CS221 Notes

Joseph Chang

Contents

1 Machine Learning

1.1 Lecture 2: Machine Learning I

1.1.1 Linear Predictors

1. **Linear predictor**: If we have feature vector $\phi(x)$ and weight vector $\mathbf{w}$, $f_w(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$.
2. **Geometric Intuition:** f_w defines a hyperplane with normal vector w. This hyperplane is known as the **decision boundary**.

1.1.2 Loss Minimization

1. How do we fit $\mathbf{w}$ from training data?
2. **Loss function:** $\text{Loss}(x, y, \mathbf{w})$ quantifies how unhappy you would be if you used $\mathbf{w}$ to make a prediction on x.
   (a) **Score** $= \mathbf{w} \cdot \phi(x)$. Score indicates confidence.
   (b) **Margin** $= (\mathbf{w} \cdot \phi(x))y$. Margin indicates correctness.
3. Loss functions:
   (a) **Zero-one Loss:** $\text{Loss}_{0-1}(x, y, \mathbf{w}) = \mathbf{1}[\mathbf{f_w(x)} \neq \mathbf{y}] = \mathbf{1}[(\mathbf{w} \cdot \phi(\mathbf{x}))\mathbf{y} \leq \mathbf{0}]$.
   (b) **Squared Loss:** $\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(\mathbf{x}) - \mathbf{y})^{\mathbf{2}}$
   (c) **Absolute Deviation Loss:** $\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = |\mathbf{w} \cdot \phi(\mathbf{x}) - \mathbf{y}|$.
   (d) **Logistic Regression:** $\text{Loss}_{\text{logistic}} = \log(1 + \exp(-(\mathbf{w} \cdot \phi(x))y))$
4. Let $\mathcal{D}$ be the training data. $\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \text{Loss}(x, y, \mathbf{w})$.

1.1.3 Stochastic Gradient Descent

1. (Batch) Gradient Descent: $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$.
2. Stochastic Gradient Descent: $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(\mathbf{w})$. Can also change step size $\eta = \frac{1}{t}$ where t is number of updates made so far.

1.2 Lecture 3: Machine Learning II

1.2.1 Features:

1. Generally utilize sparse feature vectors. (Known as **one-hot representation**)
2. Non linearity issues:
   (a) Non-monotonicity: Ex: body temperature. Extremes are bad
   (b) Saturation: Ex: 1000 may not be 10 times more relevant than 100.
   (c) Interactions between features: Add cross terms
3. Adding feature vector creates nonlinearity with respect to x.

1.2.2 Neural Networks

1. $\sigma(z) = (1 + e^{-z})^{-1}$. $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.
2. Neural networks: map an input onto a hidden intermediate representation h, where $h_j = \sigma(v_j \cdot \phi(x))$. **Sigmoid** function is commonly used but recently, **rectified linear** function $\sigma(z) = \max(z, 0)$ is gaining popularity.

1.2.3 Gradients without tears

1. Utilize intermediate functions and chain rule.

1.2.4 Nearest Neighbors

1.3 Lecture 4: Machine Learning III

1.3.1 Generalization

1. Goal is to minimize error on unseen future examples
2. **Approximation error:** how far the entire hypothesis class is from the target predictor f^*. $g = \text{argmin}_{f \in \mathcal{F}} \text{Err}(f)$. Approximation error is $\text{Err}(g) - \text{Err}(f^*)$.
3. **Estimation error:** Let $\hat{f}$ be learning algorithm returned predictor. $\text{Err}(\hat{f}) - \text{Err}(g)$.
4. Approximation error decreases, estimation error increases as hypothesis class size increases.
5. Regularization: Add additional $\frac{\lambda}{2}||w||^2$ term to cost function. Gradient descent algorithm becomes $w \leftarrow w - \eta(\nabla_w[\text{TrainLoss}(w)] + \lambda w)$.

1.3.2 Unsupervised Learning

1. Data has lots of rich latent structures, we want methods to discover this structure automatically
2. **K-means:** Have k centroids, assign each point to cluster to minimize distance. Algorithm: Repeat: Assign points to centroids, recompute centroid.

2 Search

2.1 Lecture 5: Search I

2.1.1 Tree search

1. Search Problem:
   (a) s_{start}: starting state
   (b) Actions(s): possible actions
   (c) Cost(s, a): action cost
   (d) Succ(s, a): successor

(e) IsEnd(s): reached end state?

2. Various algorithms with depth D and breadth b:

   (a) Backtracking search: tries all paths. Memory: $O(D)$, Time: $O(b^D)$.

   (b) Depth-first search: Assume zero action costs. Memory: $O(D)$, Time: $O(b^D)$.

   (c) Breadth-first search: Assume action costs, Cost(s,a) = c. Let d be the number of actions for the solution. $d \leq D$. Memory: $O(b^d)$, Time: $O(b^d)$.

   (d) DFS with iterative deepening: Modify DFS to stop at a maximum depth. Call DFS for maximum depths $1, 2, \cdots$. Check if there is a solution with d actions. Assumes Cost(s,a) = c. Memory: $O(d)$, Time: $O(b^d)$.

2.1.2 Dynamic Programming

1. Assumes acyclicity:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s,a) + \text{FutureCostCost}(\text{Succ}(s,a))] & \text{otherwise} \end{cases}$$

2. A state is a summary of all the past actions sufficient to choose future actions optimally.

2.1.3 Uniform Cost Search

1. Assumes actions are nonnegative

Add s_{start} to frontier (p-queue):
Repeat until frontier is empty:
Remove s with smallest priority p from frontier
If IsEnd(s): return solution
Add s to explored For each action $a \in$Actions(s):'
Get successor $s' \leftarrow \text{Succ}(s, a)$
If s' already explored: continue
Update frontier with s' and priority p+Cost(s, a)

2.2 Lecture 6: Search II

2.2.1 Learning Costs:

1. Structured Perceptron algorithm:

For each action: $w[a] \leftarrow 0$:
For each iteration $t = 1, \cdots, T$:
For each training example $(x, y) \in D_t$:
Compute the minimum cost path y' given w
For each action $a \in y$: $w[a] \leftarrow w[a] - 1$
For each action $a \in y'$: $w[a] \leftarrow w[a] + 1$.

2.2.2 A^* search

1. Instead of exploring in order of PastCost(s), explore in order of PastCost(s) + $h(s)$. Run uniform cost search with: $Cost'(s,a) = Cost(s,a) + h(Succ(s,a)) - h(s)$.
2. Heuristic is consistent if $Cost'(s,a) = Cost(s,a) + h(Succ(s,a)) - h(s) \geq 0$, and $h(s_{end}) = 0$.
3. A^* is correct if h is consistent.
4. Any consistent heuristic satisfies $h(s) \leq$ FutureCost(s). This is called admissibility. In tree search, we only need an admissible heuristic to find the minimum cost path. In graph search problems, we need consistent heuristics.

2.2.3 Relaxation

1. Reduce costs and remove constraints and define our heuristic as the Future Cost on this easier problem.
2. Heuristics defined as $h(s)$ =FutureCost$_{rel}(s)$ for some relaxes problem are consistent.
3. Suppose $h_1(s)$ and $h_2(s)$ are consistent. Then, $h(s) = \max(h_1(s), h_2(s))$ is consistent.

3 Markov Decision Processes

3.1 Lecture 7: Markov Decision Processes I

3.1.1 Markov Decision Processes

1. So far, have assumed that actions deterministically result in a unique successor state.
2. MDP can be represented as a graph. It is defined as:
   (a) States
   (b) $s_{\text{start}} \in$ States: starting state
   (c) Actions(s): possible actions from state s
   (d) $T(s,a,s')$: probability of s' if take action a in state s
   (e) Reward(s,a,s'): reward for the transition (s,a,s')
   (f) IsEnd(s): whether at end of game
   (g) $0 < \gamma \leq 1$: discount factor
3. **Policy:** A policy π is a mapping from each state $s \in$ States to an action $a \in$ Actions(s).

3.1.2 Policy evaluation

1. Following a policy yields a random path. Utility of a policy is the discounted sum of the rewards on the path.
2. **Value** is the expectation of this utility. This is denoted as $V_\pi(s)$.

3. $Q_\pi(s,a)$ is the **Q-value** of a policy which is the expected utility of taking action a from state s, and then following policy π.

4. Thus, we have: $Q_\pi(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_\pi(s')]$

5. **Policy evaluation:** Time: $O(t_{PE}SS')$. Works for a fixed policy π. Takes (MDP, π), outputs V_π.

   Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.
   For iteration $t = 1, \cdots, t_{PE}$:
   For each state s:
   $V_\pi^{(t)}(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[\text{Reward}(s,\pi(s),s') + \gamma V_\pi^{(t-1)}(s')]$

6. **Policy improvement:** Improves π to something slightly better π_{new}. Simple greedy that sets action to highest Q. Takes (MDP, V_π) , outputs π_{new}.

3.1.3 Policy Iteration

Takes MDP, outputs (V_{opt}, π_{opt}).
$\pi \leftarrow$ arbitrary
While π changing:
Policy evaluation to compute V_π
Policy improvement to get new π_{new}
$\pi \leftarrow \pi_{new}$

3.1.4 Value Iteration

1. $V_{opt}(s)$ is the maximum value attained by any policy. V_{opt} is maximum $Q_{opt}(s,a)$. Takes MDP, outputs (V_{opt}, π_{opt}). Hence, value iteration is:

   $V_{opt}^{(0)}(s) \leftarrow 0$
   For iteration $t = 1, \cdots, t_{VI}$:
   For each state s:
   $V_{opt}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} Q_{opt}^{(t-1)}(s,a)$

2. For convergence, we must have $\gamma < 1$ and the MDP is acyclic.

3.2 Lecture 8 Markov Decision Processes II

3.2.1 Reinforcement Learning

3.2.2 Monte Carlo methods

1. Estimate $\hat{T}(s,a,s') = \frac{\#(s,a,s')}{\#(s,a)}$, $\hat{\text{Reward}}(s,a,s')$ = average of r in (s,a,r,s').

2. Issue is that won't see (s,a) if $a \neq \pi(s)$. Need to explore.

3. Decide to cut directly to model-free learning:

$$\hat{Q}_{opt}(s,a) = \sum_{s'} \hat{T}(s,a,s')[\hat{\text{Reward}}(s,a,s') + \gamma \hat{V}_{opt}(s')] \tag{1}$$

4. **Interpolation:** Instead of thinking of averaging as a batch operation, we view it as an iterative procedure. For each (s,a,u):

$$\begin{aligned} \eta &= \frac{1}{1 + \#\text{updates to } (s,a)} \\ \hat{Q}_\pi(s,a) &\leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta u \end{aligned}$$

3.2.3 Bootstrapping methods

1. SARSA: receive (s,a,r,s',a'), $\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta[r + \gamma\hat{Q}_\pi(s',a')]$
2. Q-learning: $Q_{opt}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{opt}(s')]$.
   Algorithm:
   On each (s,a,r,s'): $\hat{Q}_{opt}(s,a) \leftarrow (1-\eta)\hat{Q}_{opt}(s,a) + \eta(r + \gamma\hat{V}_{opt}(s')))$, where
   $\hat{V}_{opt}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{opt}(s',a')$.
3. **Epislon-greedy**: provides a balance between exploitation and exploration. With probability ϵ, take a random action. With probability $1-\epsilon$, take the action with highest $\hat{Q}_{opt}(s,a)$. This becomes the stochastic gradient update rule:

$$\hat{Q}_{opt}(s,a) \leftarrow \hat{Q}_{opt}(s,a) - \eta[\hat{Q}_{opt}(s,a) - (r + \gamma\hat{V}_{opt}(s'))] \tag{2}$$

4. **Function approximation:** Define features $\phi(s,a)$ and weights w such that $\hat{Q}_{opt}(s,a;w) = w \cdot \phi(s,a)$.
5. **Q-learning with function approximation:**

$$w \leftarrow w - \eta[\hat{Q}_{opt}(s,a;w) - (r + \gamma\hat{V}_{opt}(s'))]\phi(s,a) \tag{3}$$

3.2.4 Covering the unknown

4 Game Playing

4.1 Lecture 9: Games I

4.1.1 Games, expectimax

1. Two player zero-sum games:
   (a) s_{start}
   (b) Actions(s)
   (c) Succ(s,a)
   (d) IsEnd(s)
   (e) Utility(s): **agent's** utility for **end state** s. This is only defined at end state.
   (f) Player(s) $\in$ Players: player who controls state s
2. **Expectimax:**

$$V_{\text{max,opp}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{max,opp}}(\text{Succ}(s,a)) & \text{Player(s)} = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}} V_{\text{max,opp}}(\text{Succ}(s,a)) & \text{Player(s)} = \text{opp} \end{cases}$$

4.1.2 Minimax, expectiminimax

1. **Minimax:**

$$V_{\text{max,min}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s,a)) & \text{Player(s)} = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s,a)) & \text{Player(s)} = \text{opp} \end{cases}$$

   (a) $V_{\text{max,min}}(s_{\text{start}}) \geq V_{\text{agent,min}}(s_{\text{start}})$ for all π_{agent}.

   (b) $V_{\text{max,min}}(s_{\text{start}}) \leq V_{\text{max,opp}}(s_{\text{start}})$ for all π_{opp}.

   (c) Note that if an opponent is not playing the adversarial policy, the max policy may not be the best policy.

2. **Expectiminimax:** Introduce a player that follows a known stochastic policy. Hence, for agent and opponent, value is same as minimax, but new player has value same as expectimax.

4.1.3 Evaluation functions

1. **Depth-limited search:**

$$V_{\text{max,min}}(s,d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s,a),d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{max,min}}(\text{Succ}(s,a),d-1) & \text{Player}(s) = \text{opp} \end{cases}$$

2. $\text{Eval}(s;w) = w \cdot \phi(s)$

4.1.4 Alpha-beta pruning

1. Create lower bound a_s and upper bound b_s. We maintain a lower bound (update a_s) for all max nodes s and an upper bound (update b_s) for all min nodes s. If interval of current node does not non-trivially overlap the interval of every one of its ancestors, then we can prune the current node.

4.2 Lecture 10: Games II

4.2.1 TD Learning

$$\begin{aligned} V(s;w) &= w \cdot \phi(s) \\ w &\leftarrow w - \eta[V(s;w) - (r + \gamma V(s';w))]\nabla_w V(s;w) \end{aligned}$$

4.2.2 Simultaneous games

1. Create payoff matrix V whose dimensionality is $|\text{Actions}| \times |\text{Actions}|$.

2. $V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a,b)$

3. **Minimax Theorem:** For every simultaneous 2 player zero-sum game, with a finite number of actions, $\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B)$, where π_A and π_B range over mixed strategies.

4.2.3 Non-zero-sum games

1. **Nash equilibrium:** No player has an incentive to change his/her strategy. There always exists at least one Nash equilibrium.

4.2.4 State-of-the-art

5 Constraint satisfaction problems

5.1 Lecture 11: CSPs I

5.1.1 Factor graphs

1. **Variables:** $X = (X_1, \cdots, X_n)$, where $X_i \in \text{Domain}_i$.
2. **Factors:** $f_1, \cdots, f_m$, with each $f_j(X) \geq 0$.
   (a) **Scope** of a factor f_j is the set of variables f_j depends upon.
   (b) **Arity** of a factor f_j is |scope|.
3. Each solution/assignment has $\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$. Our goal is to find the maximum weight assignment.

5.1.2 Dynamic ordering

1. **Dependent factors:** $D(x, X_i)$ is the set of factors depending on X_i but not on unassigned variables.
2. **Backtracking search:**

   If x is complete assignment, update best and return
   Choose unassigned variable X_i
   Order VALUES Domain_i of chosen X_i
   For each value v in the order:
   $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
   If $\delta = 0$, continue
   Domains' $\leftarrow$ Domains via LOOKAHEAD
   Backtrack($x \cup \{X_i : v\}, w\delta$, Domains')
3. **Forward checking:** Remove incosistent values from the domains of neighboring variables

5.1.3 Arc consistency

1. A variable X_i is **arc consistent** with respect to X_j if for each $x_i \in \text{Domain}_i$, there exists $x_j \in \text{Domain}_j$ such that $f(\{X_i : x_i, X_j : x_j\}) \neq 0$ for all factors f whose scope contains X_i and X_j.

2. EnforceArcConsistency(X_i, X_j): Remove values from Domain$_i$ to make X_i arc consistent with respect to X_j.

   AC-3:
   Add X_j to set.
   While set is non-empty:
   Remove any X_j from set.
   For all neighbors X_i of X_j:
   Enforce arc consistency on X_i w.r.t X_j
   If Domain$_i$ changed, add X_i to set.

5.1.4 Modeling

1. **N-ary constraints:** Pack A_{i-1} and A_i into variable B_i which forms a (pre,post) pair from processing X_i.

5.2 Lecture 12: CSPs II

5.2.1 Beam search: $O(nKb\log(Kb))$

Initialize C = {}
For each $i = 1, \cdots, n$:
Extend $C' \leftarrow \{x \cup \{X_i : v\} : x \in C, v \in \text{Domain}_i\}$
$C \leftarrow$ the K elements of C' with highest weights.

5.2.2 Local search

1. **Iterated conditional modes:** For each variable x, assign x's value to be highest weight. Loop until convergence.

2. **Gibbs sampling:** For each variable x, compute weights and assign x with probability proportional to weights. Loop until convergence.

5.2.3 Conditioning

1. Set one variable to fixed value so we can disconnect the graph. Suppose we had $f(x_1, x_2)$, we replace it with $g(x_1) = f(x_1, B)$, where $x_2 = B$.

2. To condition on a variable $X_i = v$, consider all factors $f_1, \cdots, f_k$ and add $g_1, \cdots, g_k$.

3. A and B are conditionally independent given C if conditioning on C produces a graph in which A and B are independent.

4. **Markov Blanket:** the neighbors of A that are not in A.

5.2.4 Elimination

1. Conditioning considers one value ($X_2 = B$). Elimination considers all possible values of X_2.

2. To eliminate a variable X_i, consider factors $f_1, \cdots, f_k$, Add $f_{new}(x) = \max_{x_i} \prod_{j=1}^{k} f_j(x)$. Scope of f_{new} is MarkovBlanket(X_i).

3. Variable elimination:

   For $i = 1, \cdots, n$:
   Eliminate X_i (produces new factor $f_{new,i}$)
   For $i = n, \cdots, 1$:
   Set X_i to the maximizing value in $f_{new,i}$.

4. Generally want to eliminate variables with the fewest neighbors.

5. **Treewidth:** Maximum arity of any factor created by variable elimination with the best variable ordering.

6 Bayesian Networks

6.1 Lecture 13: Bayesian networks I

6.1.1 Basics

A **Bayesian network** is a DAG that specifies a joint distribution over X as a product of local conditional distributions, one for each node.

$$\mathbb{P}(X_1 = x_1, \cdots, X_n = x_n) = \prod_{i=1}^{n} p(x_i | x_{\text{Parents}(i)}) \tag{4}$$

6.1.2 Independence

1. Thm: Factor graph independence implies probabilistic independence.

2. **Probabilistic Independence:** If A and B are two subsets of random variables. We have $A \perp\!\!\!\perp B$ iff $\mathbb{P}(A = a, B = b) = \mathbb{P}(A = a)\mathbb{P}(B = b)$ for all a, b.

3. Each **factor graph structure** defines a family of different probability distributions. $\mathbb{P}(X = x) \propto$ Weight(x).

4. **Bayesian network independence:** $A \perp\!\!\!\perp B$ if after marginalizing all descendants of A and B, the resulting factor graph has $A \perp\!\!\!\perp B$.

5. **v-structure:** Parents are B and E, A is descendant.

   (a) Parents B and E are conditionally dependent (condition on A).

   (b) Parents B and E are independent (marginalize out A).

6. **inverted v-structure:** Parent H, Children H, I:

   (a) Children H and I are conditionally independent (condition on A)

   (b) Children H and I are dependent (marginalize out A)

6.1.3 Examples

6.2 Lecture 14: Bayesian networks II

6.2.1 Preparation

1. General probabilistic inference strategy. To solve $\mathbb{P}(Q \mid E = e)$
   (a) Remove (marginalize) variables not ancestors of Q or E.
   (b) Convert Bayesian network to factor graph.
   (c) Condition (shade nodes/disconnect) on $E = e$.
   (d) Remove (marginalize) nodes disconnected from Q.
   (e) Run probabilistic inference

6.2.2 Forward-backward

1. From lattice representation, have

$$\begin{aligned} F_i(h_i) &= \sum_{h_{i-1}} F_{i-1}(h_{i-1}) w(h_{i-1}, h_i) \\ B_i(h_i) &= \sum_{h_{i+1}} B_{i+1}(h_{i+1}) w(h_i, h_{i+1}) \\ S_i(h_i) &= F_i(h_i) B_i(h_i) \end{aligned}$$

6.2.3 Gibbs sampling

Algorithm:
Initialize x to a random complete assignment
Loop through $i = 1, \cdots, n$ until convergence:
  Compute weight of $x \cup \{X_i : v\}$ for each v
  Choose $x \cup \{X_i : v\}$ with probability proportional to weight.

6.2.4 Particle filtering

1. Propose:

   Suppose we have a set of particles that approximates the filtering distribution over X_1, X_2. We extend the current partial assignment and sample $X_3 \sim p(x_3|x_2)$.

2. Weight:

   Weight each old particle (x_1, x_2, x_3) as $w(x_1, x_2, x_3) = p(e_3|x_3)$.

3. Resample

   Sample from the distribution K times.

6.3 Lecture 15: Bayesian networks III

6.3.1 Supervised Learning

1. 1 variable: Construct joint probability distribution with probabilities proportional to counts. Normalize by dividing by total count.
2. 2 variable: Construct joint probability distribution with probabilities proportional to counts. Normalize by dividing by total count of each variable.
3. **Parameter Sharing:** Local conditional distributions of different variables use the same parameters. Estimates are more reliable but less expressive.
4. **Learning Algorithm: MLE for Bayesian Networks:**

   Count:
   For each x in training Data:
   For each variable x_i:
   Increment $\text{count}_{d_i}(x_{\text{Parents}(i)}, x_i)$

   Normalize:
   For each d and local assignment $x_{\text{Parents}(i)}$:
   Set $p_d(x_i \mid x_{\text{Parents}(i)}) \propto \text{count}_{d_i}(x_{\text{Parents}(i)}, x_i)$

6.3.2 Laplace smoothing

For each distribution and partial assignment, add λ to $\text{count}_d(x_{\text{Parents}(i)}, x_i)$.

6.3.3 Unsupervised learning with EM

1. Expectation Maximization (EM) Algorithm:

   E-step
   Compute $q(h) = \mathbb{P}(H = h \mid E = e; \theta)$ for each h.
   Create weighted points: (h, e) with weight $q(h)$

   M-step
   Compute maximum likelihood (just count and normalize to get θ.

7 Logic

7.1 Lecture 16: Logic I

1. Ingredients of a logic:
   (a) **Syntax:** Defines a set of valid formulas (Formulas)
   (b) **Semantics:** For each formula, specify a set of models (assignments / configurations of the world)
   (c) **Inference rules:** Given f, what new formulas g can be added without changing semantics ($\frac{f}{g}$).

2. **Model:** Assignment of truth values to propositional symbols.
3. **Interpretation function:** If f is a formula, w is a mode, interpretation function $I(f, w)$ returns (T/F) whether w satisfies f.
4. $M(f)$ is the set of models w for which $I(f, w) = 1$.
5. **Knowledge Base (KB):** a set of formulas representing intersection. I.e. $M(KB) = \cap_{f \in KB} M(f)$. Another interpretation is to $\wedge$ all formulas together and find M of large formula.
6. Adding one formula to KB ($KB \to KB \cup \{f\}$) shrinks the set of models ($M(KB) \to M(KB) \cap M(f)$)
7. KB **entails** f (written KB $\vDash$ f) iff $M(f) \supseteq M(KB)$.
8. **Contradiction:** KB contradicts f iff $M(KB) \cap M(f) = \varnothing$.
9. **Contingency:** f adds non-trivial information to KB. $\varnothing \subsetneq M(KB) \cap M(f) \subsetneq M(KB)$.
10. KB contradicts f if KB entails $\neg f$.
11. **Tell**
    (a) Entailment: Already knew that
    (b) Contradiction: Don't believe that
    (c) Contingent: Learned something new
12. **Ask**
    (a) Entailment: Yes
    (b) Contradiction: No
    (c) Contingent: I don't know
13. **Bayesian Network:** $P(f \mid KB) = \frac{\sum_{w \in M(KB \cup \{f\})} P(W=w)}{\sum_{w \in M(KB)} P(W=w)}$
14. **Satisfiable:** M(KB) $\neq \varnothing$.
15. **Modus ponens inference rule:** (This is a real thing?). For propositional symbols p, q, $\frac{p, p \to q}{q}$. Above line is premises, below line is conclusion.
16. **Inference algorithm:**

    Input: set of inference rules RULES.
    Repeat until no changes to KB:
        Choose set of formulas $f_1, \cdots, f_k \in KB$
        If matching rule $\frac{f_1, \cdots, f_k}{g}$ exists:
            Add g to KB.
17. KB **derives/proves** f (KB $\vdash$ f) iff f eventually gets added to KB.
18. A set of inference rules is **sound** if: $\{f : KB \vdash f\} \subseteq \{f : KB \vDash f\}$

19. A set of inference rules is **complete** if: $\{f : KB \vdash f\} \supseteq \{f : KB \vDash f\}$

20. A **definite clause** has the following form: $(p_1 \wedge \cdots \wedge p_k) \rightarrow q$

21. A **horn clause** is either:

    - $(p_1 \wedge \cdots \wedge p_k) \rightarrow q$
    - $(p_1 \wedge \cdots \wedge p_k) \rightarrow$ false (goal clause). This is also $\neg(p_1 \wedge \cdots \wedge p_k)$

22. Modus Ponens is complete w.r.t horn clauses.

23. Modus Ponens for entailment: I.e. check if $f = p_1 \wedge, \cdots, \wedge p_k$. Add $\neg f$ into KB, run modus ponens. If derive false, $KB \vDash f$.