we require cluster i to generate this point in the M-step below.

- **M-Step:** Once we have $p(i|t)$, we pretend that we were given these assignments (as softly labeled examples) and can use them to estimate the Gaussians separately, just as in the labeled case.

$\hat{n}_i = \sum_{t=1}^{n} p(i|t)$ (effective number of points assigned to cluster i)

$\hat{p}_i = \frac{\hat{n}_i}{n}$ (fraction of points in cluster i)

$\hat{\mu}^{(i)} = \frac{1}{\hat{n}_i}\sum_{t=1}^{n} p(i|t)x^{(t)}$ (weighted mean of points in cluster i)

$\hat{\sigma}_i^2 = \frac{1}{d\hat{n}_i}\sum_{t=1}^{n} p(i|t)\left\|x^{(t)}-\hat{\mu}^{(i)}\right\|^2$ (weighted mean squared spread)

We will then use these parameters in the E-step, and iterate.

The two types of EM algorithm both come with a theoretical guarantee: they converge to a local maximum of the objective function, which is the joint log-likelihood of the observed data:

$$p'(\mathbf{y}|\mathbf{x}) = \lim_{t\to+\infty}\frac{(p(\mathbf{x},\mathbf{y}))^t}{\sum_{\mathbf{y}'}(p(\mathbf{x},\mathbf{y}'))^t} = \begin{cases}1 & \text{if } \mathbf{y} \equiv \arg\max_{\mathbf{y}} p(\mathbf{x},\mathbf{y}) \\ 0 & \text{o.w.}\end{cases}$$

What does this mean? In this case, we can see that for each input sequence $\mathbf{x}$ we only consider a single optimal $\mathbf{y}$ sequence, which is the output of the Viterbi decoding algorithm. In other words, we are essentially making an hard assignment to each input sequence $\mathbf{x}$ at the E-step of the EM algorithm. This shows that theoretically the hard EM can be viewed as a special case of the extended version of the soft EM algorithm that is augmented with a temperature t, and when

The Generative Process

1. Set $y_0 = \texttt{START}$ (we always start from this START symbol) and let $i = 1$.
2. Generate tag y_i from the conditional distribution $p(y_i|y_{i-1})$ where y_{i-1} already has a value (*e.g.*, $y_{i-1} = \texttt{START}$ when $i = 1$)
3. If $y_i = \texttt{STOP}$, we terminate the process and return $y_0, y_1, \ldots, y_i, x_1, \ldots, x_{i-1}$. Otherwise we generate x_i from the emission distribution $p(x_i|y_i)$.
4. Set $i = i+1$, and return to step 2.

Computing Joint Likelihood of a HMM

To calculate the probabilities of the sequence of words $(x_0...x_n)$ generated given the tags $(y_0...y_{n+1})$

$$p(x_1...x_n, y_0...y_{n+1}) = p(y_0...y_{n+1})\cdot p(x_1...x_n|y_0...y_{n+1})$$

We assumed strong independence between the variables such that

$$p(y_2|y_1,y_0) \approx p(y_2|y_1)$$

Expanding the joint probabilitiy terms individually,

$$p(y_0...y_{n+1}) = \prod_{j=0}^{n} p(y_n+1|y_n)$$
$$p(x_1...x_n|y_0...y_n+1) = \prod_{j=1}^{n} p(x_j|y_j)$$
$$\text{Hence, } p(x_1...x_n, y_0...y_{n+1}) = \prod_{j=0}^{n} p(y_n+1|y_n)\cdot\prod_{j=1}^{n} p(x_j|y_j)$$

For simplicity, we rewrite the notation of individual transmission and emission probabilities.

$$\text{Transmission Probability} \quad a_{u,v} = \frac{count(u,v)}{count(u)}$$
$$\text{Emission Probability} \quad b_{u(o)} = \frac{count(u\to o)}{count(u)}$$

$$\prod_{j=0}^{n} a_{y_j,y_{j+1}} = \prod_{j=0}^{n} p(y_n+1|y_n)$$
$$\prod_{j=1}^{n} b_{y_j(x_j)} = \prod_{j=1}^{n} p(x_j|y_j)$$
$$\text{Joint Probability} \leftarrow \prod_{j=0}^{n} a_{y_j,y_{j+1}}\cdot\prod_{j=1}^{n} b_{y_j(x_j)}$$

Supervised Learning

Decoding

Finding most probable label sequence y given the word sequence x

$$\begin{aligned}y* &= \arg\max_y p(y|x) \\ &= \arg\max_y p(x,y)/p(x) \\ &= \arg\max_u p(x,y)\end{aligned}$$

Brute Force Enumeration

This method is not feasible once there are too many label sequences.
There will be $O(|T|^n)$ possible sequences.

Viterbi Algorithm

Since the HMM has a simple dependence structure, we can exploit this in a dynamic programming algorithm.

1. We initialize $\pi(0,u)$
   - 1 if $u = START$
   - 0 otherwise
2. For $j = 0...n-1$,
   $\pi(j+1,u) = \max_v\{\pi(j,v)\cdot b_u(x_{j+1})\cdot a_{v,u}\}$
   $y_n^* = \arg\max_u\{\pi(j,u)\cdot a_u, y_{j+1}^*\}$
3. Finally,
   $\pi(n+1,STOP) = \max_v\{\pi(n,v)\cdot a_{v,STOP}\}$

Time complexity: $O((n-1)T^2 + T = T) = O(nT^2)$

- T is the number of nodes in each column, and we carry out n-1 operations for each column.
- Calculation for each node is $O(T)$, and do it for $n*t$ nodes so it is $O(T^2)$.
- $O(T)$ calculation at start and stop nodes.

Space complexity: $O(nT)$

Unsupervised Learning

We can use EM algorithms to learn model parameters in an unsupervised manner.

Hard EM

E-step

1. Assign each observed sequence of outputs (data point) as $x^{(1)}...x^{(m)}$
2. Assign each x to a single state sequence/tag sequence y *(no partial membership)*.
3. For each observation sequence, use Viterbi algorithm to find the most probable state sequence $y^{(t)*}$

M-step

$$Y^* = \arg\max_Y P(Y|X)$$

Parameter estimation using MLE with labeled data from the E-step

$$\text{Transmission Probability} \quad a_{u,v} = \frac{count(u,v)}{count(u)} \quad \text{for any } u,v \in \{0,1,2...N+1\}$$
$$\text{Emission Probability} \quad b_{u(o)} = \frac{count(u\to o)}{count(u)} \quad \text{for any } u \in \{0,1,2...N+1\}, o \in \sum$$

Soft EM

For soft EM, we must evaluate a posterior probability over possible tag sequences, so we cannot use Viterbi algorithm.

E-step

Re-assign the partial memberships for each $\mathbf{x}^{(i)}$. In this case we would like to calculate *expected* counts, added across sequences:

A distribution over possible ys

$$\text{count}(u,v) = \sum_{i=1}^{m}\sum_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}^{(i)})\text{count}(\mathbf{x}^{(i)},\mathbf{y},u\to v)$$

The number of times we see a transition from u to v in the sequence pair $(\mathbf{x}^{(i)},\mathbf{y})$

M-step

Re-estimate transition probabilities based on the expected counts:

$$a_{u,v} = \frac{\text{Count}(u;v)}{\sum_{v'}\text{Count}(u;v')}$$

where the denominator ensures that

$$\sum_{v'=1}^{N+1} a_{u,v'} = 1$$

The sum is over an **exponential** number of possible hidden state sequences $\mathbf{y}$. How do we calculate such terms efficiently? Suppose we could efficiently calculate marginal posterior probabilities

$$p(y_j = u, y_{j+1} = v|\mathbf{x};\theta) = \sum_{\mathbf{y}:y_j=u,y_{j+1}=v} p(\mathbf{y}|\mathbf{x};\theta)$$

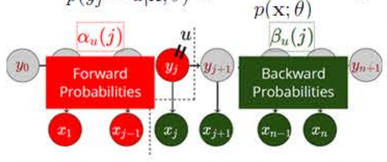
for any $u,v \in 0,\ldots,N+1, j \in 1,\ldots,n$. These are the posterior probabilities that the state in position j was u and we transitioned into v at the next step. The probability is conditioned on the observed sequence $\mathbf{x}$ and the current setting of the model parameters θ. Now, under this assumption,

$$\sum p(\mathbf{y}|\mathbf{x}^{(i)};\theta)\text{Count}(\mathbf{x}^{(i)},\mathbf{y},u\to v) = \sum^{n} p(y_j = u, y_{j+1} = v|\mathbf{x}^{(i)};\theta)$$

Inference

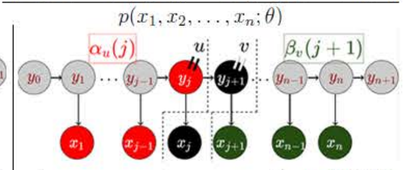
To compute the posterior possibilities efficiently for any $u \in 1...N, j \in 1...n$,

$$p(y_j = u|\mathbf{x};\theta) = \frac{p(y_j = u,\mathbf{x};\theta)}{p(\mathbf{x};\theta)} = \frac{p(x_1,x_2,\ldots,x_{j-1},y_j = u,x_j,x_{j+1},\ldots,x_n;\theta)}{p(x_1,x_2,\ldots,x_n;\theta)}$$



$$p(x_1,x_2,\ldots,x_{j-1},y_j = u,x_j,x_{j+1},\ldots,x_n;\theta)$$
$$= p(x_1,x_2,\ldots,x_{j-1},y_j = u;\theta) \quad \alpha_u(j)$$
$$\times\, p(x_j,x_{j+1},\ldots,x_n|y_j = u;\theta) \quad \beta_u(j)$$

$$p(y_j = u|\mathbf{x};\theta) = \frac{p(x_1,x_2,\ldots,x_{j-1},y_j=u,x_j,x_{j+1},\ldots,x_n;\theta)}{\sum_v p(x_1,x_2,\ldots,x_{k-1},y_k=v,x_k,x_{k+1},\ldots,x_n;\theta)} = \frac{\alpha_u(j)\beta_u(j)}{\sum_v \alpha_v(k)\beta_v(k)}$$



$$p(x_1,\ldots,x_{j-1},y_j = u,x_j,\ldots,x_n;\theta) = \alpha_u(j)\,\beta_u(j)$$

$$p(y_j = u,y_{j+1} = v|\mathbf{x};\theta) = \frac{p(x_1,x_2,\ldots,x_{j-1},y_j=u,x_j,y_{j+1}=v,x_{j+1},\ldots,x_n;\theta)}{\sum_v p(x_1,x_2,\ldots,x_{k-1},y_k=v,x_k,x_{k+1},\ldots,x_n;\theta)} = \frac{\alpha_u(j)\cdot b_u(x_j)\cdot a_{u,v}\cdot\beta_v(j+1)}{\sum_v \alpha_v(k)\beta_v(k)}$$

$\alpha_u(j+1) = \sum_v \alpha_v(j)a_{v,u}b_v(x_j)$

$\beta_u(j) = \sum_v a_{u,v}b_u(x_j)\beta_v(j+1)$

$\alpha_u(j)$ is the probability of emitting the symbols $x_1,\ldots,x_{j-1}$ and ending in state u in position j without (yet) emitting the corresponding output symbol. These are analogous to the $\pi(k,v)$ probabilities in the Viterbi algorithm with the exception that $\pi(k,v)$ included generating the corresponding observation in position k. Note that, unlike before, we are summing over all the possible sequences of states that could give rise to the observations $x_1,\ldots,x_{j-1}$. In the Viterbi algorithm, we maximized over the tag sequences.

The Forward-backward Algorithm

For every state sequence $y_1, y_2, \ldots, y_n$ there is

Time Complexity: $O(nT^2)$
- $O(T)$ operations for T nodes at n positions

Time complexity of 1 EM iteration: $O(mnT^2)$

- A path through the graph (shown in class on the whiteboard) that has the sequence of states $\texttt{START}, \langle 1,y_1\rangle, \ldots, \langle n,y_n\rangle, \texttt{STOP}$
- The path associated with state sequence $\mathbf{y} = y_1,\ldots,y_n$ has score equal to $p(\mathbf{x},\mathbf{y};\theta)$.
- $\alpha_u(j)$ is the sum of scores of all paths from START to the state (j,u).
- $\beta_u(j)$ is the sum of scores of all paths from state $\langle j,u\rangle$ to the final state STOP.

Given an input sequence $x_1,\ldots,x_n$ for any $u \in 1,\ldots,N, j \in 1,\ldots,n$ the forward and backward probability can be calculated recursively.

Going Forward

- $\alpha_u(j)$ = the sum of the scores of all paths from STA
- For the first node, there is no emission probability.

$$\alpha_u(1) = \alpha_{START,u}$$
$$= p(x_0, y_1 = u)$$

- For the other nodes,

$$\alpha_u(j+1) = p(x_1...x_j, y_{j+1} = u)$$
$$= \sum_v p(x_1...x_j, y_{j+1} = u, y_j = v$$
$$= \sum_v p(x_1...x_{j-1}, y_j = v, x_j, y_{j+}$$
$$= \sum_v p(x_1...x_{j-1}, y_j = v)\cdot b_v(x_j$$
$$= \sum_v \alpha_v(j)\cdot b_v(x_j)\cdot\alpha_{v,u}$$

Going Backward

- $\beta_u(j)$ = the sum of the scores of all paths from node u at j to STOP.
- Base case:

$$\beta_u(n) = a_{u,\texttt{STOP}}b_u(x_n) \;\; \forall u \in 1,\ldots,N-1$$

- For the other nodes,

$$\beta_u(j) = p(x_j...x_n|y_j = u)$$
$$= \sum_v p(x_j...x_n, y_{j+1} = v|y_j = u)$$
$$= \sum_v p(x_j, y_{j+1} = v, x_{j+1}...x_n|y_j = u)$$
$$= \sum_v b_u(x_j)\cdot a_{u,v}\cdot p(x_{j+1}...x_n|y_{j+1} = v)$$
$$= \sum_v b_u(x_j)\cdot a_{u,v}\cdot\beta_v(j+1)$$

$$\forall u \in 1,\ldots,N-1, j = n-1,\ldots,1$$

Bayesian Networks

Bayesian Networks are **Directed Acyclic Graphs** (DAG) over the variables.

- We usually generate the graph from x_1 first, and each node x_i has a Conditional Probability Distribution $P(X_i|\arg P(X_i))$
- If there is a directed edge from $x_1 \to x_3$, then x_1 is a parent of x_3. And once we know the parents, we can write the probability distribution over all the variables with **chain rule** as

$$P(X_1 = x_1,...,X_d = x_d) = \prod_{i=1}^{d} P(X_i = x_i|\arg P(X_i))$$

Relationship	Y is not given	Y is given
X→Y→Z	✔: ○→○ →○	✘: ○ →● →○
X←Y→Z	✔:○ ← ○ → ○	✘: ○ ← ● → ○
X→Y←Z	✘: ○ → ○ ←○	✔: ○ → ● ←○ **Explaining Away**

Are X and Z dependent?

- ○ refers to known variable,
- ● refers to unknown variable
- →: causal direction
- ←: evidential direction

Learning Bayesian networks

where $\theta_i(x_i|\mathbf{x}_{pa_i})$ are the probability tables that we must estimate.

$$P(X_1 = x_1,\ldots,X_d = x_d) = \prod_{i=1} P(X_i = x_i|\mathbf{X}_{pa_i} = \mathbf{x}_{pa_i}) = \prod_{i=1}\theta_i(x_i|\mathbf{x}_{pa_i})$$

$$l(D;\theta;G) = \sum_{t=1}^{m}\log\left[\prod_{i=1}^{d}\theta_i(x_i^{(t)}|\mathbf{x}_{pa_i}^{(t)})\right] = \sum_{t=1}^{m}\sum_{i=1}^{d}\log\theta_i(x_i^{(t)}|\mathbf{x}_{pa_i}^{(t)}) = \sum_{i=1}^{d}\left[\sum_{t=1}^{m}\log\theta_i(x_i^{(t)}|\mathbf{x}_{pa_i}^{(t)})\right]$$

$$\sum_{t=1}^{m}\log\theta_i(x_i^{(t)}|\mathbf{x}_{pa_i}^{(t)}) = \sum_{x_i,\mathbf{x}_{pa_i}}\text{Count}\Big((x_i,\mathbf{x}_{pa_i}) \text{ in } D\Big)\log\theta_i(x_i|\mathbf{x}_{pa_i})$$

where $\text{Count}\Big((x_i,\mathbf{x}_{pa_i}) \text{ in } D\Big)$ gives the number of observations in data D for which $X_i = x_i$ and $\mathbf{X}_{pa_i} = \mathbf{x}_{pa_i}$ So, if we fix $\mathbf{x}_{pa_i}$, then $\text{Count}\Big((\cdot,\mathbf{x}_{pa_i}) \text{ in } D\Big)$ specifies the counts for a multinomial $\theta_i(\cdot|\mathbf{x}_{pa_i})$. The corresponding maximum likelihood parameter estimate is simply (analogously to a single multinomial)

$$\hat{\theta}_i(x_i|\mathbf{x}_{pa_i}) = \frac{\text{Count}\Big((x_i,\mathbf{x}_{pa_i}) \text{ in } D\Big)}{\text{Count}\Big((\mathbf{x}_{pa_i}) \text{ in } D\Big)}, x_i, \in \{1,\ldots,r_1\}$$

where $\text{Count}\Big((\mathbf{x}_{pa_i}) \text{ in } D\Big)$ is the number of times we see the pattern $\mathbf{x}_{pa_i}$ in D:

$$\text{Count}\Big((\mathbf{x}_{pa_i}) \text{ in } D\Big) = \sum_{x_i}\text{Count}\Big((x_i',\mathbf{x}_{pa_i}) \text{ in } D\Big)$$

Repeating the procedure for each setting of $\mathbf{x}_{pa_i}$, and for different variables, yields the maximum likelihood parameter estimates $\hat{\theta}_i(x_i|\mathbf{x}_{pa_i}), i = 1,\ldots,d$.

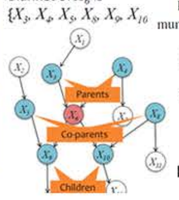
Markov Blanket

Def: the **co-parents** of a node are the parents of its children

Def: the **Markov Blanket** of a node is the set containing the node's parents, children, and co-parents.

Thm: a node is **conditionally independent** of every other node in the graph given its **Markov blanket**

Example: The Markov Blanket of X_6 is $\{X_3, X_4, X_5, X_8, X_9, X_{16}$



Reinforcement Learning

Problem: $f: S \to A$

- S: Set of States
- A: Set of Actions

Transition Probability

- Moving from one state to another state is not definite
- $T(s,a,s') = p(s'|s,a)$

Reward

- Incentivize a target state to be reached / not reached.
- Can be positive (reward) / negative (penalty)
- $R(s,a,s')$: Generally
- $R(s')$: if the system only requires the target state to be reached.

Markov Decision Process

Utility (Long Term Reward)

- Maximizing the rewards alone is not sufficient to incentivise the program to reach the final state that you want. The program may infinitely loop around a few states
- Hence we apply a **discount** γ to long-term rewards.

Given
- a set of states S
- a set of actions A
- a transition probability function $T(s,a,s') = p(s'|s,a)$
- a reward function $R(s,a,s')$ or $R(s')$

$$U([s_0,s_1,\ldots,s_N,\ldots,s_{N+k}]) = U([s_0,s_1,\ldots,s_N]) \quad \forall k \geq 1$$
$$U([s_1,s_2,...s_n]) = R(s_1) + \gamma R(s_2) + \gamma_2 R(s_3) + ...$$
$$= \sum_{t=0}^{\infty} R(s_t)$$

Value Iteration Policy

$$\frac{R_{min}}{1-\gamma} = \sum_{t=0}^{\infty}\gamma^t R_{min} \leq U([s_0,s_1,s_2,\ldots]) \leq \sum_{t=0}^{\infty}\gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$$

We first define the following

- $\pi(s)$: a particular policy that specifies the action we should take in state s
- $V^{\pi}(s)$: The value of state s under policy π
- $Q^{\pi}(s,a)$: The Q-value of state s and action a under policy π

and the optimal version of them

- $\pi^*(s)$: a particular policy that specifies the action we should take in state s
- $V^*(s)$: The value of state s under the optimal policy π^*
- $Q^*(s,a)$: The Q-value of state s and action a under policy π^*

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \qquad \pi^*(s) = \arg\max_a Q^*(s,a)$$

$$\begin{aligned}V^*(s) &= Q^*(s,\pi^*(s)) = \max_a Q^*(s,a) \\ &= \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \\ &= \sum_{s'} T(s,\pi^*(s),s')[R(s,\pi^*(s),s') + \gamma V^*(s')]\end{aligned}$$

The Value Iteration Algorithm

- Start with $V_0^*(s) = 0$, for all $s \in S$
- Given V_i^*, calculate the values for all states $s \in S$ (depth $i+1$):

$$V_{i+1}^*(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i^*(s')]$$

- Repeat the above until convergence (until $V_{i+1}(s)$ is nearly $V_i(s)$ for all states)

The convergence of this algorithm is guaranteed. Consider a simple MDP with a single state and a single action.

$$V_{i+1} = R + \gamma V_i$$
$$V^* = R + \gamma V^*$$
$$(V_{i+1} - V^*) = \gamma(V_i - V^*)$$

Thus, after each iteration, the difference between the estimate and the optimal value decreases by a factor $\gamma < 1$. Hence the value for V converges to V^*.

Once the values are computed, we can turn them into the optimal policy:

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

$$= \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$
$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

The Q-Value Iteration Algorithm

- Start with $Q_0^*(s,a) = 0$ for all $s \in S, a \in A$.
- Given $Q_i^*(s,a)$, calculate the Q-values for all states (depth $i+1$) and for all actions a:

$$Q_{i+1}^*(s,a) \leftarrow \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_i^*(s',a')]$$

- Repeat the above step until convergence.

This algorithm has the same convergence guarantees as its value iteration counterpart. As before, the optimal policy can be easily recovered from the Q-values as:

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

Now, we will consider a set-up where neither reward no transitions are known a priori.

Model-based learning

$$T(s,a,s') = \frac{\text{Count}(s,a,s')}{\text{Count}(s,a)}$$
$$R(s,a,s') = \frac{\sum_t R_t(s,a,s')}{\text{Count}(s,a,s')}$$

Model-free Learning

Q-learning Algorithm

- Collect a sample: s,a,s' and $R(s,a,s')$.
- Update Q-values, by incorporating the new sample into a running average over samples:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[R(s,a,s') + \gamma\max_{a'} Q(s',a')\right]$$
$$= Q(s,a) + \alpha\left[R(s,a,s') + \gamma\max_{a'} Q(s',a') - Q(s,a)\right]$$

In fact, it has the same convergence conditions as the gradient ascent algorithm. Each sample corresponds to (s,a), *i.e.*, being in state s and taking action a. We can assign a separate learning rate for each such case, *i.e.*, $\alpha = \alpha_k(s,a)$, where k is the number of times that we saw (s,a). Then, in order to ensure convergence, we should have

$$\sum_k \alpha_k(s,a) \to \infty \qquad \sum_k \alpha_k^2(s,a) < \infty$$

Obviously $\alpha_k(s,a) = 1/k$ satisfies the above two conditions.

Exploration/Exploitation Trade-Off In the Q-learning algorithm, we haven't specified how to select an action for a new sample. One option is to do it fully randomly. While this exploration strategy has a potential to cover a wide spectrum of possible actions, most likely it will select plenty of suboptimal actions, and leads to a poor exploration of the relevant (high reward) part of the state space. Another option is to exploit the knowledge we have already obtained during previous iterations. Remember that once we have Q estimates, we can compute a policy. Since our estimates are noisy in the beginning, and the corresponding policy is weak, we wouldn't want to follow this policy completely. To allow for additional exploration, we select a random action every once in a while. Specifically, with probability ϵ, the action is selected at random and with probability $1-\epsilon$, we follow the policy induced by the current Q-values. Over time, we can decrease ϵ, to rely more heavily on the learned policy as it improves based on the Q-learning updates.