YZM2011

Introduction to Machine Learning

Week 14: Bayesian Networks and Hidden Markov Models

Instructor: Ekrem Çetinkaya

Date: 09.06.2026

Today's Agenda

Probabilistic Graphical Models

- Why graphs for probability?
- Bayesian Network structure
- Factorization and CPTs
- Local Markov property
- Markov Blanket

D-Separation

- Three connection types
- Explaining Away (collider)
- D-Separation formal definition
- Reading independence from graphs

Inference Overview

- Types of queries in BNs
- Exact vs approximate inference

Sampling Methods

- Monte Carlo basics
- MCMC and stationary distributions
- Gibbs Sampling

Hidden Markov Models

- HMM structure and parameters
- Three fundamental problems
- Forward-Backward algorithm
- Viterbi algorithm

Recap - Week 13: Ensembles

Last week we covered ensemble methods: Bagging, Random Forest, Boosting, and Stacking. The recurring theme was that **combining many imperfect models** can outperform any single model, either by reducing variance (Bagging) or by reducing bias through sequential correction (Boosting).

All of those methods model $P(y|x)$ directly from data - they are discriminative. This week we take a different approach: instead of predicting labels from features, we build an explicit **probability model over a set of variables**, encoding how they relate to each other through a graph structure.

This shifts the question from *"what label does this input get?"* to *"how do these variables influence each other, and what can we infer when some are observed?"* - which opens the door to reasoning under uncertainty, missing data, and sequences.

Why Probabilistic Graphical Models?

The core problem: if you have N random variables, the full joint distribution $P(X_1, X_2, \ldots, X_N)$ requires $O(2^N)$ parameters to specify exactly. With 100 binary variables that is 10^{30} parameters - neither storable nor computable.

Real systems, however, are not fully coupled. A patient's blood pressure depends on a handful of variables, not on everything else in the world simultaneously. **Conditional independence** - the idea that some variables are irrelevant once you know others - is the key to breaking the exponential curse.

A Probabilistic Graphical Model encodes these independence assumptions explicitly in a graph. Nodes are random variables. Edges encode direct dependencies. Everything not connected is assumed conditionally independent given the right context. This gives a compact, interpretable, and computationally tractable representation.

Probabilistic Graphical Models - Two Families

Directed Models (Bayesian Networks)

Edges are directed arrows - $X \to Y$ means X is a direct cause or influence on Y. The graph must be a Directed Acyclic Graph (DAG): no variable can be its own ancestor. The joint distribution factorizes as a product of each variable's conditional given its parents:

$$P(X_1, \ldots, X_N) = \prod_{i=1}^{N} P(X_i \mid \mathrm{pa}(X_i))$$

BNs naturally express **causal structure** and are the focus of this lecture.

Undirected Models (Markov Random Fields)

Edges are undirected lines expressing symmetric dependencies. The joint factorizes over **cliques** (fully connected subgraphs):

$$P(X_1, \ldots, X_N) = \frac{1}{Z} \prod_{C} \psi_C(X_C)$$

where Z is a normalization constant. MRFs are powerful for image processing and spatial models (e.g., pixel grids where every pixel influences its neighbors symmetrically). We will not cover them in depth today.

Bayesian Networks - Definition

A **Bayesian Network** is a pair (G, θ) where G is a DAG and θ is a set of conditional probability tables (CPTs). Each node X_i has a CPT defining $P(X_i \mid \text{pa}(X_i))$ - its probability distribution given every possible combination of its parents' values.

The joint distribution over all variables is:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid \text{pa}(X_i))$$

This is not an approximation - it is exact, as long as the graph captures the true dependencies. The savings come from the fact that each node depends only on its **local parents**, not on all other variables. A variable with k binary parents needs 2^k parameters instead of 2^{N-1}.

The Alarm Network - A Running Example

Consider a home security system with five variables: **B**urglary, **E**arthquake, **A**larm, **J**ohn calls, **M**ary calls. The causal story is: a burglary or earthquake can trigger the alarm; if the alarm sounds, John and Mary may call to report it.

The DAG encodes exactly this story: $B \to A$, $E \to A$, $A \to J$, $A \to M$. The joint distribution is:

$$P(B, E, A, J, M) = P(B)\, P(E)\, P(A|B, E)\, P(J|A)\, P(M|A)$$

Parameter savings: The naive joint requires $2^5 - 1 = 31$ numbers. The BN requires only $1 + 1 + 4 + 2 + 2 = 10$ - a 68% reduction, achieved purely through exploiting the conditional independence structure encoded in the graph.

Conditional Probability Tables (CPTs)

Prior probabilities (no parents):

B	P(B)		E	P(E)
T	0.001		T	0.002
F	0.999		F	0.998

P(A | B, E) - has 4 rows (2×2 parents):

B	E	P(A=T)
T	T	0.95
T	F	0.94
F	T	0.29
F	F	0.001

P(J | A) and **P(M | A)**:

A	P(J=T)	P(M=T)
T	0.90	0.70
F	0.05	0.01

Each row of a CPT must sum to 1. The CPT for A shows the alarm is almost certain if there is both a burglary and an earthquake (0.95), nearly as likely for burglary alone (0.94), occasionally trips in earthquakes (0.29), and very rarely fires without cause (0.001).

Given the alarm sounded, John calls 90% of the time; Mary, 70%. Without the alarm, both rarely call (5% and 1%).

Naive Bayes Is a Bayesian Network

In Week 9 we classified digits by assuming features were conditionally independent given the class: $X_i \perp X_j \mid C$. That assumption is exactly a Bayesian Network structure - a star graph with the class at the center and every feature as a leaf.

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

This is the BN factorization with $\text{pa}(X_i) = \{C\}$ for every feature and $\text{pa}(C) = \emptyset$. The BN framework tells us exactly what the Naive Bayes assumption means structurally: all paths between any two features X_i and X_j pass through the class node C, which is a **fork connection** - so $X_i \perp X_j \mid C$ follows directly from d-separation. Naive Bayes is one of the simplest and most useful BNs in existence.

Parameter count advantage: The full joint over C and n binary features needs 2^{n+1} parameters. Naive Bayes needs only $1 + 2n$ - because each feature's CPT is just $P(X_i = 1 \mid C = c)$, two numbers per feature. For 64-pixel digits: full joint needs $2^{65} \approx 10^{19}$; Naive Bayes needs 129.

Local Markov Property and Markov Blanket

The **Local Markov Property** states: each node is conditionally independent of all its non-descendants given its parents.

$$X_i \perp \text{NonDesc}(X_i) \mid \text{Pa}(X_i)$$

In the Alarm network: once we know whether the alarm sounded (A), John's calling behavior (J) is independent of the burglary (B), the earthquake (E), and Mary's behavior (M). Knowing the alarm fully "screens off" everything upstream and sideways from J.

The **Markov Blanket** of a node X is the minimal set of nodes that renders X independent of the entire rest of the network:

$$\text{MB}(X) = \text{Pa}(X) \cup \text{Ch}(X) \cup \text{Pa}(\text{Ch}(X))$$

It consists of X's parents, X's children, and the other parents of X's children (co-parents). For A: $\text{MB}(A) = \{B, E, J, M\}$ - observe all four and A is fully determined, regardless of anything else. The Markov Blanket is the minimal sufficient neighborhood for updating or sampling any single node.

D-Separation - Three Connection Types

D-Separation is the graphical test for reading conditional independence from the BN structure. There are three types of connections between any three nodes X, Y, Z:

Chain (serial): $X \to Y \to Z$

Information flows from X to Z through Y. If Y is observed, the path is **blocked**: $X \perp Z \mid Y$. Example: Fire → Smoke → Alarm. Once we know whether there is smoke, the alarm state tells us nothing new about the fire.

Fork (diverging): $X \leftarrow Y \to Z$

Y is a common cause. X and Z are marginally dependent (both caused by Y), but conditionally independent once Y is observed: $X \perp Z \mid Y$. Example: Flu → Fever, Flu → Cough. Once flu status is known, fever gives no extra information about cough.

Collider (converging): $X \to Y \leftarrow Z$

This is the counterintuitive case. X and Z are **marginally independent** - they have no common cause. But observing Y (or any descendant of Y) **opens** the path and creates dependence: $X \not\perp Z \mid Y$.

Why? If you know the alarm sounded ($Y = T$) and you learn there was no burglary ($X = F$), the probability of an earthquake (Z) jumps. One cause being ruled out makes the other more likely. This is called **explaining away**.

Collider rule summary: a collider **blocks** by default; observing it (or its descendants) **unblocks** it.

D-Separation - Formal Definition

A path between X and Z is **blocked** by a set $\mathcal{Z}$ if there exists a node W on the path such that:

- W is a chain or fork node **and** $W \in \mathcal{Z}$ (observed), or
- W is a collider **and** $W \notin \mathcal{Z}$ **and** no descendant of W is in $\mathcal{Z}$

X and Z are **d-separated** by $\mathcal{Z}$ (written $X \perp_d Z \mid \mathcal{Z}$) if **every** undirected path between them is blocked. D-separation implies probabilistic independence:

$$X \perp_d Z \mid \mathcal{Z} \;\Rightarrow\; X \perp_P Z \mid \mathcal{Z}$$

Alarm network table - d-separation results:

Query	Given $\mathcal{Z}$	Result	Reason
$B \perp E$	$\emptyset$	✓	Collider A blocks
$B \perp E$	$\{A\}$	×	Collider A opened
$J \perp M$	$\emptyset$	×	Fork A not observed
$J \perp M$	$\{A\}$	✓	Fork A blocks

Inference in Bayesian Networks - Overview

Once a BN is built, it supports several types of queries:

Marginal inference - $P(X_i)$: what is the marginal distribution of a single variable, integrating out all others? Useful for unconditional risk estimates.

Conditional inference - $P(X_q \mid X_e = e)$: given observed evidence e, what is the distribution of query variable X_q? This is the most common query: *"Given John called, what is the probability of a burglary?"*

MAP (Maximum a Posteriori) - $\arg\max_{X_q} P(X_q \mid X_e = e)$: the most probable single value for the query variable given evidence.

Exact inference methods (Variable Elimination, Belief Propagation) give the correct answer but are NP-hard in general - tractable only for small networks or special structures (e.g., trees). **Approximate inference** via sampling scales to large networks at the cost of statistical error. The main sampling approaches are Monte Carlo methods, which we turn to next.

Worked Inference Example - P(Burglary | JohnCalls=T)

By Bayes' rule: $P(B \mid J{=}T) \propto P(B) \cdot P(J{=}T \mid B)$. We expand $P(J{=}T \mid B)$ by summing out E and A:

$$P(J{=}T \mid B) = \sum_{E} \sum_{A} P(E)\, P(A \mid B, E)\, P(J{=}T \mid A)$$

Using CPT values from the alarm network ($P(B{=}T) = 0.001$, $P(E{=}T) = 0.002$):

B	E	A	$P(E)$	$P(A \mid B, E)$	$P(J{=}T \mid A)$	**Joint term**
T	T	T	.002	.95	.90	.002×.95×.90 ≈ .00171
T	T	F	.002	.05	.05	.002×.05×.05 ≈ .000005
T	F	T	.998	.94	.90	.998×.94×.90 ≈ .844
T	F	F	.998	.06	.05	.998×.06×.05 ≈ .003

$P(J{=}T \mid B{=}T) \approx 0.849$, so $P(B{=}T, J{=}T) \approx 0.001 \times 0.849 = 0.000849$.

By analogous computation $P(J{=}T) \approx 0.0521$, giving:

$$P(B{=}T \mid J{=}T) = \frac{0.000849}{0.0521} \approx \mathbf{0.016}$$

John's call raises the burglary probability from the prior of 0.1% to about 1.6% - a 16× lift, but still low because John calls even without a burglary.

Sampling Methods - The Monte Carlo Idea

Many inference problems reduce to computing an expectation:

$$\mathbb{E}_p[f(z)] = \int f(z)\, p(z)\, dz$$

When $p(z)$ is too complex to integrate analytically (high-dimensional, unnormalized), Monte Carlo methods estimate this integral by drawing samples $z^{(1)}, \ldots, z^{(L)} \sim p(z)$:

$$\hat{f} = \frac{1}{L} \sum_{l=1}^{L} f(z^{(l)}) \quad \xrightarrow{L \to \infty} \quad \mathbb{E}_p[f(z)]$$

The estimate is unbiased and its variance shrinks as $1/L$ regardless of dimension - a massive improvement over grid-based quadrature, which degrades exponentially in dimension. The practical challenge is that we usually cannot sample from $p(z)$ directly. Two broad strategies exist: **rejection/importance sampling** (which draw independent samples but become inefficient in high dimensions) and **Markov Chain Monte Carlo** (which draws correlated but asymptotically correct samples).

Markov Chain Monte Carlo (MCMC)

Key idea: Instead of sampling $p(z)$ directly, construct a Markov chain whose **stationary distribution** is $p(z)$. Run the chain long enough and the current state is approximately a sample from p.

A transition kernel $T(z' \mid z)$ leaves p invariant if the **detailed balance** condition holds:

$$p(z)\,T(z' \mid z) = p(z')\,T(z \mid z')$$

In practice: discard the first B samples (burn-in, while the chain converges from its initial state) and optionally thin by taking every k-th sample to reduce autocorrelation. The key diagnostic is the **trace plot**: a well-mixing chain looks like white noise; a stuck chain drifts or stays flat for long stretches.

MCMC methods including Metropolis-Hastings (propose a move, accept/reject based on the ratio $p(z')/p(z)$) and Gibbs Sampling are the workhorses of Bayesian inference for large networks.

Gibbs Sampling

Gibbs Sampling is the canonical MCMC method for models with many variables. It sidesteps the need to design a good global proposal by sampling **one variable at a time**, conditioned on all current values of the others.

```
Algorithm: Gibbs Sampling
──────────────────────────────────────
Initialize z = (z₁, z₂, ..., zₙ) randomly

For each iteration:
    For i = 1 to n:
        Sample zᵢ ~ p(zᵢ | z₁, ..., zᵢ₋₁, zᵢ₊₁, ..., zₙ)
        (update zᵢ in place)

Collect z after burn-in
──────────────────────────────────────
```

In a Bayesian Network, $p(z_i \mid z_{-i})$ is the conditional given all other nodes - and thanks to the Markov Blanket property, this reduces to $p(z_i \mid \mathrm{MB}(z_i))$. Only the Markov Blanket neighbors need to be queried, making each step local and cheap. Gibbs Sampling is easy to implement, always accepts (no rejection step), and mixes well when variables are not too strongly correlated.

Hidden Markov Models

Everything we have covered so far assumes the variables form a static network - no time, no sequence. Hidden Markov Models extend the graphical model framework to **sequential data**, where observations arrive over time and we believe there is an underlying discrete state driving them.

The classic examples: in speech recognition, the acoustic signal (observable) is generated by a sequence of phonemes (hidden); in finance, stock prices (observable) are driven by market regimes such as bull or bear (hidden); in biology, DNA sequences (observable) reveal coding vs non-coding regions (hidden). In each case, the hidden state evolves over time according to a Markov chain, and each observation is generated independently from the current hidden state.

Running example: a weather HMM where the hidden state is Sunny or Rainy, and the observation is whether someone carries an umbrella. We observe the umbrella sequence; we want to infer the weather sequence.

HMM - Graphical Structure

The HMM is a directed graphical model over two parallel chains - one hidden, one observed:

$$\begin{array}{ccccccc} z_1 & \to & z_2 & \to & z_3 & \to \cdots \to & z_T \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ x_1 & & x_2 & & x_3 & \cdots & x_T \end{array}$$

Two independence assumptions follow directly from the graph structure. The **Markov property**: $p(z_t \mid z_{1:t-1}) = p(z_t \mid z_{t-1})$ - each hidden state depends only on the immediately preceding one. **Conditional independence of observations**: $p(x_t \mid z_{1:T}, x_{1:T\setminus t}) = p(x_t \mid z_t)$ - each observation depends only on the current hidden state, not on any other observation or hidden state.

The joint distribution factors as:

$$p(X, Z \mid \lambda) = p(z_1) \prod_{t=2}^{T} p(z_t \mid z_{t-1}) \prod_{t=1}^{T} p(x_t \mid z_t)$$

HMM - Parameters

An HMM with K hidden states is fully specified by three parameter sets, collectively written $\lambda = (\pi, A, B)$:

Initial distribution π: $\pi_i = P(z_1 = i)$, a vector of length K summing to 1. Which state the chain starts in.

Transition matrix A: $A_{ij} = P(z_t = j \mid z_{t-1} = i)$, a $K \times K$ row-stochastic matrix. How the hidden state evolves over time.

Emission distribution B: $B_i(x) = P(x_t = x \mid z_t = i)$, the probability of each observable outcome from each hidden state.

Weather example: $K = 2$ states (Sunny, Rainy), observations are umbrella (0/1). With $\pi = [0.6,\ 0.4]^\top$, the chain starts more likely Sunny. The transition matrix $A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$ means sunny days tend to stay sunny (70%) and rainy days tend to stay rainy (60%). Emission: on a sunny day umbrella probability is 10%; on a rainy day it is 80%.

Three Fundamental Problems in HMMs

Every HMM application reduces to one or more of three problems:

1. Evaluation (Likelihood): Given the model λ and an observation sequence $X = x_1, \ldots, x_T$, compute $P(X \mid \lambda)$. Used for model comparison: which HMM best explains this sequence? Naïvely summing over all K^T hidden state sequences is exponential - the **Forward algorithm** solves it in $O(K^2 T)$.

2. Decoding (Most likely path): Given λ and X, find the most probable hidden state sequence $Z^* = \arg\max_Z P(Z \mid X, \lambda)$. This is the answer to "what was the most likely weather sequence given these umbrella observations?" The **Viterbi algorithm** solves it in $O(K^2 T)$ via dynamic programming.

3. Learning (Parameter estimation): Given observation sequences but unknown λ, find $\lambda^* = \arg\max_\lambda P(X \mid \lambda)$. The **Baum-Welch algorithm** (a special case of EM) solves this iteratively, but we will not cover it in detail today - it builds directly on the Forward-Backward outputs.

Forward Algorithm - Efficient Likelihood

The naive likelihood $P(X \mid \lambda) = \sum_Z P(X, Z \mid \lambda)$ sums K^T terms. The Forward algorithm avoids this by computing **forward variables** recursively:

$$\alpha_t(i) = P(x_1, x_2, \ldots, x_t,\ z_t = i \mid \lambda)$$

$\alpha_t(i)$ is the probability of having seen observations $x_1 \ldots x_t$ **and** being in hidden state i at time t. It is computed by the recurrences:

Initialization: $\alpha_1(i) = \pi_i \cdot B_i(x_1)$

Recursion: $\alpha_{t+1}(j) = B_j(x_{t+1}) \sum_{i=1}^{K} \alpha_t(i)\, A_{ij}$

Termination: $P(X \mid \lambda) = \sum_{i=1}^{K} \alpha_T(i)$

Each $\alpha_{t+1}(j)$ collects probability mass from all states at time t (weighted by transition) and multiplies by the emission at $t+1$. The grid of α values has $K \times T$ cells, each costing $O(K)$ to compute - total $O(K^2 T)$.

Backward Algorithm and State Posteriors

The **backward variable** captures the other direction:

$$\beta_t(i) = P(x_{t+1}, x_{t+2}, \ldots, x_T \mid z_t = i, \lambda)$$

It is the probability of the future observations given that we are in state i at time t. Initialized as $\beta_T(i) = 1$ and computed backward:

$$\beta_t(i) = \sum_{j=1}^{K} A_{ij}\, B_j(x_{t+1})\, \beta_{t+1}(j)$$

Combining forward and backward gives the **state posterior** - the probability of being in state i at time t given the full observation sequence:

$$\gamma_t(i) = P(z_t = i \mid X, \lambda) = \frac{\alpha_t(i)\, \beta_t(i)}{\sum_{j=1}^{K} \alpha_t(j)\, \beta_t(j)}$$

This is the answer to: *"At time t, how confident are we about each hidden state?"* Summing $\gamma_t(i)$ over t gives the expected number of times state i is visited - the sufficient statistic for the Baum-Welch M-step. Forward-Backward is exactly Belief Propagation on a chain graph, which is why it runs in $O(K^2 T)$ rather than the exponential naïve cost.

Forward Algorithm - Python

```python
import numpy as np

def forward(X, pi, A, B):
    T, K = len(X), len(pi)
    alpha = np.zeros((T, K))
    # Initialization
    alpha[0] = pi * B[:, X[0]]
    alpha[0] /= alpha[0].sum()          # normalize to prevent underflow
    # Recursion
    for t in range(1, T):
        for j in range(K):
            alpha[t, j] = B[j, X[t]] * np.sum(alpha[t-1] * A[:, j])
        alpha[t] /= alpha[t].sum()
    return alpha
```

Normalization at each step prevents numerical underflow for long sequences (otherwise products of many small probabilities underflow to zero). Once `alpha` is computed, the likelihood is one line: `P_X = forward(X, pi, A, B)[-1].sum()`. To recover the true log-likelihood, track the log of each normalization constant and accumulate it.

Viterbi Algorithm - Most Likely Path

The Viterbi algorithm finds $Z^* = \arg\max_Z P(Z \mid X, \lambda)$ using the same dynamic programming structure as Forward, but replacing the **sum** with a **max**:

$$\delta_t(i) = \max_{z_1,\ldots,z_{t-1}} P(x_1, \ldots, x_t,\ z_1, \ldots, z_{t-1},\ z_t = i \mid \lambda)$$

$\delta_t(i)$ is the probability of the **best** path ending in state i at time t. A backpointer $\psi_t(j) = \arg\max_i \delta_{t-1}(i)\, A_{ij}$ records which previous state led to each best path.

Initialization: $\delta_1(i) = \pi_i \cdot B_i(x_1)$, $\psi_1(i) = 0$

Recursion: $\delta_t(j) = B_j(x_t) \cdot \max_i \left[\delta_{t-1}(i)\, A_{ij}\right], \quad \psi_t(j) = \arg\max_i \left[\delta_{t-1}(i)\, A_{ij}\right]$

Termination: $z_T^* = \arg\max_i \delta_T(i)$

Backtracking: $z_t^* = \psi_{t+1}(z_{t+1}^*)$ for $t = T-1, \ldots, 1$

The only difference from Forward is $\sum \to \max$ in the recursion. Same $O(K^2 T)$ complexity.

Viterbi - Worked Example (Weather HMM)

Observations $X = [1, 0, 1]$ (umbrella, no umbrella, umbrella). States: S=0 (Sunny), R=1 (Rainy).

t=1, x_1=1 (umbrella):

$$\delta_1(S) = \pi_S \cdot B_S(1) = 0.6 \times 0.1 = 0.060$$

$$\delta_1(R) = \pi_R \cdot B_R(1) = 0.4 \times 0.8 = 0.320 \quad \leftarrow \text{winner}$$

t=2, x_2=0 (no umbrella):

$$\delta_2(S) = B_S(0) \cdot \max[\delta_1(S) \cdot A_{SS},\ \delta_1(R) \cdot A_{RS}] = 0.9 \cdot \max[0.042,\ 0.128] = 0.115 \quad \psi_2(S){=}R$$

$$\delta_2(R) = B_R(0) \cdot \max[\delta_1(S) \cdot A_{SR},\ \delta_1(R) \cdot A_{RR}] = 0.2 \cdot \max[0.018,\ 0.192] = 0.038 \quad \psi_2(R){=}R$$

t=3, x_3=1 (umbrella):

$$\delta_3(S) = 0.1 \cdot \max[0.115{\times}0.7,\ 0.038{\times}0.4] = 0.1 \cdot 0.081 = 0.0081 \quad \psi_3(S){=}S$$

$$\delta_3(R) = 0.8 \cdot \max[0.115{\times}0.3,\ 0.038{\times}0.6] = 0.8 \cdot 0.035 = 0.0277 \quad \psi_3(R){=}S \quad \leftarrow \text{winner}$$

Backtrace: $z_3^*{=}R \xrightarrow{\psi_3(R){=}S} z_2^*{=}S \xrightarrow{\psi_2(S){=}R} z_1^*{=}R$

Most likely sequence: Rainy → Sunny → Rainy - matches intuition: umbrella implies rain, no umbrella implies sun, umbrella again implies rain.

Viterbi - Python

```python
def viterbi(X, pi, A, B):
    T, K = len(X), len(pi)
    delta = np.zeros((T, K))
    psi   = np.zeros((T, K), dtype=int)
    delta[0] = np.log(pi) + np.log(B[:, X[0]])  # log-space init
    for t in range(1, T):
        for j in range(K):
            scores     = delta[t-1] + np.log(A[:, j])
            psi[t, j]  = np.argmax(scores)
            delta[t, j] = np.log(B[j, X[t]]) + scores[psi[t, j]]
    path = np.zeros(T, dtype=int)
    path[-1] = np.argmax(delta[-1])
    for t in range(T-2, -1, -1):
        path[t] = psi[t+1, path[t+1]]
    return path
```

Working in log-space turns products into sums, avoiding underflow for long sequences - essential when T is in the hundreds or thousands.

Viterbi vs Forward - Side-by-Side

	Forward	**Viterbi**
Core operation	$\sum$ over states	$\max$ over states
Output	$P(X \mid \lambda)$ - likelihood	Best state path Z^*
Interpretation	Marginalizes over all paths	Finds single best path
Auxiliary storage	α matrix only	δ matrix + ψ backpointers
Complexity	$O(K^2T)$	$O(K^2T)$

The only code change is `np.sum` → `np.argmax` / `np.max` in the inner loop, plus the backtracking pass. This structural similarity is not a coincidence - both are instances of the **sum-product** and **max-product** message-passing algorithms on a chain graph.

HMM - Three Algorithms Summary

Problem	Algorithm	Core op
$P(X \mid \lambda)$	Forward	$\sum$
Z^*	Viterbi	$\max$
λ^*	Baum-Welch	EM

All three use **dynamic programming on the chain graph** and run in $O(K^2T)$ per iteration.

Baum-Welch (mentioned for completeness): an EM algorithm that alternates between the Forward-Backward E-step (computing γ_t and ξ_t) and an M-step that updates π, A, B from expected counts. Converges to a local maximum of the likelihood - multiple restarts are standard practice.

How to choose:

- Need to score a sequence against a model → **Forward**
- Need to decode the hidden sequence → **Viterbi**
- Have unlabeled sequences and need to learn the model → **Baum-Welch**

Applications of HMMs:

- Speech recognition (phoneme sequences)
- Gene finding (coding/non-coding regions)
- POS tagging (word → part-of-speech)
- Financial regime detection
- Activity recognition from sensor streams

Exercises

Theory

1. In the Alarm network, is $B \perp M \mid \emptyset$? What about $B \perp M \mid A$? Apply d-separation formally to justify both answers.
2. The Markov Blanket of A is $\{B, E, J, M\}$. Explain why the co-parents (B and E as co-parents of J and M) must be included, not just parents and children.
3. Compare chain, fork, and collider connections. For each: (a) when is the path blocked? (b) give a real-world example.
4. The Forward variable $\alpha_t(i)$ and the Viterbi variable $\delta_t(i)$ differ only in that one uses $\sum$ and the other uses $\max$. Write out the recursion for both side by side and explain intuitively why $\max$ gives the most likely path while $\sum$ gives the total likelihood.

Practical

5. Implement the weather HMM from class ($K = 2$, $\pi = [0.6, 0.4]^\top$, A, B as given). Generate a random observation sequence of length $T = 20$ from the true model. Run Forward to compute the likelihood, then run Viterbi to recover the most likely state sequence. How often does Viterbi match the true hidden states?
6. Modify your Viterbi implementation to work in log-space (replace multiplications with additions). Verify it gives identical paths to the linear-space version on short sequences.
7. Extend the BN CPT exercise: given the Alarm network CPTs and the query $P(\mathrm{Burglary} \mid \mathrm{JohnCalls} = T)$, enumerate all combinations of E and A to compute the answer exactly. Does John calling raise the burglary probability substantially?

Summary - Bayesian Networks and HMMs

Probabilistic Graphical Models

Graphs encode conditional independence. Directed graphs (BNs) factorize the joint as a product of locals CPTs - exponentially more compact than the full joint. The key benefit is not just storage: it enables modular design, efficient inference, and interpretable causal reasoning.

D-Separation

Three connection types determine information flow: chains and forks block when the middle node is observed; colliders block by default and unblock when observed. D-separation reads all conditional independence relationships from the graph without computing any probabilities.

Inference

Exact methods (Variable Elimination, Belief Propagation) are tractable on small or tree-structured networks. For large, loopy

Sampling Methods

Monte Carlo methods estimate expectations via samples, with variance $\propto 1/L$ independent of dimension. MCMC constructs a Markov chain with the target as its stationary distribution. Gibbs Sampling updates one variable at a time from its conditional - tractable in BNs because each conditional reduces to the Markov Blanket.

Hidden Markov Models

HMMs extend BNs to sequential data with a hidden Markov chain driving observed emissions. Three fundamental problems - likelihood, decoding, learning - are solved by Forward, Viterbi, and Baum-Welch respectively. All three run in $O(K^2T)$ through dynamic programming on the chain graph, transforming an exponential problem into a polynomial one.

Connection

Thank You!

What We Covered Today

Probabilistic Graphical Models give us a language for reasoning under uncertainty at scale. Bayesian Networks encode causal structure and exploit conditional independence to make joint distributions tractable. D-Separation lets us read independence directly from the graph.

Hidden Markov Models bring this framework to sequential data - a ubiquitous setting in speech, biology, finance, and time-series ML. The Forward and Viterbi algorithms are dynamic programming at its most elegant: turning exponential state-space enumeration into polynomial recursions.

Looking Ahead

Week 14 is the final lecture of the semester. The final exam will cover:

- Weeks 1–13 material (standard coverage)
- **BNs and HMMs from today** - focus on d-separation reasoning, CPT calculations, and algorithm complexity

For deeper study:

- Koller & Friedman - *Probabilistic Graphical Models* (the definitive reference)
- Rabiner (1989) - *A Tutorial on HMMs* (the original HMM paper, very readable)
- Kevin Murphy - *Machine Learning: A Probabilistic Perspective*, Ch. 17