Contents

These are notes for Fall 2024 iteration of CS 188 at UC Berkeley.

Full textbook

Removed Search, CSPs, and Games.

Chapter 1

MDPs

Markov Decision Processes: Used to solve non-deterministic search problems.

Note: With a negative-only reward, agent would want to exit ASAP.

We initialize values to 0.

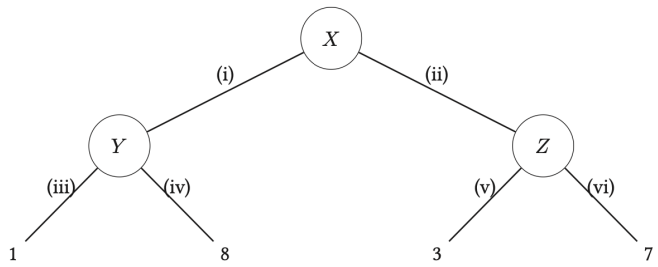


Figure 1.1: For example, MDP table of a Robotron actions can include only initial state X, since Robotron can't choose actions from Y or Z.

Note: Terminal states have a value of 0.

Chapter 2

RL

1 Model-based learning

Do a bunch of simulations to learn empirical transitions $\hat{T}(s, a, s')$ and rewards.

Space: $|S| \times |A|$.

Collect episodes; count successful transitions to s' (max likelihood estimate!).

2 Model-free learning

Estimate directly values or Q-values.

2.1 On-policy learning

Direct evaluation

Space: $|S|$, evaluate a policy through a simulation to get $V^\pi(s)$ (bad, since long to converge).

$$V^*(s) = \frac{1}{n} \sum_{i=1}^{n} V^*(s'_i) \tag{2.1}$$

where n is the number of times we visited that state s; s'_i is the state we transitioned from s (may be different).

Temporal Difference Learning

Space: $|S|$, makes more recent samples more important (past does not matter anyway).

Note: Samples depend on estimated values of other states!

$$\begin{aligned} \text{sample} &= R(s, \pi(s), s') + \gamma V^\pi(s') \\ V^\pi_{k+1}(s) &\leftarrow (1 - \alpha) V^\pi_k(s) + \alpha \cdot \text{sample}_k \end{aligned} \tag{2.2}$$

where $\alpha \in [0, 1]$ is a learning rate.

Converges in fewer states than direct evaluation. Only evaluates current policy; doesn't learn optimal value/policy.

2.2 Off-policy learning

Q-Learning

Space: $|S| \times |A|$.

$$\begin{aligned} \text{sample} &= R(s,a,s') + \gamma \cdot \max_{a'} Q(s',a') \\ V_{k+1}^{\pi}(s) &\leftarrow (1-\alpha) V_k^{\pi}(s) + \alpha \cdot \text{sample} \end{aligned} \tag{2.3}$$

Produces a deterministic, optimal policy π^* given infinite time.

Off-policy learning, which learns optimal policy even taking *suboptimal* actions.

Approximate Q-Learning

Feature vectors $f_i(s,a)$ represent state-action pairs, and we want to learn weight vector w.

$$\begin{aligned} \text{difference} &= [R(s,a,s') + \gamma \max_{a'} Q(s',a')] - Q(s,a) \\ w_i &\leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s,a) \end{aligned} \tag{2.4}$$

For example, characterize actions as fast/long roller coaster, but no features distinguising states.

2.3 Exploration vs exploitation

Exploration: Add some type of randomness to actions.

- **ϵ-greedy policy:** Choose a random action with probabilty $\epsilon \in [0,1]$. Else, follow established policy (exploit). For example: $P(\text{WM}) = (1-\epsilon) + 1/5\epsilon$, $P(\text{all other states}) = 1/5\epsilon$.
- **Exploration functions:** Modified Q-value iteration uses exploration function $f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$, where $N(s,a)$ is the number of times we've taken action a from state s; k is a hyperparameter.

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

  Prefers less visited state-action pairs.

Chapter 3

Bayes Nets

Note: A cycle makes Bayes Net invalid!

1 Structure

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | parents(X_i))$$

1. Each node is conditionally independent of its non-descendants given its parents.
2. Each node is conditionally independent of all other variables given its Markov blanket (includes parents, children, and spouses of the node).

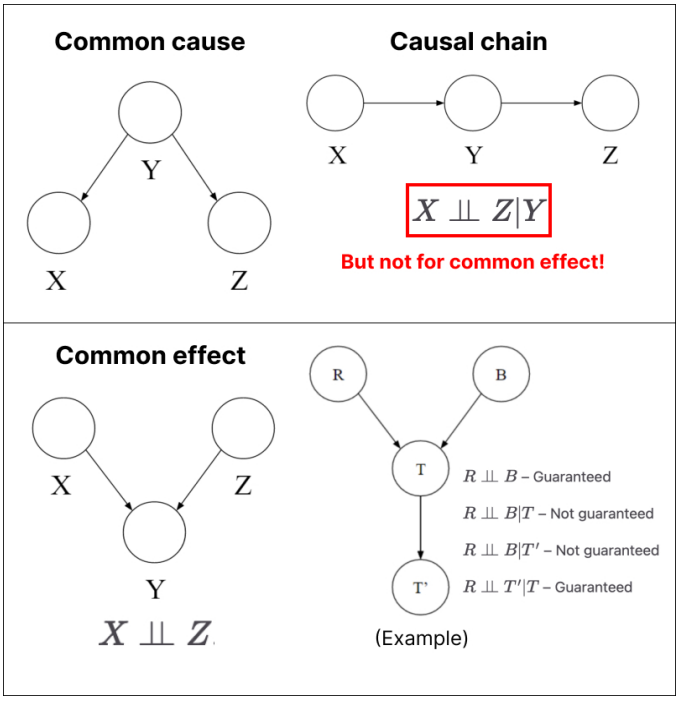


Figure 3.1: `Active triplets.`

2 D-separation

Algorithm:

- Marry parents of nodes. Make edges undirected. Then, remove variables that were conditioned on.
- Disconnected variables are independent given these variables.

3 Inference by Enumeration (IBE)

Compute probability distribution $P(Q_1 \dots Q_m | e_1 \dots e_n)$.

1. Collect all rows consistent with the observed evidence variables.
2. Sum out (marginalize) all the hidden variables.
3. Normalize the table so that it is a probability distribution (i.e. values sum to 1).

Note: Requires joining full CPTs (conditional probability table)!

4 Variable Elimination (VE)

Factor out random variables. Ordering sometimes matters.

Factor is an unnormalized probability, which is proportional to actual probability (but doesn't sum to 1 like a probability distribution should).

Size of the factor = number of entries in CPT. For example, size of $f(S, A, M)$ with three binary variables is 2^3.

Note: VE can use smaller factors and reduce computations (can't be worse than IBE).

If a variable doesn't have any descendants, then eliminating that variable doesn't generate a new variable.

5 Approximate Inference

Note: Only prior sampling may use samples contradicting evidence variables. Also, it does not fix evidence variables.

5.1 Prior sampling

Can require a large number of samples in order to perform analysis of very unlikely scenarios.

Algorithm:

- Generate samples
- Then, discard samples inconsistent with evidence
- Caculate probability

- for $i = 1, \ldots, n$ (in topological order):
  - sample x_i from $P(X_i|parents(X_i))$
- return $(x_1, \ldots, x_n)$

5.2 Rejection sampling

Idea: Early reject any sample inconsistent with the evidence. (Inefficient if the evidence is very rare.)

Algorithm:

- input: evidence $e_1, \ldots, e_k$
- for $i = 1, \ldots, n$:
  - sample x_i from $P(X_i|parents(X_i))$
  - if x_i is inconsistent with evidence
    - reject: return, and no sample is generated in this cycle
- return $(x_1, \ldots, x_n)$

5.3 Gibbs sampling

Idea: Use a consecutive sequence of samples. (States = complete assignments to all vars.)

Algorithm:

- Fix evidence variables, randomly set non-evidence variables.
- Generate subsequent states by looping through non-evidence variables: sample new value for chosen variable to generate new sample.

Use Markov blanket to check which variables' distributions are needed for calculations:

$$P(X_i|x_1, \ldots, x_{i-1}, x_{i+1}, x_n) = P(X_i|markov\_blanket(X_i))$$

Note: Considers downstream and upstream evidence!

5.4 Likelihood weighting

Idea: Ensure never generate a bad sample by manually setting all variables equal to evidence in the query.

Algorithm:

- input: evidence $e_1, \ldots, e_k$
- $w = 1.0$
- for $i = 1, \ldots, n$:
  - if X_i is an evidence variable
    - x_i = observed value for X_i
    - set $w = w * P(x_i|parents(X_i))$
  - else
    - sample x_i from $P(X_i|parents(X_i))$
- return $(x_1, \ldots, x_n), w$

All samples are used!

Note: Downstream variables are influenced by upstream variables, but upstream variables are not influenced by downstream variables.

For example, suppose we want to calculate $P(T| + c, +e)$. For the jth sample, we'd perform these steps:

- Set $w_j = 1.0$ and $c = true$ and $e = true$
- For T: This is not an evidence variable, so we sample t_j from $P(T)$
- For C: This is an evidence variable, so we multiply the weight if the sample by $P(+c|t_j)$: $w_j = w_j \cdot P(+c|t_j)$
- For S: Sample s_j from $P(s_j|t_j)$
- For E: Multiply the weight by $P(+e| + c, s_j)$: $w_j = w_j \cdot P(+e| + c, s_j)$

Then, normalize all weights.

Chapter 4

Decision Networks and VPIs

1 Structure

- Chance nodes: similar to Bayes' nets, associated with a probability, ovals
- Action nodes: represent choice from a number of actions, rectangles
- Utility nodes: children of some combination of chance and action nodes, utility determined from parents' values, diamonds

Expected utlity (EU) of taking action a given evidence e with n chance nodes:

$$\boxed{EU(a|e) = \sum_{x_1,\ldots,x_n} P(x_1,\ldots,x_n|e) \cdot U(a, x_1,\ldots,x_n)}$$

Maximum expected utility (MEU): choose an action that maximizes EU:

$$\boxed{MEU(E = e) = \max_a EU(A = a|E = e)}$$

For example, SP24 FQ5:

$EU(\text{Right}|X = \text{Left}) = P(D = \text{Left}|X = \text{Left}) \cdot U(\text{Right}, D = \text{Left}) + P(D = \text{Right}|X = \text{Left}) \cdot U(\text{Right}, D = \text{Right}) = 0.7 \cdot 0 + 0.3 \cdot 100 = 30.$

$MEU(X = \text{Left}) = \max_a EU(a|X = \text{Left}) = EU(\text{Left}|X = \text{Left}) = 0.7 \cdot 100 + 0.3 \cdot 0 = 70.$

2 Value of Perfect Information

VPI mathematically quantifies how much the amount of agent's maximum expected utility is expected to increase if it observes some new evidence E', given the current evidence e:

$$\boxed{VPI(E'|e) = MEU(e, E') - MEU(e)}$$

$$
\begin{aligned}
MEU(e) &= \max_a \sum_s P(s|e) \cdot U(s,a) && \text{current MEU with evidence } e \\
MEU(e,e') &= \max_a \sum_{s'} P(s'|e,e') \cdot U(s',a) && \text{addiitonal evidence } e' \\
\boxed{MEU(e,E') &= \sum_{e'} P(e'|e) \cdot MEU(e,e')} && \text{don't know } e' \rightarrow \text{replace by r.v. } E'
\end{aligned} \tag{4.1}
$$

Properties:

1. Nonnegativity: observing new evidence can only help you (or be irrelevant, i.e. 0 value).
   $\boxed{\forall E',\ VPI(e,E') \geq 0}$
2. Nonnaditivity: observing evidence E_j might affect how much we care about E_k.
   $\boxed{VPI(E_j,E_k|e) \neq VPI(E_j|e) + VPI(E_k|e) \text{ in general}}$
3. Order independence: order of observing evidence does not matter.
   $\boxed{VPI(E_j,E_k|e) = VPI(E_j|e) + VPI(E_k|e,E_j) = VPI(E_k|e) + VPI(E_j|e,E_k)}$

Since $MEU(\varnothing) = \max_a EU(a)$ has no evidence, we need only distribution of $P(X_n)$; no observation updates.

From the tables in the decision network, we can derive $P(D|X = \text{Left}, Y, Z)$, shown below.

	D	Y	Z	$P(D\|X = \text{Left}, Y, Z)$
(i)	Left	Left	Left	0.927
(ii)	Right	Left	Left	0.073
(iii)	Left	Left	Right	0.7
(iv)	Right	Left	Right	0.3
(v)	Left	Right	Left	0.7
(vi)	Right	Right	Left	0.3
(vii)	Left	Right	Right	0.3
(viii)	Right	Right	Right	0.7

(d) [3 pts] What is VPI($Y, Z|X = \text{Left}$), the value of learning the other two sensor readings, rounded to the nearest integer?
Hint: Our solution uses the probabilities in rows (i), (iii), (v), and (viii) in the table above, along with values from the utilities table.

16

5 or 6

For each instantiation of evidence, we can get the MEU by multiplying $P(D|X, Y, Z)$ from the table (the probability the prize is behind the door we picked), by 100 (the utility of picking the prize). This gives us the four values of:

92.7 (pick Left when all three sensors say Left, correct 0.927 of the time)

70 (pick Left when X and Y say Left, but Z says Right, correct 0.7 of the time)

70 (pick Left when X and Z say Left, but Y says Right, correct 0.7 of the time)

70 (pick Right when X says Left, but Y and Z say Right, correct 0.7 of the time)

We can average this MEU values (as mentioned in an earlier subpart). The four numbers sum to 302.7. Divide by 4 to get roughly 75.

This tells us that the MEU, given all three sensors, is roughly 75.

From the earlier subparts, we found that the MEU, given only a single sensor, is 70.

The gain in MEU from learning two extra sensor readings is approximately 5, exact is 5.675

Figure 4.1: SP24 FQ5: we take MEU of each evidence variables' combination, and then average across four to get an estimate of 75. Note that we want to take optimal action, which differs given different evidence variables.

Chapter 5

Hidden Markov models

Note: Time-based sequences: observe evidence at a timestep and incorporate it into model.

State variable: random variable encoding belief at a timestep.

Evidence variable: random variable encoding observation at a timestep.

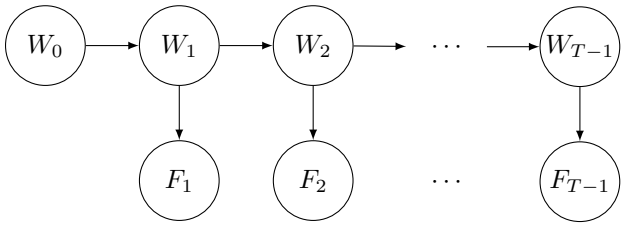
Represent HMM using initial distribution, transition model, and sensor model.

1 Example

Consider weather forecasting:

Weather on day i (state variable) :

Forecast on day i (evidence variable) :



Given initial distribution $P(W_0)$.

$$\begin{aligned} &F_1 \perp W_0 \mid W_1 \\ &\forall i = 2, \ldots, n; \quad W_i \perp \{W_0, \ldots, W_{i-2}, F_1, \ldots, F_{i-1} \mid W_{i-1}\} \\ &\forall i = 2, \ldots, n; \quad F_i \perp \{W_0, \ldots, W_{i-1}, F_1, \ldots, F_{i-1} \mid W_i\} \end{aligned} \tag{5.1}$$

From above, the transition model $P(W_{i+1}|W_i)$ and the sensor model $P(F_i|W_i)$ are stationary.

Define **belief distribution** at time i given evidence $f_1, \ldots, f_i$ observed up to date:

$$B(W_i) = P(W_i|f_1, \ldots, f_i); \quad B'(W_i) = P(W_i|f_1, \ldots, f_{i-1}).$$

Forward algorithm: $\boxed{B(W_{i+1}) \propto P(F_{i+1}|W_{i+1}) \sum_{w_i} P(W_{i+1}|w) B(w_i)}$

- **Time elapse update:** advance model's state by one timestep

$$\boxed{B'(W_{i+1}) = \sum_{w_i} P(W_{i+1}|w_i)B(w_i)}$$

- **Observation update:** incorporate new evidence

$$\boxed{B(W_{i+1}) \propto P(f_{i+1}|W_{i+1})B'(W_{i+1})}$$

2 Forward algorithm

Elapse time:

$$P(X_t|e_{1:(t-1)}) = \sum_{x_{t-1}} P(X_t|x_{t-1})P(x_{t-1}|e_{1:(t-1)})$$

Observe:

$$P(X_t|e_{1:t}) = \frac{P(e_t|X_t)P(X_t|e_{1:(t-1)})}{\sum_{x_t} P(e_t|x_t)P(x_t|e_{1:(t-1)})}$$

Note: Consists of time elapse and observation update. If given no evidence, rely only on time elapse: $P(X_t|X_{t-1})$.

For example, SU24 FQ3: $B(J_{i+1}) = P(C_{j+1}|J_{i+1}) \cdot P(S_{i+1}|J_{i+1}) \cdot P(L_{i+1}|J_{i+1}) \sum_{j_i} P(J_{i+1}|j_i) \cdot P(j_i)$.

3 Particle filtering

Problem: Exact inference is infeasible when domain of vars grows too large.

Idea: Use a set of samples (particles) to represent belief state.

Algorithm:

- store n particles ($n << |X|$)
- prediction: sample state from transition model
  $x_{t+1} \sim P(X_{t+1}|x_t)$
- update: observe e_{t+1} and weight sample based on evidence
  $w = P(e_{t+1}|x_{t+1})$
  - fitting data better means getting higher weight
  - normalize weights to sum to 1 across all particles
- resample: n times, sample with replacement and get new particles
  - avoids tracking weighted samples
- repeat for next timestep

Note: If no evidence is provided, weight of each particle is 1, and resampling is bad, since we could lose particles.

Note: If the particle is drawn upon resampling, the resulting new particle will still be at state t_i.

For example, consider a transition model for our weather scenario using temperature as the time-dependent random variable as follows: for a particular temperature state, you can either stay in the same state (80% of probability) or transition to a state *one degree away* (uniformly split the remaining 20% probability), within the range $[10, 20]$.

Assume we have $n = 10$ particles: $[15, 12, 12, 10, 18, 14, 12, 11, 11, 10]$. (Note that we have 11 possible states.)

To perform a **time elapse update** for the first particle, which is in state $T_i = 15$, we define the transition model:

T_{i+1}	**14**	**15**	**16**
$P(T_{i+1}\|T_i = 15)$	0.1	0.8	0.1

We allocate a distribution range of values based on cumulative transition probabilities:
The range for $T_{i+1} = 14$ is $0 \leq r < 0.1$; for $T_{i+1} = 15$ is $0.1 \leq r < 0.9$; for $T_{i+1} = 16$ is $0.9 \leq r < 1$.

In order to resample our particle in state $T_i = 15$, we simply generate a random number in range $[0, 1)$ and see which range it falls in. Hence if our random number is $r = 0.467$, then the particle at $T_i = 15$ remain in $T_{i+1} = 15$ since $0.1 \leq r < 0.9$. Doing this for remaining nine particles is an **observation update**.

Assuming that sensor model $P(F_i|T_i)$ assigns a correct forecast f_i (here, assume $f_i = t_i$, i.e. temperature stays the same) with probability 80%, while predicting any of the other ten states with uniform probability 2%.

We assign weights to states using the sensor model, then aggregate by states $\in \{10, \ldots, 20\}$. We normalize those weights by states to sum up to 1. Then, we **resample** particles, by generating ten random numbers in $[1, 0)$ again and assigning them into states by ranges of cumulative weights.

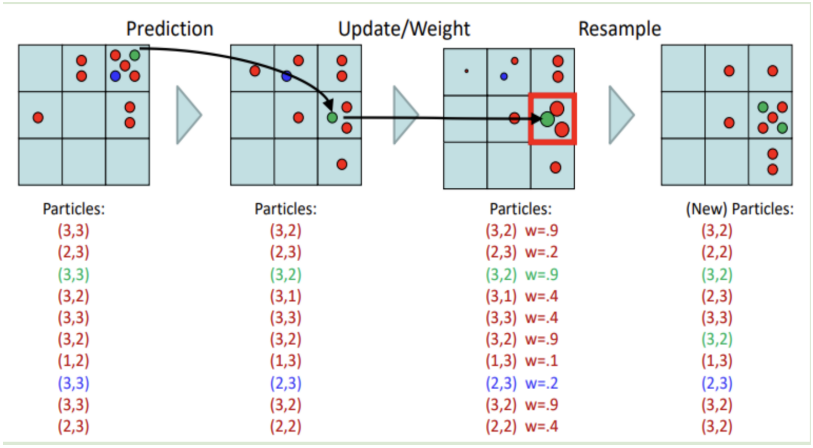


Figure 5.1: Partcile filtering: we move one particle and then update (via time elapse and observation) and resample.

Chapter 6

ML

1 Machine Learning

Core idea: We give machines access to data and they learn for themselves!

Data is often split into training, validation, and test sets:

- training: used to fit the model
- valdiation: used to tune **hyperparameters** (learning rate, model structure, etc.)
- test: used to test the entire model accuracy

Some types of machine learning problems: classification (predict classes), regression (predict numerical values), and clustering (predict groups).

Types of learning: supervised (given labels, e.g. classification) and unsupervised (no labels, e.g. clustering).

2 Naïve Bayes

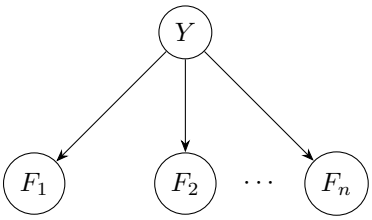
Idea: A model that can predict a label Y given features, where we assume all features are independent given label.

$$\begin{aligned} prediction(f_1, \ldots, f_n) &= \arg\max_y P(Y = y | F_1 = f_1, \ldots, F_n = f_n) \\ &= \arg\max_y P(Y = y, F_1 = f_1, \ldots, F_n = f_n) \\ &= \arg\max_y P(Y = y) \prod_{i=1}^{n} P(F_i = f_i | Y = y) \end{aligned} \tag{6.1}$$

2.1 Bag-of-words Naïve Bayes:

Consider a spam filter, where:

- target label $Y \in \{spam, ham\}$
- features: $F_i \in \{0, 1\}$ is whether "the word at position i in the vocabulary" is present in the email



Provided $P(F_i = f_i | Y = y)$, the probability of whether this word f_i is present in email with label y.

Define probability:

$$\theta = P(F_i = 1 | Y = ham)$$

Likelihood function:

$$\mathcal{L}(\theta) = \prod_{j=1}^{N_h} P(F_i^{(j)} = f_i^{(j)} | Y = ham) = \prod_{j=1}^{N_h} \theta^{f_i^{(j)}} (1 - \theta)^{1 - f_i^{(j)}}$$

Log likelihood function:

$$\log \mathcal{L}(\theta) = \log \theta \sum_{j=1}^{N_h} f_i^{(j)} + \log(1 - \theta) \sum_{j=1}^{N_h} (1 - f_i^{(j)})$$

Maximizer:

$$\frac{\partial \log \mathcal{L}(\theta)}{\partial \theta} = \frac{1}{\theta} \sum_{j=1}^{N_h} f_i^{(j)} - \frac{1}{1 - \theta} \sum_{j=1}^{N_h} (1 - f_i^{(j)}) = 0 \tag{6.2}$$

$$\theta = \frac{1}{N_h} \sum_{j=1}^{N_h} f_i^{(j)}, \text{ where } f_i^{(j)} \in \{0, 1\} \tag{6.3}$$

individual i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
first observation $X_v^{(i)}$	0	0	1	0	1	0	0	1	1	1	0	1	1	0	1	1	1	0	0	0
second observation $X_a^{(i)}$	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
individual's type $Y^{(i)}$	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0

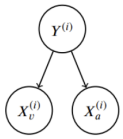
The superscript (i) denotes that the datum is the ith one. Now, the individual with $i = 20$ comes out, and you want to predict the individual's type $Y^{(20)}$ given that you observed $X_v^{(20)} = 1$ and $X_a^{(20)} = 1$.

(a) Assume that the types are independent, and that the observations are independent conditioned on the type. You can model this using naïve Bayes, with $X_v^{(i)}$ and $X_a^{(i)}$ as the features and $Y^{(i)}$ as the labels. Assume the probability distributions take on the following form:

$$P(X_v^{(i)} = x_v | Y^{(i)} = y) = \begin{cases} p_v & \text{if } x_v = y \\ 1 - p_v & \text{if } x_v \neq y \end{cases}$$

$$P(X_a^{(i)} = x_a | Y^{(i)} = y) = \begin{cases} p_a & \text{if } x_a = y \\ 1 - p_a & \text{if } x_a \neq y \end{cases}$$

$$P(Y^{(i)} = 1) = q$$



for $p_v, p_a, q \in [0, 1]$ and $i \in \mathbb{N}$.

(i) What's the maximum likelihood estimate of p_v, p_a and q?

$p_v =$ $\frac{4}{5}$ $p_a =$ $\frac{3}{5}$ $q =$ $\frac{1}{2}$

To estimate q, we count 10 $Y = 1$ and 10 $Y = 0$ in the data. For p_v, we have $p_v = 8/10$ cases where $X_v = 1$ given $Y = 1$ and $1 - p_v = 2/10$ cases where $X_v = 1$ given $Y = 0$. So $p_v = 4/5$. For p_a, we have $p_a = 2/10$ cases where $X_a = 1$ given $Y = 1$ and $1 - p_v = 0/10$ cases where $X_v = 1$ given $Y = 0$. The average of $2/10$ and 1 is $3/5$.

Figure 6.1: `Here, we are given probability of the feature matching the label. Since the label` $y \in \{0, 1\}$`, we sum over two possibilities.`

2.2 Maximum likelihood estimation

Problem: How to estimate CPTs, since we don't actually know them? Parameter estimation with MLE!

Find the probabilties (CPTs) $\theta = P(\cdot)$ that maximize the probability of the observations $P(\text{observations } \mathcal{D} \mid \theta)$:

$$\begin{aligned} P(Y = y) &= MLE(\theta|(F, Y)) = \frac{\text{\# of examples with } Y = y}{\text{total \# of examples}} \\ P(F_i = f_i|Y = y) &= MLE(\theta|(F, Y)) = \frac{\text{\# of examples with } (F_i = f_i, Y = y)}{\text{\# of examples with } Y = y} \end{aligned} \tag{6.4}$$

Essentially, we choose the most likely parameter value given the data, "fitting" probabilties into observations.

2.3 Laplace smoothing

Problem: Chance of overfitting (model doesn't generalize well post-training) with our parameter estimation.

Pretend we saw each of the $|X|$ possible outcomes k more times:

$$P_{LAP,k}(x) = \frac{count(x) + k}{N + k|X|}; \quad P_{LAP,k}(x|y) = \frac{count(x, y) + k}{count(y) + k|X|}.$$

k is a hyperparameter:

- smaller k means your probability estimates follow the training data more closely: $P_{LAP,0} = P_{MLE}(x)$
- larger k means your probability estimates are more uniform: $P_{LAP,\infty} = \frac{1}{|X|}$

3 Perceptron

Note: If data is not linearly separable, will never converge (i.e. reach perfect accuracy).

Note: The separating decision boundary is not guaranteed to be max-margin after the perceptron algorithm terminates.

3.1 Binary perceptron

Idea: Linearly separate data into two classes using a decision boundary (defined by a set of weights).

Binary decision rule: $y^* \in \{+1, -1\}$ with some hyperplane formula.

Algorithm:

1. Initialize all weights to zero: $\mathbf{w} = \mathbf{0}$
2. For each training sample, with features $f(x)$ and true class label $y^* \in \{+1, -1\}$, do:
   - Classify the sample with current weights w and denote predicted label as y:
     $$y = \text{classify}(x) = \begin{cases} +1 & \text{if activation}_w(x) = \mathbf{w}^\top \mathbf{f}(x) > 0, \\ -1 & \text{if activation}_w(x) = \mathbf{w}^\top \mathbf{f}(x) < 0. \end{cases}$$
   - Compare the predicted label y to true label y^*:
     - if correct (i.e. $y = y^*$), no change
     - if wrong, update the weight vector: $\mathbf{w} \leftarrow \mathbf{w} + y^* \cdot \mathbf{f}(x)$
3. If you went through every training sample without having to update the weight vector (all samples predicted correctly), then terminate. Otherwise, repeat step 2.

Note: To find boundaries that don't have to cross the origin, incorporate a "bias" feature that always has value 1 (for example, if the separation line is parallel to x-axis).

3.2 Multiclass perceptron

Multiclass decision rule: Define a weight vector for each class $\mathbf{w_y}$. Then, score (activation) of a class y is $\mathbf{w_y}^\top \mathbf{f}(x)$. Prediction highest score wins: $y = \arg\max_y \mathbf{w_y}^\top \mathbf{f}(x)$.

- if wrong, lower score of wrong answer, raise score of correct answer:
  $\mathbf{w_y} \leftarrow \mathbf{w_y} - \mathbf{f}(x)$, $\mathbf{w_{y^*}} \leftarrow \mathbf{w_{y^*}} + \mathbf{f}(x)$ (do not update other weights)

3.3 Properties

1. Separability: true if some parameters get the training set perfectly correct.
2. Convergence: if the training is separable, perceptron will eventually converge (binary case).
3. Mistake bound: max number of mistakes (binary case) related to the *margin* or degree of separability.

Problems with perceptrons include noise (use averaged perceptron that averages weight vectors over time), mediocre generalization, and overtraining (test/validation accuracy usually rises, then falls).

4 Logistic Regression

Use sigmoid function to increase/decrease probabilities exponentially:

$$\phi(z) = \frac{1}{1+e^{-z}} \in [0,1]$$

(Note that when $z = 0$, the boundary is 0.5. Also, $\phi'(z) = \phi(z) \cdot (1 - \phi(z))$.)

Note: Decision boundary is always linear for logistic regression.

4.1 Binary logistic regression

Binary probabilistic decision rule: If the value of the output is greater than 0.5, classify as +1.

$$\begin{aligned} P(y = +1 \mid \mathbf{x}, \mathbf{w}) &= \frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}}} = h_{\mathbf{w}}(\mathbf{x}) \quad \text{where } \mathbf{x} = \mathbf{f}(x) \\ P(y = -1 \mid \mathbf{x}, \mathbf{w}) &= 1 - h_{\mathbf{w}}(\mathbf{x}) \end{aligned} \tag{6.5}$$

Use L2 loss and gradient descent to estimate $\mathbf{w}$: $Loss(\mathbf{w}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2$.

4.2 Multiclass logistic regression

Multiclass probabilistic decision rule: Use softmax function, given K classes:

$$P(y = i \mid \mathbf{x}, \mathbf{w}) = \frac{e^{\mathbf{w}_i^\top \mathbf{x}}}{\sum_{k=1}^{K} e^{\mathbf{w}_k^\top \mathbf{x}}}$$

Use cross-entropy loss: $Loss(\mathbf{w}_j) = -[y \ln h_{\mathbf{w}_j}(\mathbf{x}) + (1-y)\ln(1 - h_{\mathbf{w}_j}(x))]$.

Note that $\frac{\partial Loss}{\partial \mathbf{w}_j} = \mathbf{x}_j(h_{\mathbf{w}}(\mathbf{x}) - y)$, since $\frac{\partial h_{\mathbf{w}}(\mathbf{x})}{\partial \mathbf{w}_j} = \frac{\partial \phi(-\mathbf{w}^\top \mathbf{x})}{\partial \mathbf{w}_j} = \mathbf{x}_j \phi(-\mathbf{w}^\top \mathbf{x})(1 - \phi(-\mathbf{w}^\top \mathbf{x}))$.

5 Optimization

Hill Climbing: Iterative numerical optimization.

- start wherever
- repeat: move to the best neighboring state
- if no neighbors better than current, quit

For continuous optimization, consider following the gradient (the direction of steepest increase).

Gradient Ascent: Perform an update in the direction of the uphill for each coordinate (used if the objective is a function that we are trying to maximize).

- randomly initialize $\mathbf{w}$
- while $\mathbf{w}$ not converged, do: $w \leftarrow \mathbf{w} + \alpha \cdot \nabla_{\mathbf{w}} f(\mathbf{w})$

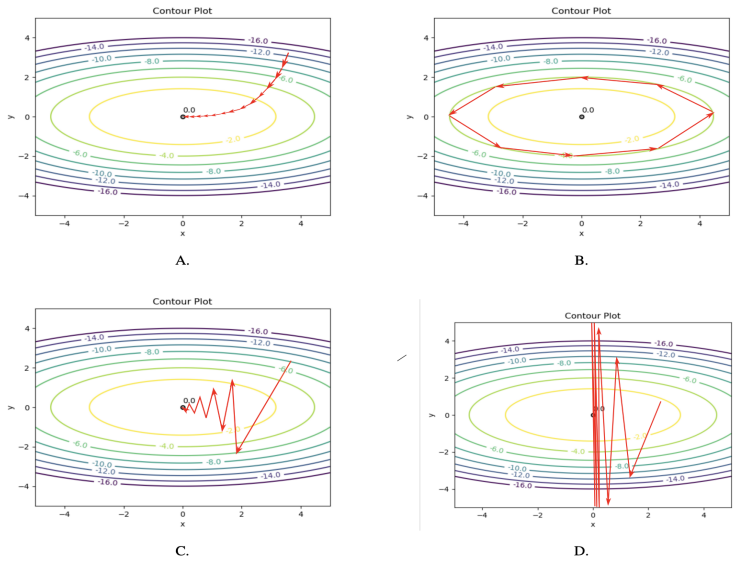


Figure 6.2: The values in the contour lines denote the objective function value we're trying to maximize. Options (A), (C), (D) are valid, while (B) is impossible, since gradients must be perpendicular to the contour (the steepest direction). (C) has a larger learning rate than (A). (D) is large enough to make method diverge.

Note: If set learning rate $\alpha = 1$ when we started search not at the optimal value, will never converge.

Gradient Descent:

- randomly initialize $\mathbf{w}$
- while $\mathbf{w}$ not converged, do: $w \leftarrow \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} f(\mathbf{w})$

Preventing overfitting in optimization: early stopping, weight regularization (e.g. Lasso).

6 NNs

Motivation: Most problems are nonlinear.

Common neural network class is the **multilayer perceptron (MLP)** algorithm:

- instead of having binary $\{+1, -1\}$ choices, after each node, we use a nonlinear activation function, e.g. ReLU
- nodes still dot product their inputs with their own weight vectors: $h^{(l)} = \phi(W^{(l)}h^{(l-1)} + b^{(l)})$
- use gradient descent to update all weights (backpropagation)

Universal Function Approximation Theorem: Any continuous function can be approximated by a two-layer NN (with sufficient amount of neurons).

Training neural networks:

- Set the weights to some initial values
- Input training data, run **forward pass** to generate values at all nodes, calculate loss function on final output
- Run **backward pass**, calculating the gradient of loss function with respect to each of the weights
- Use gradient descent to update all the weights
- Repeat with more data!

Note: Number of points of slope change = number of ReLU activations (given a NN with affine layers).

Note: A 1-dimensional affine layer can only scale, shift, and flip graph. But 2-dimensional affine layer can rotate, too.