Probabilistic Graphical Models

Moritz Gerhardt

moricetamol

Contents

I - Introduction . . . 1
II - Basic Probability Theory . . . 1
1 - Events and Spaces . . . 1
(a) - Probability Measures . . . 2
2 - Random Variables . . . 3
3 - Joint Probability Distribution . . . 3
(a) - Marginalization . . . 4
(b) - Bayes Theorem . . . 4
4 - Structural Properties . . . 4
(a) - Independence . . . 4
III - Bayesian Networks . . . 6
1 - Independence in Bayesian Networks . . . 6
2 - Naïve Bayes . . . 9
3 - Independence Maps . . . 10
(a) - Obtaining P-Maps . . . 12
4 - Inference in Bayesian Networks . . . 13
(a) - Inference in Simple Chains . . . 13
(b) - Variable Elimination . . . 13
(c) - Potentials . . . 15
(d) - Abductive Inference in BNs . . . 15
(e) - Complexity of Conditional Probability Queries . . . 16
IV - Markov Random Field (MRF) . . . 18
1 - Domain Graph / Moral Graph . . . 19
2 - Variable Elimination in MRFs . . . 20
3 - Triangulated Graphs . . . 20
4 - Join Trees . . . 21
5 - Junction Trees . . . 22
(a) - Optimal Junction Trees . . . 23
(b) - Propagation on Junction Trees . . . 23
6 - Handling Non-Triangulated Graphs . . . 25
V - Learning Bayesian Networks from Data . . . 26
1 - Data Representation . . . 26
(a) - Hidden / Latent Variables . . . 26
2 - Parameter Estimation . . . 27
(a) - Fixed Structure, Complete Data . . . 27
(b) - Fixed Structure, Incomplete Data . . . 29
3 - Structure Learning . . . 31
(a) - Constraint-Based Structure Learning . . . 31
(b) - Score-Based Structure Learning . . . 32
VI - Approximate Inference . . . 34
1 - Loopy Belief Propagation . . . 34
(a) - Sum-Product Belief Propagation . . . 35
2 - Sampling . . . 37
(a) - Sampling Basics . . . 37
(b) - Forward Sampling . . . 38
(c) - Gibbs Sampling . . . 39
(d) - Likelihood Weighting . . . 41
VII - Dynamic Bayesian Networks (DBNs) . . . 43
1 - Hidden Markov Models (HMMs) . . . 43
(a) - Decoding HMMs . . . 44
(b) - Best State Sequence in HMMs . . . 45
(c) - Parameter Estimation in HMMs . . . 45
2 - Kalman Filters . . . 46
3 - General DBNs . . . 48
VIII - Tractable Probabilistic Models . . . 49
1 - Sum-Product Networks (SPNs) . . . 49
IX - Deep Generative Models . . . 52
1 - Supervised vs. Unsupervised Learning . . . 52
2 - Generative Models . . . 52
3 - Neural Networks . . . 53
(a) - PixelRNN . . . 55
(b) - PixelCNN . . . 56
(c) - Variational Autoencoders (VAEs) . . . 56

(d) - Generative Adversarial Networks (GANs) 58
(e) - Diffusion Models 59
X - Relational Probabilistic Models 61
1 - Relations 61
2 - Markov Logic Networks (MLNs) . 62
3 - Distribution Semantics 63
(a) - Causal Probabilistic Logic . 64

I - Introduction

Probabilistic Graphical Models (PGMs) are a powerful framework for representing and reasoning about uncertainty in complex systems. They combine probability theory and graph theory to model the relationships between random variables.

Main questions in this field include:

- How to represent knowledge?
- How to answer queries with a model using inference?
- How to create the right model from data using learning?

PGMs are used in a wide array of real-world applications, including:

- Speech recognition
- Image and text analysis
- Medical Diagnosis
- Robotics
- Information extraction from documents
- Social network analysis
- Planning under uncertainty
- Combinatorial problems like scheduling or logistics

II - Basic Probability Theory

1 - Events and Spaces

Sample Space Ω

The sample space $\boldsymbol{\Omega}$ or $\mathbf{W}$ is the set of all possible outcomes of a random experiment.

For example, when rolling a six-sided die, the sample space is $\Omega = \{1, 2, 3, 4, 5, 6\}$.

Events

An event is a subset of the sample space Ω. It represents a specific outcome or a group of outcomes.

For example, when rolling a die, the event of rolling an even number can be represented as the set $\{2, 4, 6\}$.

Event Space S

The event space **S** is the set of all *possible* events. It includes all possible groupings of outcomes from the sample space Ω, which means it also contains the empty set $\{\}$ and the entire sample space Ω itself.

For example, when tossing a coin, the sample space is $\Omega = \{\mathrm{H}, \mathrm{T}\}$, and the event space S includes the events $\{\}$, $\{\mathrm{H}\}$, $\{\mathrm{T}\}$, and $\{\mathrm{H}, \mathrm{T}\}$.

(a) - Probability Measures

Probability measures defined over (Ω, S) must satisfy the following axioms:

- $\boxed{\forall \alpha \in S : P(\alpha) \geq 0}$: All probabilities are non-negative.
- $\boxed{P(\Omega) = 1}$: The probability of the entire sample space is 1.
- If α and β are disjoint events, then $\boxed{P(\alpha \cup \beta) = P(\alpha) + P(\beta)}$: The probability of the union of disjoint events is the sum of their individual probabilities.

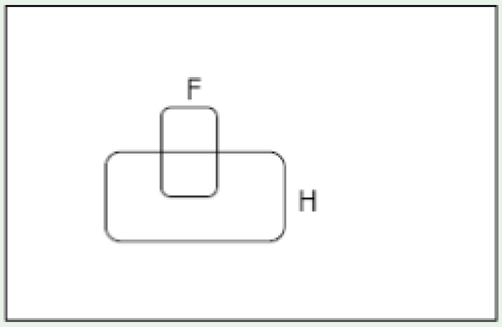
Using these axioms we can define conditional probabilities:

Conditional Probability

The conditional probability of an event **F** given another event **H** is defined as:

$$\boxed{P(F \mid H) = \frac{P(F \cap H)}{P(H)}}$$

provided that P(H) > 0.

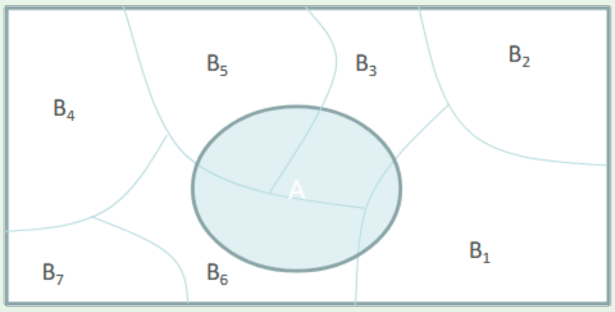


This concept can be expanded to the rule of total probability:

Total Probability

If $\{B_1, B_2, ..., B_n\}$ is a partition of the sample space Ω, then for any event A, the total probability of A can be calculated as:

$$\boxed{P(A) = \sum P(A \mid B_i) * P(B_i)}$$



2 - Random Variables

As defining every single event in the event space S is impractical for large sample spaces, we introduce random variables to simplify the representation of events and their probabilities.

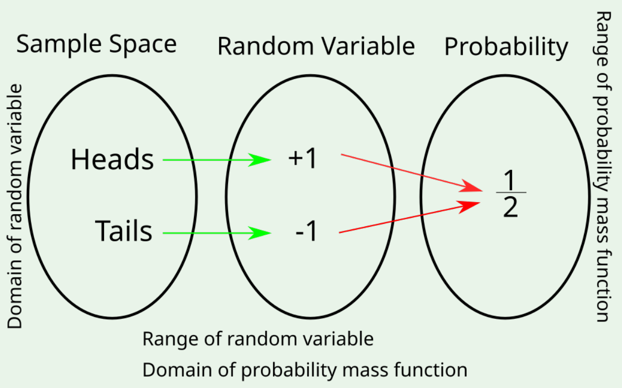
Random Variable

A random variable is a function that maps outcomes from the sample space Ω to real numbers. It assigns a numerical value to each outcome of a random experiment.

For example, take a coin toss where Ω = {H, T}. We can define a random variable X such that:

- X(H) = 1
- X(T) = −1

Since there are only two outcomes, that have an equal number of mappings to real numbers, we can say that $P(X = 1) = \frac{1}{2}$ and $P(X = -1) = \frac{1}{2}$.


Sample Space
Random Variable
Probability
Domain of random variable
Heads
Tails
+1
-1
1/2
Range of probability mass function
Range of random variable
Domain of probability mass function

3 - Joint Probability Distribution

As we've established, random variables encode attributes, but not all possible combinations of these attributes are equally likely. To capture the relationships between multiple random variables, we use joint probability distributions $\boxed{P(X = x, Y = y) = P(x, y)}$.

Chain Rule

The chain rule allows us to decompose a joint probability distribution into a product of conditional probabilities. For two random variables X and Y, the chain rule states that:

$$\boxed{P(x, y) = P(x)P(y|x) = P(y)P(x|y)}$$

This can be expanded to more variables:

$$\boxed{\begin{gathered} P(x, y, z) = P(x)P(y|x)P(z|x, y) = ... \\ P(x, y, z, w) = P(x)P(y|x)P(z|x, y)P(w|x, y, z) = ... \\ P(x_1, x_2, ..., x_n) = P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)...P(x_n \mid x_1, x_2, ..., x_{n-1}) \end{gathered}}$$

Likewise, the conditional probabilities utilizing random variables can be expressed as:

$$\boxed{P(X = x \mid Y = y) = \frac{P(X = x \cap Y = y)}{P(Y = y)}}$$

or simpler:

$$\boxed{P(x \mid y) = \frac{P(x, y)}{P(y)}}$$

(a) - Marginalization

Assume we have a joint probability distribution over multiple random variables, but we are only interested in the distribution of a subset of these variables. Marginalization allows us to obtain the marginal probability distribution of a subset by summing over the unwanted variables:

$$\boxed{P(x) = \sum_y P(x, y)}$$

This also can be expanded to more variables:

$$\boxed{P(x) = \sum_y \sum_z P(x, y, z)}$$

etc.

(b) - Bayes Theorem

Bayes Theorem is essentially a recombination of the definition of conditional probability and the rule of total probability. It is a fundamental concept, that describes how to update your belief about somethine after you get new evidence.

Bayes Theorem is comprised of:
- Prior $P(H)$: The initial belief about the probability of a hypothesis before observing any evidence.
- Likelihood $P(E \mid H)$: The probability of observing the evidence given that the hypothesis is true.
- Marginalization $P(E)$: The total probability of observing the evidence under all possible hypotheses.
- Posterior $P(H \mid E)$: The updated belief about the probability of the hypothesis after observing the evidence.

$$\boxed{\underbrace{P(H \mid E)}_{\text{Posterior}} = \frac{\overbrace{P(H)}^{\text{Prior}} \cdot \overbrace{P(E \mid H)}^{\text{Likelihood}}}{\underbrace{P(E)}_{\text{Marginalization}}}}$$

It can also be expanded to multiple variables:

$$\boxed{P(H_1, \ldots, H_n \mid E_1, \ldots, E_m) = \frac{P(H_1, \ldots, H_n) \cdot P(E_1, \ldots, E_m \mid H_1, \ldots, H_n)}{P(E_1, \ldots, E_m)}}$$

4 - Structural Properties

(a) - Independence

Two random variables X and Y are considered independent if the occurrence of one does not affect the probability of the occurrence of the other. Formally, X and Y are independent if:

$$\boxed{P(X = x, Y = y) = P(X = x) \cdot P(Y = y) \Rightarrow X \perp Y}$$

This also means that $\boxed{P(X \mid Y = y) = P(X)}$.

In most models random variables are rarely independent, but we can still leverage the concept of conditional independence to simplify our models:

Conditional Independence

Two random variables X and Y are conditionally independent given a third variable Z if the occurrence of X does not affect the probability of Y when Z is known. Formally, X and Y are conditionally independent given Z if:

$$\boxed{P(X = x, Y = y \mid Z = z) = P(X = x \mid Z = z) \cdot P(Y = y \mid Z = z) \Rightarrow X \perp Y \mid Z}$$

This also means that:

- $\boxed{P(X \mid Y = y, Z = z) = P(X \mid Z = z)}$
- $\boxed{P(Y \mid X = x, Z = z) = P(Y \mid Z = z)}$

This also brings some properties:

- Symmetry: $\boxed{(X \perp Y \mid Z) \Rightarrow (Y \perp X \mid Z)}$
- Decomposition: $\boxed{(X \perp (Y, W) \mid Z) \Rightarrow (X \perp Y \mid Z)}$
- Weak Union: $\boxed{(X \perp (Y, W) \mid Z) \Rightarrow (X \perp Y \mid (Z, W))}$
- Contraction: $\boxed{(X \perp Y \mid Z) \land (X \perp W \mid (Y, Z)) \Rightarrow (X \perp (Y, W) \mid Z)}$
- Intersection: $\boxed{(X \perp Y \mid (W, Z)) \land (X \perp W \mid (Y, Z)) \Rightarrow (X \perp (Y, W) \mid Z)}$
    - Only holds for strictly positive distributions

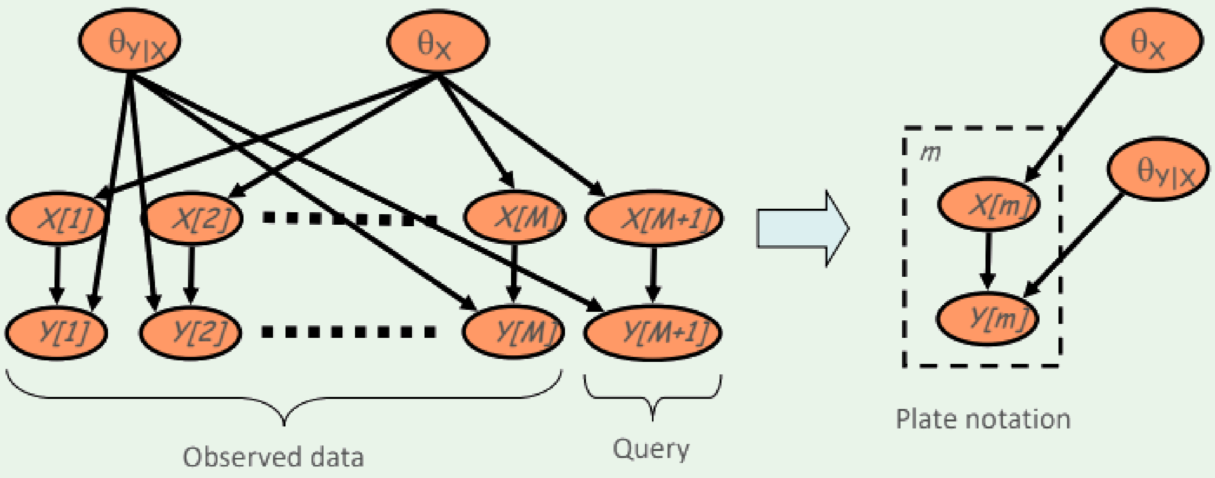
III - Bayesian Networks

As probabilities with multiple conditionals get exponentially more difficult to compute the more parameter / variables we add, we can employ the concept of Bayesian Networks using conditional independences to simplify the distribution.

A Bayesian Network (BN) is a probabilistic graphical model that represents a set of random variables and their conditional dependencies as a directed acyclic graph (DAG). In a BN, each node represents a random variable, and the directed edges between nodes represent the conditional dependencies between these variables.

Plate Notation

Plate notation is a compact way to represent repeated structures in graphical models, such as Bayesian Networks. It uses a rectangular box (plate) to indicate that the enclosed nodes and edges are repeated multiple times, typically for different instances of a variable or a set of variables.

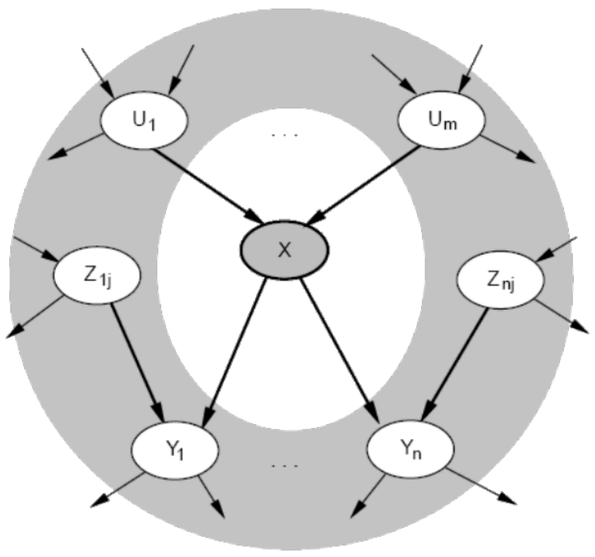


1 - Independence in Bayesian Networks

If we must consider a set of all conditions for a variable, we can reduce the number of parameters we need to store by only considering the parents (direct causes), children (direct effects) and spouses (co-causes) of a variable. This is also known as the Markov Blanket of a variable, as according to the Markov Property.

Local Markov Property

A variable is conditionally independent of all other variables in the network given its Markov Blanket.



Using graphs as a representation not only yields implementational benefits, but also allows us to visually reason about the relationships between variables and their dependencies. Especially, we can apply the independence properties to identify independencies on a given path. To determine the independencies, we usually use **trails**, which are undirected paths between two nodes in the graph. Trails essentially have two states:

- **Active**: A trail is active if there is no node on the trail that blocks the flow of information between the two nodes.
  - This makes the two nodes dependent.
- **Blocked**: A trail is blocked if there is at least one node on the trail that blocks the flow of information between the two nodes.
  - This makes the two nodes **independent**.

Independence Properties in Bayesian Networks

Chain structure:

$$\begin{tikzcd} X \arrow[r] & Z \arrow[r] & Y \end{tikzcd}$$

$\Rightarrow X \not\perp Y$ but: $X \perp Y \mid Z$

- **Blocked Trail if:** Z is **observed**.
- **Active Trail if:** Z is **<u>not</u>** **observed**.

Fork structure:

$$\begin{tikzcd} X & Z \arrow[l] \arrow[r] & Y \end{tikzcd}$$

$\Rightarrow X \not\perp Y$ but: $X \perp Y \mid Z$

- **Blocked Trail if:** Z is **observed**.
- **Active Trail if:** Z is **<u>not</u>** **observed**.

Collider structure:

$$\begin{tikzcd} X \arrow[r] & Z & Y \arrow[l] \end{tikzcd}$$

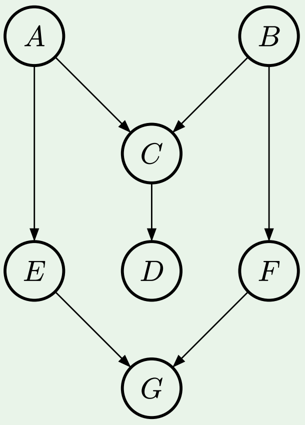
$\Rightarrow X \perp Y$ but: $X \not\perp Y \mid Z$

- **Blocked Trail if:** Z is **<u>not</u>** **observed**.
- **Active Trail if:** Z (<u>or its descendants</u>) is **observed**.

Important: Just because **<u>a structure</u>** is blocked, does not immediately mean that the two nodes are independent. There might be other active trails between the two nodes.

d-separation

Two nodes X and Y are d-separated by a set of nodes Z if all trails between X and Y are blocked by Z. If X and Y are d-separated by Z, then they are conditionally independent given Z. (Note that Z here can also be a set, including the empty set {}.)



In this example, A and B are d-separated given {}, as **all** trails between them are blocked and therefore independent (G not observed and C or D not observed).

If we observe C,D or G, the trail between A and B becomes active, making them dependent.

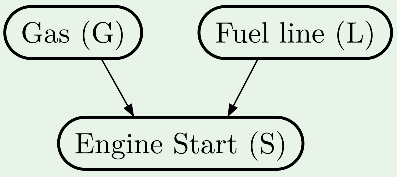
d-separation is useful as it only captures true independencies in the graph structure, making it applicable for any probability distribution that the graph represents.

d-separation Algorithm: Bayes-Ball

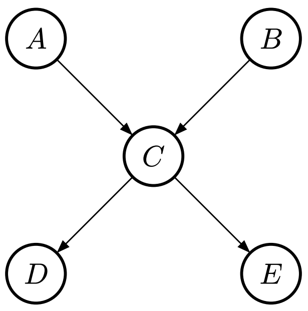
1. Start at source node
2. Get successors:
   - If the current node is **unobserved:**
     - If coming from a parent node, get all children.
     - If coming from a child node, get all parents and children.
   - If the current node is **observed:**
     - If coming from a parent node, **do not** get any successors.
     - If coming from a child node, get all children.
3. Repeat this process for all successors until there are no more successors to explore.
4. If any successor is the target node, then the source and target are not d-separated. If there are no more successors to explore, then the source and target are d-separated.

Context-Specific Independence (CSI)

While d-separation captures independencies that hold for all values of the variables (e.g. $X \perp Y$ always holds, regardless of their values), sometimes independencies only hold for specific values of the conditioning variables. This is known as **Context-Specific Independence (CSI)**. Two variables X and Y might be independent given a specific value of a third variable $Z = z_1$, but dependent given another value $Z = z_2$. For example:



Usually $S \not\perp L$ as the state of the engine start depends on if the fuel line is working. Same with gas: $S \not\perp G$. However, if we know that there is no gas ($G = \text{no Gas}$), then the state of the fuel line does not matter anymore, as the engine won't start regardless. Therefore, in this context we have $\boxed{S \perp L \mid (G = \text{no Gas})}$.



For a Bayesian Network such as this the Joint Distribution normally would be:

$$P(A, B, C, D, E) = P(A)P(B \mid A)P(C \mid A, B)P(D \mid A, B, C)P(E \mid A, B, C, D)$$

However, by leveraging conditional independences we can reduce this to:

$$P(A, B, C, D, E) = P(A)P(B)P(C \mid A, B)P(D \mid C)P(E \mid C)$$

which is significantly easier to compute and store.

So the general form for a Bayesian Network's Joint Distribution is:

$$\boxed{P(X_1, X_2, ..., X_n) = \prod_{i=1}^{n} P(X_i \mid \text{Parents}(X_i))}$$

2 - Naïve Bayes

Naïve Bayes is a simple yet powerful algorithm for classification. Its main goal is to look at a piece of data or "instance", examine its features or "attributes" and assign it to a specific category or "class". One simple example is spam filtering, whereby:

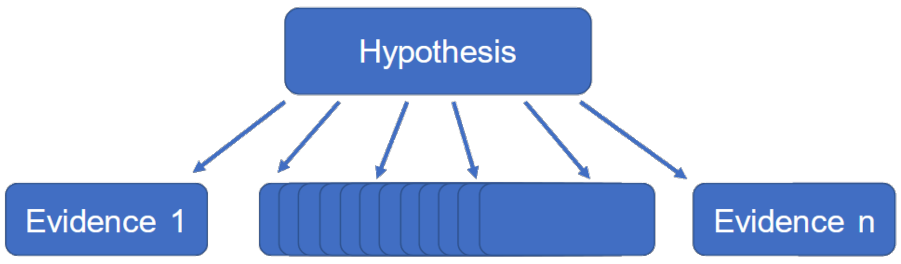
- **Instance**: An email
- **Attributes**: Words in the email
- **Class**: Spam or Not Spam

The model is called "naïve" because it makes a strong assumption that **all attributes are conditionally independent given the class label**. This simplifies the computation so much that makes it feasible to use even with large datasets.

In the example this means that specific words like "estate", "tax", and "income" are considered independent features when determining if an email is spam, though this might not be true in reality.

Formally, the Naïve Bayes classifier is a simple Bayesian Network where one hypothesis H is the parent to all evidence nodes $E_1, E_2, ..., E_n$, with the goal to find the most likely class H_{MAP} (Maximum A Posteriori) given the evidence:

$$\boxed{H_{MAP} = \arg\max_{h_j \in H} \underbrace{P(h_j \mid E_1, E_2, ..., E_n)}_{\text{Posterior}} = \arg\max_{h_j \in H} \underbrace{P(h_j)}_{\text{Prior}} \cdot \prod_{i=1}^{n} \underbrace{P(E_i \mid h_j)}_{\text{Likelihood}}}$$



In naïve bayes, the Hypothesis H is often also referred to as the **class variable C**, while the evidence variables $E_1, E_2, ..., E_n$ are called **attribute variables** or **features**.

There are many more use cases, though Naïve Bayes is mostly used for text classification tasks like spam detection, sentiment analysis, and document categorization.

Despite its “naïve” assumption, Naïve Bayes often performs surprisingly well in practice, especially when the independence assumption holds reasonably well. It is computationally efficient, easy to implement, and requires relatively small amounts of training data to estimate the necessary parameters.

Learning the model

To use a model we need to find the parameters (probabilities) from data. These include:

1. **Class prior probabilities**: $\boxed{P(C = c_j)}$ The overall probability of each class in the training data.
2. **Attribute likelihoods**: $\boxed{P(E_i \mid C = c_j)}$ The probability of each attribute given each class.

The simplest approach is to use Maximum Likelihood Estimation (MLE) to estimate these probabilities from the training data by counting occurrences:

$$\hat{P}(c_j) = \frac{N(c_j)}{N}$$

$$\hat{P}(E_i \mid c_j) = \frac{N(E_i, c_j)}{N(c_j)}$$

The problem here comes from zero probabilities, when an attribute never occurs with a specific class in the training data. This would break the classifier as the entire posterior probability would become zero due to multiplication with zero probabilities. To solve this, we can use Laplace Smoothing (“Add-1” smoothing):

$$\hat{P}(H_i \mid c_j) = \frac{N(H_i, c_j) + 1}{N(c_j) + k}$$

where k is the number of possible values that H_i can take.

For example for text classification, k would be the size of the vocabulary.

3 - Independence Maps

As a probability distribution (P) can be represented as a bayesian network (G), there can be multiple graphs that represent the same distribution. To formalize this, we can use Independence Maps (I-Maps).

Independence Map (I-Map)

A bayesian network G is an I-Map of a probability distribution P if all the conditional independencies represented in G also hold in P. In other words, the graph G is an I-Map of P if all the independency assumptions in G are valid in P.

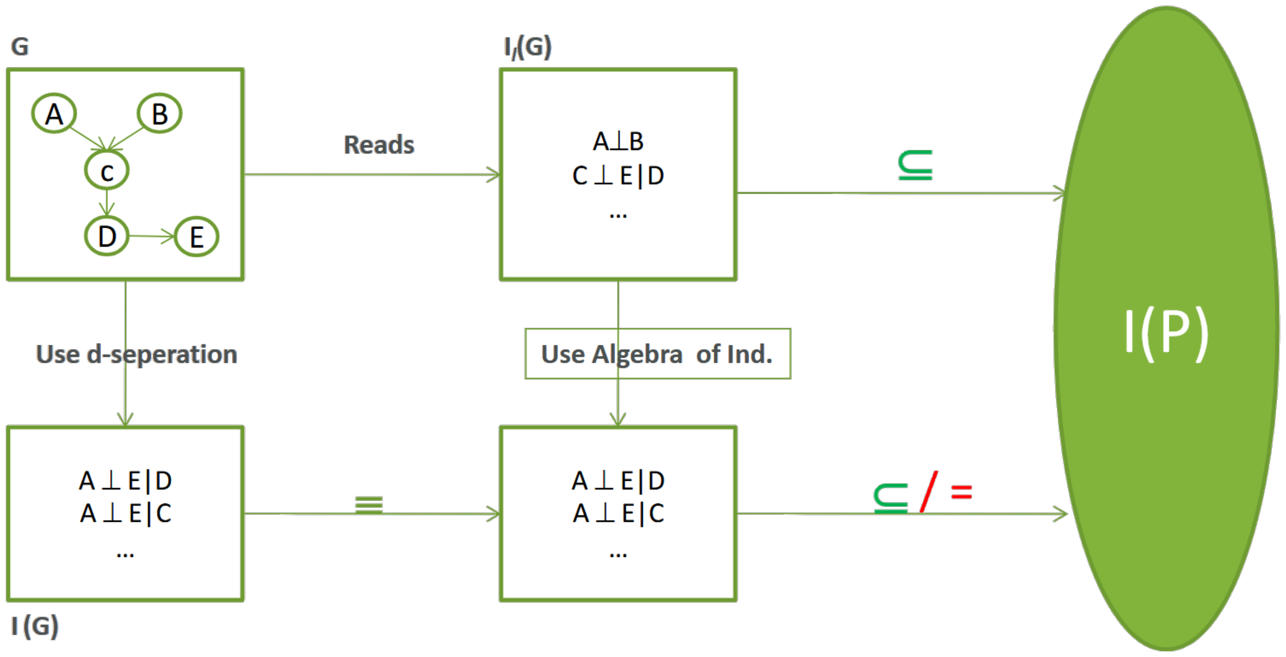
Formally:

$$I_I(G) \subseteq I(P)$$

where $I_I(G)$ are the independencies in G and $I(P)$ are the independencies in P.

Therefore, a distribution P can have multiple I-Maps, as different graphs can represent the same set of conditional independencies.

What this means in practice is, that when constructing a bayesian network, adding extra edges (dependencies) will always result in a valid I-Map, as it does not remove any independencies. However, removing edges (dependencies) can lead to invalid I-Maps if the removed edge represented a true dependency in the distribution P.



Minimal I-Map

An I-Map $I_{l(G)} \subseteq I(P)$ is minimal if removing any edge from G would result in a graph that is no longer an I-Map of P. In other words, a minimal I-Map contains the smallest number of edges necessary to represent all the conditional independencies in P.

Perfect I-Map (P-Map)

A bayesian network G is a Perfect I-Map (P-Map) of a probability distribution P if the conditional independencies represented in G are exactly the same as those in P. In other words, G is a P-Map of P if:

$$\boxed{I_I(G) = I(P)}$$

This means that every conditional independence in G holds in P, and every conditional independence in P is represented in G.

I-Equivalence

Two bayesian networks G_1 and G_2 are I-Equivalent if they represent the same set of conditional independencies. In other words, G_1 and G_2 are I-Equivalent if:

$$\boxed{I_I(G_1) = I_I(G_2)}$$

This means that both graphs encode the same independency relationships among the variables, even if their structures (i.e., the arrangement of nodes and edges) are different.

This also means that:

$$\boxed{I_I(G_1) = I_I(G_2) \wedge I_I(G_1) = I(P) \Rightarrow I_I(G_2) = I(P)}$$

Additionally, if two bayesian networks are I-Equivalent, they will have the same skeleton (the underlying undirected graph) and the same set of v-structures (collider).

Immoralities

An immorality is in concept very close to a v-structure (collider), with the only difference being that the two parent nodes are not directly connected by an edge. This is also often called "unmarried parents (😔)".

Formally, an immorality is a structure of the form $X \to Z \leftarrow Y$ where there is no edge between X and Y.

This is important because:

Two Bayesian Networks G_1 and G_2 have the same skeleton and immoralities *if and only if* G_1 and G_2 are I-Equivalent.

(a) - Obtaining P-Maps

To obtain a P-Map from a given distribution P, we can use the following approach:

1. **Identify the Skeleton**
   (a) Start by assuming every node is connected to every other node (complete undirected graph).
   (b) For each pair of variables X_i, X_j, look for a set of variables U such that $\left(X_i \perp X_j \mid U\right)$
   (c) If such a set U exists, remove the edge between X_i and X_j.
   (d) Continue this process until no more edges can be removed.
   (e) The resulting undirected graph is the skeleton of the graph.

2. **Identify the Immoralities**
   (a) Locate all possible immoralities in the skeleton $(X - Z - Y)$
   (b) Check the separator set U used to remove the edge between X and Y:
      - If Z was not in U:
         - X and Y are independent without knowing Z, therefore Z must be a common effect (collider).
         - Orient the edges as $X \to Z \leftarrow Y$.
      - If Z was in U:
         - This cannot be an immorality
         - Must be a chain or fork structure
         - $X \to Z \to Y$, $X \leftarrow Z \leftarrow Y$ or $X \leftarrow Z \to Y$
         - Leave the edges undirected for now.
   (c) Fill in undirected edges without creating directed cycles or new immoralities IF they are not ambivalent (can go either way).

This might not lead to a perfect directed graph, as some edges could go either way without changing the independencies. The resulting structure is called a **Partially Directed Acyclic Graph (PDAG)**.

This process is very important in **Structure Learning**.

4 - Inference in Bayesian Networks

Inference in BNs is very complex, even approximate inference is NP-hard. In practice, we therefore usually exploit the BN structure.

(a) - Inference in Simple Chains

$$\begin{tikzcd} X_1 \arrow[r] & X_2 \arrow[r] & X_3 \arrow[r] & \dots \arrow[r] & X_n \end{tikzcd}$$

In a simple chain like this, how do we compute $P(X_n)$?

The naïve approach would be to sum over all other variables:

$$\boxed{P(X_n) = \sum_{X_{n-1}} \dots \sum_{X_2} \sum_{X_1} P(X_1, X_2, \dots, X_n) = \sum_{X_{n-1}} \dots \sum_{X_2} \sum_{X_1} P(X_1) P(X_2 \mid X_1) \dots P(X_n \mid X_{n-1})}$$

This, of course, is incredible inefficient, as the number of summations grows exponentially with n: $O(k^n)$ with k being the number of possible values for each variable.

One easier way is to iteratively compute $P(X_1)$, $P(X_2)$, ..., $P(X_n)$:

$$\boxed{P(X_{i+1}) = \sum_{x_i} P(X_i) \cdot P(X_{i+1} \mid X_i)}$$

This is a lot more efficient as we only have to do n summations, each with k terms, leading to a total complexity of $O(n \cdot k^2)$.

(b) - Variable Elimination

Variable elimination is an algorithm for performing inference in Bayesian Networks by systematically eliminating variables from the network to compute the desired probabilities more efficiently.

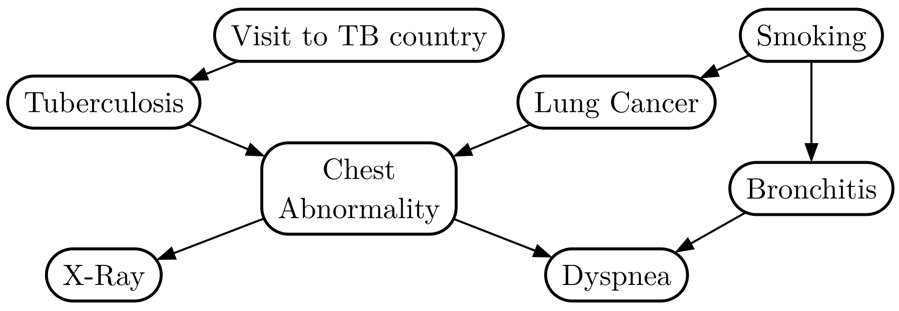
The main idea behind variable elimination is to break down the joint probability distribution into smaller factors, which can be manipulated and combined to compute the desired probabilities without having to consider the entire joint distribution at once.

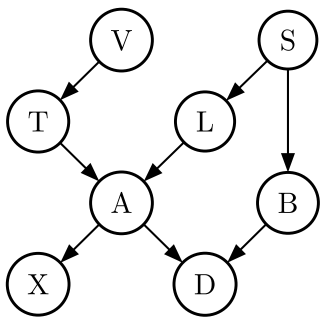
Generally, we want to write a query in the form:

$$\boxed{P(X_n, e) = \sum_{x_k} \dots \sum_{x_3} \sum_{x_2} \prod_i P(X_i \mid \text{Parents}(X_i))}$$

And then iteratively:
1. Move all irrelevant factors (not involving query or evidence variables) out of the innermost sum
2. Perform innermost sum, creating a new factor (or potential)
3. Insert the new term back into the distribution

For example:





The probability distribution for this network is:

$$\boxed{P(v,s,t,l,b,a,x,d) = P(v)P(s)P(t \mid v)P(l \mid s)P(b \mid s)P(a \mid t,l)P(x \mid a)P(d \mid b,d)}$$

Assume we want to compute $P(d)$. In that case we need to eliminate: v, s, x, t, l, a, b.

Starting with eliminating v:

1. Gather all factors involving v: $P(v), P(t|v)$
2. Sum over v: $f_v(t) = \sum_v P(v)P(t|v)$
   - This is just $P(t)$
3. New factors: $f_v(t)(= P(t)), P(s), P(l|s), P(b|s), P(a|t,l), P(x|a), P(d|a,b)$

Repeat this for all other variables:

$$\begin{aligned}
P(v,s,x,t,l,a,b,d) &= \underline{P(v)}P(s)\underline{P(t|v)}P(l|s)P(b|s)P(a|l,t)P(x|a)P(d|a,b) \\
v \Rightarrow P(s,x,t,l,a,b,d) &= \boldsymbol{f_v(t)}\underline{P(s)P(l|s)P(b|s)}P(a|l,t)P(x|a)P(d|a,b) \\
s \Rightarrow P(x,t,l,a,b,d) &= f_v(t)\boldsymbol{f_s(b,l)}P(a|t,l)\underline{P(x|a)}P(d|a,b) \\
x \Rightarrow P(t,l,a,b,d) &= \underline{f_v(t)}f_s(b,l)\boldsymbol{f_x(a)}\underline{P(a|t,l)}P(d|a,b) \\
t \Rightarrow P(l,a,b,d) &= \underline{f_s(b,l)}f_x(a)\underline{\boldsymbol{f_t(a,l)}}P(d|a,b) \\
l \Rightarrow P(a,b,d) &= \underline{f_x(a)\boldsymbol{f_l(a,b)}P(d|a,b)} \\
a \Rightarrow P(b,d) &= \underline{\boldsymbol{f_a(b,d)}} \\
b \Rightarrow P(d) &= \boldsymbol{f_b(d)}
\end{aligned}$$

(Note: The underlined terms are the ones that get moved out of the summation in the next step, while the **colored-in terms** are the new factors created after summation.)

If a factor like $P(x \mid a)$ stands alone (no other factors involving the same variable), we can directly sum it out, as it will always sum to 1.

Variable Elimination with Evidence

When evidence is present (e.g. $V = \texttt{true}$), we do not need to eliminate the evidence variables. Instead, we can simply replace the factors involving the evidence variables with their observed values:

$$\boxed{\begin{gathered} f_{P(V)} = P(V = t) \\ f_{P(T \mid V)}(T) = P(T \mid V = t) \end{gathered}}$$

The resulting terms that are constant can then be moved out of the summations during variable elimination, simplifying the computation further. The rest of the process remains the same as before.

In general, the efficiency of variable elimination depends a lot on the order in which variables are eliminated. Choosing an optimal elimination order can significantly reduce the computational complexity of the inference process. However, finding the optimal order is itself a challenging problem and often requires heuristics or approximations.

In graphs that are "tree-like", one common practice is to start eliminating from the leaves of the tree towards the root, as this often leads to smaller intermediate factors and more efficient computation.

(c) - Potentials

Potential

A **potential** f_A or more often ϕ_A over a set of variables A is a function that maps each configuration into a *non-negative real number*. Therefore it is not necessarily a probability, a conditional probability, or anything else. In essence it's just a table of numbers associated with each configuration of the variables in A.

Conditional Probability Table $P(F \mid D, E)$:

D	E	P(F)
T	T	0.8
T	F	0.5
F	T	0.2
F	F	0.7

Potential $\phi_{D,E,F}$:

d	e	f	0.8
d	e	$\neg f$	0.2
d	$\neg e$	f	0.5
d	$\neg e$	$\neg f$	0.5
$\neg d$	e	f	0.2
$\neg d$	e	$\neg f$	0.8
$\neg d$	$\neg e$	f	0.7
$\neg d$	$\neg e$	$\neg f$	0.3

Potentials can be combined using **factor multiplication** and **factor marginalization** (summing out variables), similar to how we handled probabilities in variable elimination.

(d) - Abductive Inference in BNs

So far we've only discussed the goal to obtain posterior probabilities given evidence. Abductive inference aims to find the configuration of a set of variables (hypotheses) which best explains the observed evidence.

We usually differentiate between two types of abductive inference:

Maximum A Posteriori (MAP)

The most probable configuration of a subset of variables given the evidence:

$$MAP = \max_m P(M = m \mid E = e) = \max_m \sum_r P(M = m, R = r \mid E = e)$$

with $M \subseteq H =$ the set of hypothesis variables and

$R = H \setminus M =$ the set of remaining hypothesis variables

Most Probable Explanation (MPE)

The most probable configuration of <u>all variables</u> given the evidence:

$$MPE = \max_h P(H = h \mid E = e)$$

with H = the set of all hypothesis variables

For example:

Assume we have a bayesian network representing a medical diagnosis system, where:
- Hypotheses: Flu (F), Allergy (A), Sinus infection (S) and Nose running (N)

given:
- Evidence: Headache (H) = `true`

The **MPE** would then be defined as

$$\max_{f,a,s,n} P(F = f, A = a, S = s, N = n \mid H = t)$$

(the best overall explanation for the headache considering all variables),

whereas the **MAP** could be defined as

$$\max_a P(A = a \mid H = t)$$

(The best single value for allergy considering all other possibilities.)

MPE and MAP are not consistent. Just because a value is part of the MPE, does not imply that it is part of the MAP. "The most probable complete story (MPE) might not contain the most probable individual part (MAP)"

Finding MPE

To find the MPE, we can use a modified version of variable elimination:
- Instead of summing over the hypothesis variables, we take the maximum.
- The rest of the process remains the same as in standard variable elimination.

So the individual steps for each variable we want to eliminate are:
1. Gather all factors involving the variable.
2. Take the maximum of the product of these factors over the variable.
3. Insert the new factor back into the distribution.

This might seem a bit weird, as this would just yield the probability of the MPE, but not the actual configuration. What we do to get the configuration is to keep track of the values that yielded the maximum ($\underline{\arg}\max$) for each step. Once we have attained the MPE probability, we can backtrack through these stored values to reconstruct the most probable configuration of the hypothesis variables.

(e) - Complexity of Conditional Probability Queries

As mentioned before, inference in Bayesian Networks is generally NP-hard. This means that there is no known polynomial-time algorithm that can solve all instances of the problem efficiently. The complexity arises from the combinatorial nature of the problem, as the number of possible configurations of the variables grows exponentially with the number of variables in the network.

This, does however, not mean, that we cannot solve inference for any given network. There simply is no general algorithm that can solve *all*.

IV - Markov Random Field (MRF)

A Markov Random Field (MRF), also known as a Markov Network, is an undirected graphical model that represents the joint distribution of a set of random variables. In an MRF, nodes represent random variables, and edges represent dependencies between these variables. Unlike Bayesian Networks, which use directed edges to represent causal relationships, MRFs use **undirected** edges to capture mutual dependencies.

So in essence, the difference between BNs and MRFs is, that BNs represent conditional probability distributions using directed edges, while MRFs are parameterized by **potentials** over cliques (fully connected subgraphs) in the graph using undirected edges.

Cliques

A **clique** is a subset of nodes where every node has a direct edge to every other node in the subset. A **maximal clique** is a clique that cannot be extended by including an adjacent node, meaning it is not a subset of a larger clique.

Joint Distribution in MRFs

The probability of a configuration of random variables $x = \{x_1, x_2, ..., x_n\}$ in an MRF is given by:

$$P(X = x) = \frac{1}{Z} \prod_k \phi_k\left(x_{\{k\}}\right)$$

$$\text{where } k = \text{the cliques in the graph}$$

$$Z = \text{the partition function: sum of the product of potentials over all cliques}$$

$$x_{\{k\}} = \text{the variables in clique k}$$

The **partition function Z** is a normalizing constant that ensures that the probabilities sum to 1 over all possible configurations of the variables. It is computed by summing the product of the potentials over all possible configurations of the variables in the MRF. This can be computationally expensive, especially for large networks with many variables and complex dependencies ($O(k^n)$, with k being the number of states per variable).

Markov Assumptions in MRFs

MRFs rely on the Markov assumptions to simplify the representation of dependencies between variables. The key Markov assumptions in MRFs are:

1. **Pairwise Markov Property**: Two non-adjacent nodes are conditionally independent given all other nodes in the graph.
2. **Local Markov Property**: A node is conditionally independent of all other nodes in the graph given its neighbors.
3. **Global Markov Property**: If a set of nodes S seperates a set A from a set B (All paths from A to B pass through S), then A and B are conditionally independent given S ($A \perp B \mid S$).

(Note: Global $\Rightarrow$ Local $\Rightarrow$ Pairwise)

Usually the Global Markov Property is equivalent to the other two, but not always, for example when there are deterministic relationships between variables.

If $\forall x : P(x) > 0$ (for all configurations), then the three properties are equivalent (Hammersley-Clifford Theorem).

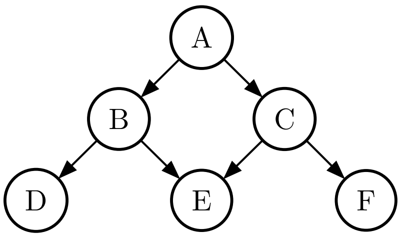
These properties allow us to factor the joint distribution into smaller, more manageable components, making it easier to work with and perform inference on the MRF.

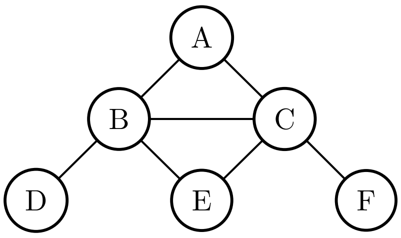
1 - Domain Graph / Moral Graph

Formal Definition

Let $F = \{f_1, ..., f_n\}$ be a set of potentials over a set of variables $U = \{A_1, ..., A_m\}$. Each potential f_i has a domain D_i, over the variables it depends on. Hereby:

- The **domain graph** is an **undirected** graph $G = (V, E)$ where:
  - $V = U$: The nodes are the set of variables
  - $E = \{(A_i, A_j) \mid \exists f_k \in F : A_i \in D_k \land A_j \in D_k\}$ (an edge between two variables if they appear together in the domain of the same potential)





CPDs:

$P(A)$
$P(B \mid A)$
$P(C \mid A)$
$P(D \mid B)$
$P(E \mid B, C)$
$P(F \mid C)$

Domains:

$D_1(\phi_A) = \{A\}$
$D_2(\phi_B) = \{B, A\}$
$D_3(\phi_C) = \{C, A\}$
$D_4(\phi_D) = \{D, B\}$
$D_5(\phi_E) = \{E, B, C\}$
$D_6(\phi_F) = \{F, C\}$

The domain graph (also moral graph) is a graph where edges between nodes represent the occurrence of variables together in the same potential. It is undirected, as the relationships represented by potentials do not have a direction like in Bayesian Networks. As such **it is an MRF** representation of the same distribution as the original Bayesian Network.

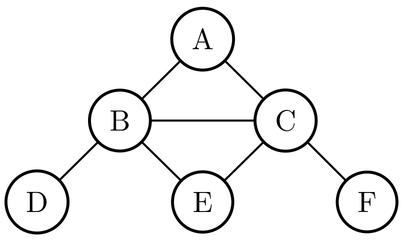
As we lose independence information when converting from a BN to a domain graph, we cannot go back from a domain graph to a BN uniquely.

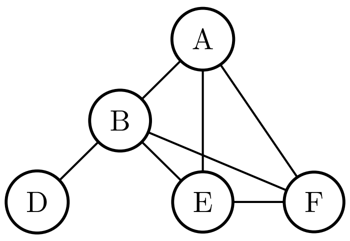
We can do variable elimination on the domain graph similarly to how we did it on the Bayesian Network, by eliminating nodes and connecting their neighbors. This process helps us understand how the elimination of variables affects the dependencies between the remaining variables in the network.

2 - Variable Elimination in MRFs

The steps for variable elimination in MRFs are:
1. Select a variable to eliminate.
2. Identify all neighbors of the variable in the graph.
3. Connect all neighbors to each other (if not already connected) to form a clique. (Fill-ins)
4. Remove the variable from the graph.

This process is repeated until all variables except the query and evidence variables are eliminated. The resulting graph can then be used to compute the desired probabilities more efficiently.





This example shows the elimination of variable C. Its neighbors B, A, E, and F are all connected to each other to form a clique before removing C from the graph.

This is very dependant on the elimination order, as eliminating a variable with many neighbors can create large cliques, increasing the complexity of subsequent eliminations.

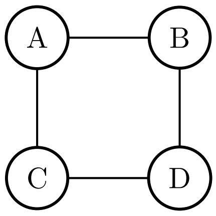
We usually want to avoid creating large cliques (so eliminating variables with many neighbors) to keep the graph as sparse as possible, which helps in reducing the computational complexity of inference. So eliminating "leaf nodes" first (nodes with few connections) is often a good strategy.

If we have a tree-structured graph, we can always eliminate leaf nodes first without creating any new edges, keeping the graph a tree throughout the process, making inference linear.

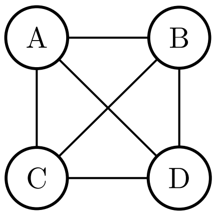
3 - Triangulated Graphs

A graph is **triangulated** if it has a perfect elimination order (no fill-ins).

More formally, a graph is triangulated if every cycle of **four or more nodes** has a chord (an edge that is not part of the cycle but connects two nodes of the cycle). This property ensures that when we perform variable elimination, we do not need to add any fill-in edges, as the existing edges already connect all necessary neighbors.

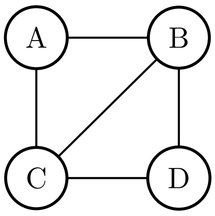


Not Triangulated



Triangulated

(Order does not matter)



Triangulated

(Order: Start with A or D)

Simplicial Nodes

A **simplicial node** is a node whose neighbors form a clique. In other words, all the neighbors of a simplicial node are directly connected to each other.

Eliminating simplicial nodes does not require any fill-in edges, as their neighbors are already fully connected. Therefore, if we can find a perfect elimination order that consists entirely of simplicial nodes, the graph is triangulated:

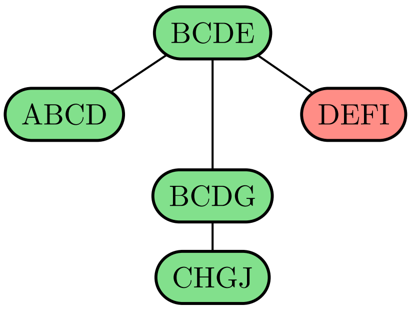
Let G be a triangulated graph and X a simplicial node in G.

Then the graph G' obtained by removing X from G is also triangulated.

4 - Join Trees

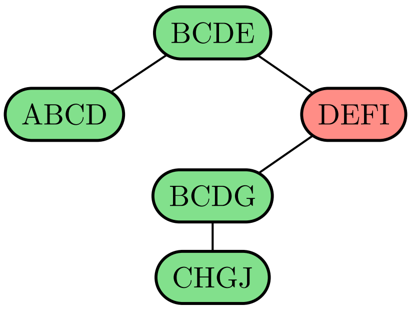
A graph G consisting of the set of cliques from an undirected graph is a **join tree** if it is a tree and satisfies the **running intersection property**:

G is a **join tree** if for any pair of cliques V, W in G, the intersection V ∩ W is contained in every clique on the unique path between V and W in G.



Join Tree

(All nodes on path contain C)



Not a Join Tree

(A Node on path does not contain C)

It holds:

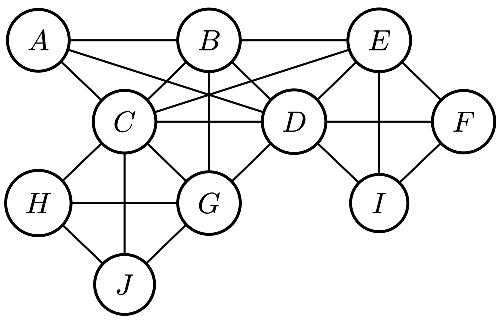
Join Tree ⇔ Triangulated Graph

If G is a triangulated graph, then the graph of its maximal cliques is a join tree.

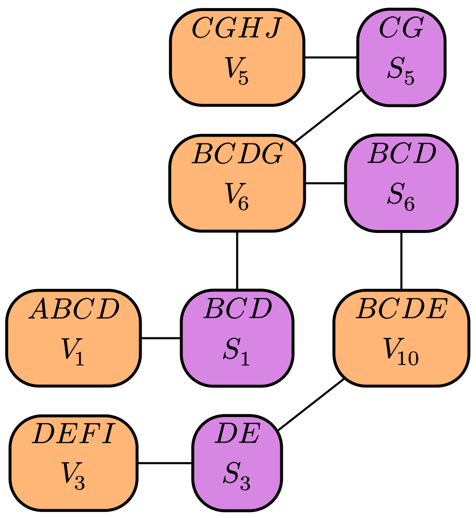
Conversely, if the graph of maximal cliques of G is a join tree, then G is triangulated.

Going from Triangulated Graph to Join Tree

1. Identify cliques
   (a) Find a simplicial node x in the triangulated graph
   (b) Family of x is a maximal clique C
   (c) Elimate all nodes from C that only have neighbors in C
   (d) Give clique of x number i = number of eliminated nodes so far and denote clique as V_i
   (e) Denote the set of remaining nodes as S_i (**separators**)
   (f) Repeat until there are no nodes left to eliminate
2. Connect as tree
   (a) Each clique V_i is connected to its separator S_i
   (b) Connect each seperator S_i to a clique V_j, with $j > i$, such that $S_i \subseteq V_j$



Triangulated Graph



Triangulated Graph

The potential representation of a join tree is the product of the clique potentials, divided by the product of the seperator potentials

$$P(X) = \frac{\prod_C \phi_C(X)}{\prod_S \phi_S(X)}$$

5 - Junction Trees

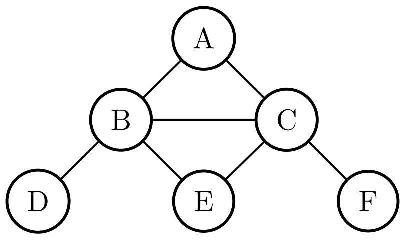
A **Junction Tree** is a specialized type of join tree used in probabilistic graphical models, particularly for performing efficient inference in Bayesian Networks and Markov Random Fields.

It is constructed from the cliques of a triangulated graph and satisfies the running intersection property, similar to join trees. In addition to the basic structure of a join tree, junction trees also contain information about the **potentials associated with each clique and separator**.

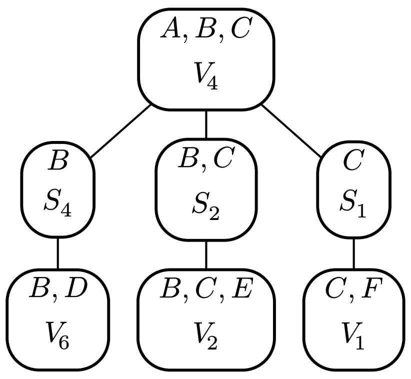
Formal Definition

Let F be a set of potentials with a triangulated domain graph G. A **junction tree** for F is a join tree for G with:

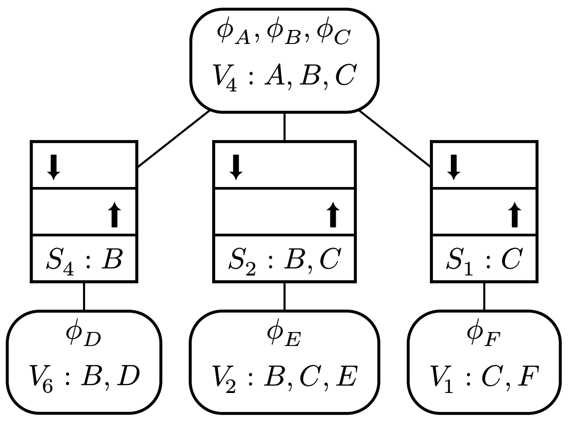
- Each potential ϕ in F is associated to a clique containing the domain $D(\phi)$
- Each link has separator attached containing two "mailboxes", one for each direction of message passing.



MRF



Join Tree



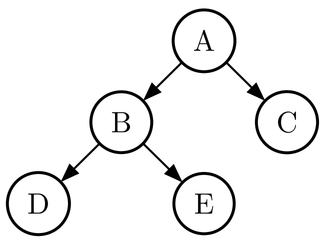
Junction Tree

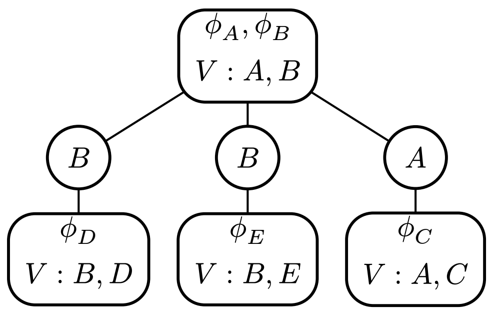
(a) - Optimal Junction Trees

We can always find a **trivial junction tree** with one node containing all variables of the original graph. This is undesirable, as it does not help in reducing the complexity of inference.

An **optimal junction tree** is a junction tree that that minimize the size of the cliques. Again, finding the optimal junction tree is NP-hard, but heuristics exist to find good approximations.

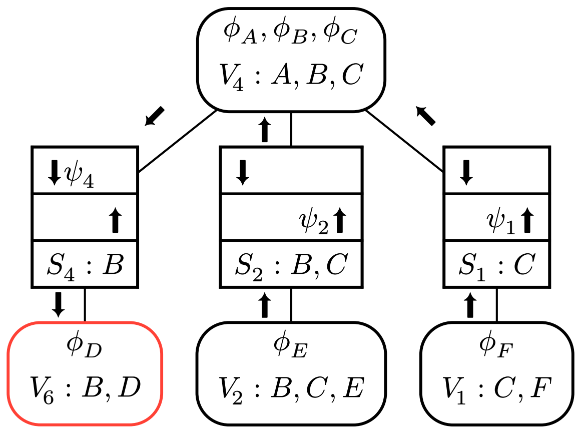
A special case of optimal junction trees are those derived from **tree-structured BNs**:





(b) - Propagation on Junction Trees

- Each node V can send exactly one message to a neighbor W, **only** after it has received messages from all of its other neighbors.
- Choose one clique as the root of the tree.
- Collect message to this node and then distribute messages from this node to all other nodes.
- After collection and distribution, each clique contains the marginal distribution over its variables.



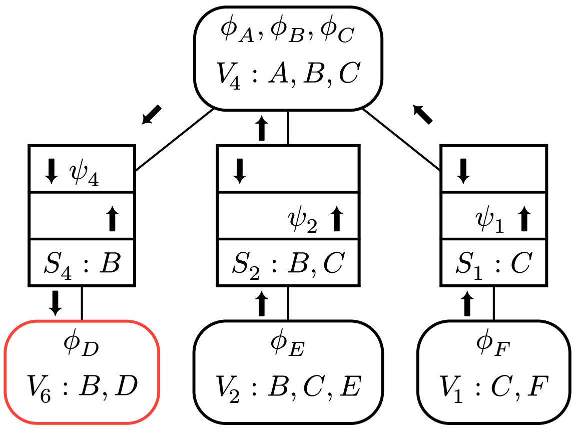
Here we want to find $P(D)$, so we need to find the clique containing D, which is V_6. We choose this as the root of the tree.

We then send messages from leaves to the root. V_4 assembles messages from its other neighbors before sending to V_6.

Message Calculation

The message ψ sent from clique V to clique W over separator S is calculated as:

- Marginalize V's potential to get new potential $\phi_S^* = \psi_S$ for S:
$$\phi_S^* = \sum_{V \setminus S} \phi_V$$
- Update the potential for W:
$$\phi_W^* = \phi_W \cdot \frac{\phi_S^*}{\phi_S}$$
- Update the separator potential (Initially all separator potentials are 1):
$$\phi_S = \phi_S^*$$

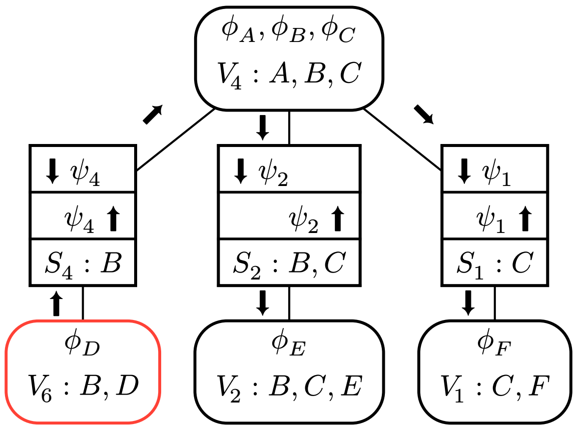


In this example:

$$\phi_1^* = \sum_{V_1 \setminus S_1} \phi_F = \sum_F P(F \mid C) = 1$$
$$\phi_2^* = \sum_{V_2 \setminus S_2} \phi_E = \sum_E P(E \mid B, C) = 1$$
$$\phi_4^* = \sum_{V_4 \setminus S_4} (\phi_A \cdot \phi_B \cdot \phi_C \cdot \phi_1^* \cdot \phi_2^*) = \sum_A \phi_A \cdot \phi_B \cdot \sum_C \phi_C \cdot \phi_1^* \cdot \phi_2^*$$
$$= \sum_A P(A)P(B \mid A) \cdot \sum_C P(C \mid A) \cdot 1 \cdot 1$$
$$= P(B)$$

$$P(D) = \sum_B \phi_D \cdot \phi_4^* = \sum_B P(D \mid B)P(B) = \sum_B P(B, D) = P(D)$$

To get the other marginals, we can just send messages from the root to the leaves, using the same message calculation as before. After this, each clique will contain the marginal distribution over its variables.



6 - Handling Non-Triangulated Graphs

Real-world bayesian networks often do not lead to triangulated domain graphs. In such cases, we need to triangulate the graph before constructing a junction tree.

We can do that by:
- Choose an elimination order for the variables.
- During elimination, add fill-in edges to connect all neighbors of the eliminated variable, ensuring that the resulting graph is triangulated.
- If we then use the original graph with the added fill-in edges to construct the junction tree, we can perform inference as usual.

V - Learning Bayesian Networks from Data

Learning is the process of improving a model over time based on observed data.

For Bayesian Networks, we essentially want to solve this formula:

$$\text{Data} + \text{Prior Knowledge} \overbrace{\longrightarrow}^{\text{Learning Algorithm}} \text{Model Parameters}$$

We want to learn from data because:

- **Knowledge Acquisition Bottleneck:** It is expensive, difficult and time-consuming to get experts to manually specify all the probabilities in a Bayesian Network.
- **Data Availability:** Data is cheap and abundant in many domains, making it feasible to learn models directly from data.

1 - Data Representation

In general, we assume a dataset D to be consisting of **attributes / variables** $A_1, A_2, ..., A_n$ and **data classes** $X_1, X_2, ..., X_m$, whereby each each pair of attribute and class has a specific value.

Unfortunately, real-world data is often incomplete, meaning some of the states for some random variables might be missing. Other variables might be unobserved (**latent or hidden**) altogether.

	A_1	A_2	...	A_n
X_1	true	true	...	false
X_2	false	true	...	true
$\vdots$	$\vdots$	$\vdots$	...	$\vdots$
X_m	false	true	...	true

Ideal, Complete Dataset

	A_1	A_2	...	A_n
X_1	?	?	...	false
X_2	false	?	...	true
$\vdots$	$\vdots$	$\vdots$	...	$\vdots$
X_m	false	?	...	?

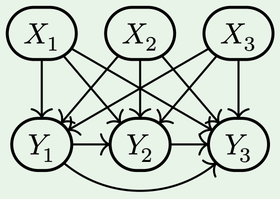
Real-World, Incomplete Dataset

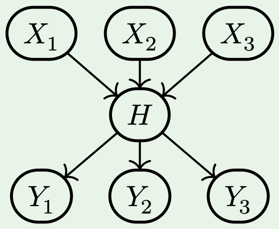
(a) - Hidden / Latent Variables

Hidden or latent variables are not always unwanted. When modelling complex systems, they oftentimes are introduced conciously to simplify a model or to capture underlying patterns in the data.

Simplification / Parameter Reduction

By introducing latent variables, we can often reduce the number of parameters needed to represent the model. This is particularly useful when dealing with high-dimensional data, where the number of possible configurations of observed variables can be extremely large.



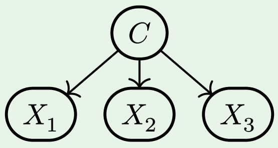


Clustering

Latent variables can also help in capturing underlying patterns or structures in the data. For example, in clustering tasks, latent variables can represent cluster memberships, allowing the model to group similar data points together based on shared characteristics.

We've already seen that in Naïve Bayes, but it's also used in other models such as Autoclass or kMeans.

Hereby, we have a bunch of observed variables $X_1, X_2, ..., X_n$ but not labels for clusters. We introduce a latent variable C that acts as a parent node for all observed variables, representing the cluster assignment. We assume that the observed variables are conditionally independent given the cluster assignment.



2 - Parameter Estimation

Given a fixed structure and $\mathcal{X} = \{X_1, X_2, ..., X_n\}$ a set of data over m random variables, where $X_i \in \mathcal{X}$ is called a **data case**, we want to find the parameters Θ of the CPDs that best match the observed data.

In parameter estimation, we usually necessitate that we know the structure of the Bayesian Network beforehand.

(a) - Fixed Structure, Complete Data

The simplest case is when we have a fixed structure and complete data. Here, we can use **Maximum Likelihood Estimation (MLE)** to find the best parameters for the CPDs. In MLE, we want to find the parameter Θ that are most likely to have produced the observed data.

Hereby, we utilize the **IID** assumption, meaning that we assume that all data points are **Independent and Identically Distributed**. This means that every data case is sampled independently from the same underlying distribution.

$$\begin{aligned}\Theta^* &= \arg\max_{\Theta} P(\mathcal{X}|\Theta) \\ &= \arg\max_{\Theta} P(\Theta|\mathcal{X}) \cdot \frac{P(\Theta)}{P(\mathcal{X})} \\ &= \arg\max_{\Theta} P(\Theta|\mathcal{X})\end{aligned}$$

This is because $P(\mathcal{X})$ is constant with respect to Θ and we usually assume a uniform prior over the parameters, making $P(\Theta)$ constant as well.

To make calculation easier, we usually work with the log-likelihood ($\mathcal{LL}$) instead, as the logarithm is a monotonically increasing function and does not change the location of the maximum.

$$\begin{aligned}\Theta^* &= \arg\max_{\Theta} \log P(\mathcal{X}|\Theta) \\ &= \arg\max_{\Theta} \log P(\Theta|\mathcal{X})\end{aligned}$$

Since we are working with Bayesian Networks, we can decompose the likelihood according to the structure of the network; This means, that instead of working with the joint distribution over all variables, we can work with the product of the CPDs for each variable given its parents.

$$\begin{aligned}\mathcal{LL}(\Theta|\mathcal{X}) &= \sum_{j=1}^{m}\sum_{i=1}^{n} \log P\left(x_i^j | pa\left(x_i^j\right), \Theta_i\right) \\ &= \sum_{j=1}^{m} \mathcal{LL}(\Theta_j|\mathcal{X})\end{aligned}$$

MLE for Multinomial Variables

For multinomial variables (variables with discrete states) the MLE for the parameters of the CPDs can be calculated as the relative frequencies of the observed data:

$$\Theta_k^* = \frac{N_k}{\sum_l N_l}$$

where:

- N_k is the number of occurrences of state k in the data
- $\sum_l N_l$ is the total number of observations

MLE for Conditional Multinomial Variables

If we add conditional variables (parents), the MLE can be calculated as:

$$\Theta_{k|pa}^* = \frac{N_{k,pa}}{N(pa)}$$

where:

- $N_{k,pa}$ is the number of occurrences of state k given parent configuration pa in the data
- $N(pa)$ is the total number of observations for parent configuration pa

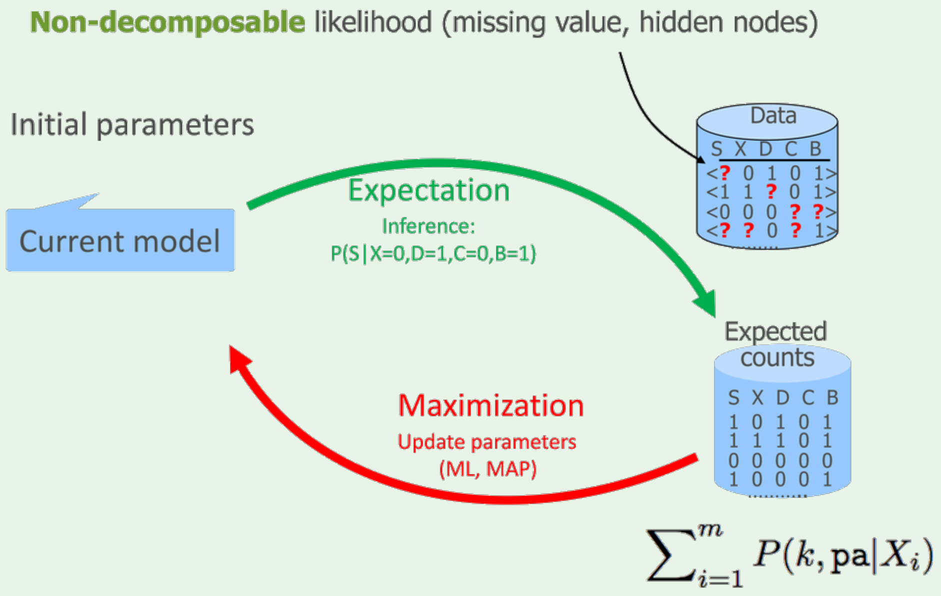
(b) - Fixed Structure, Incomplete Data

Since real-world data is often incomplete, which makes parameter estimation more difficult, as we can no longer simply "count occurrences" of states in the data, we can use the **Expectation-Maximization (EM)** algorithm to iteratively estimate the parameters.

Expectation-Maximization (EM) Algorithm

The EM algorithm consists of two main steps that are repeated until convergence:

- **Expectation:** In this step, we use the current estimates of the parameters to compute the expected sufficient statistics for the missing data. This involves calculating the expected counts of each state for each variable, given the observed data and the current parameter estimates.
- **Maximization:** In this step, we use the expected sufficient statistics computed in the E-step to update the parameter estimates. This is done by maximizing the expected log-likelihood with respect to the parameters, similar to the MLE approach for complete data.



Formal EM Algorithm

1. Initialize parameters Θ^0 randomly or based on prior knowledge.
2. Compute pseudo counts for each variable:

$$\Theta^*_{k|pa} = \frac{\sum_{i=1}^{m} P(k, pa \mid X_i)}{\sum_{i=1}^{m} P(pa \mid X_i)}$$

(e.g. junction tree algorithm)

3. Set parameters to the Maximum Likelihood Estimates based on the pseudo counts.
4. If not converged, return to step 2.

This works, as the algorithm is monotonically increasing the likelihood of the observed data with each iteration, and it can be shown that it converges to a local maximum of the likelihood function.

In practice, this algorithm works, but not without its pitfalls:

- **Initial Parameters:** The choice of initial parameter values can significantly affect the outcome and speed of the EM algorithm.

- **Stopping Criteria:** Deciding when to stop the iterations can be tricky. Common criteria include checking for the size of the change in likelihood or parameters between iterations.
- **Local Maxima:** The EM algorithm can converge to local maxima of the likelihood function, which may not be the global maximum. Restarts with different initializations can help mitigate this issue.
- **Convergence Speed:** The convergence of the EM algorithm can be slow, especially when the amount of missing data is large or when the model is complex.

So we essentially just fill in the missing data based on our current model, and then re-estimate the parameters based on this "completed" data. This process is repeated until the parameters converge to stable values.

After we have arrived at our final parameter estimates, we can then simply use the learned parameters as before:

"MLE" of Multinomials with Incomplete Data

For multinomial variables with incomplete data, we can use the expected counts from the E-step of the EM algorithm to estimate the parameters:

$$\Theta_k^* = \frac{E[N_k]}{\sum_l E[N_l]}$$

where:

- $E[N_k]$ is the expected number of occurrences of state k in the data
- $\sum_l E[N_l]$ is the total expected number of observations

"MLE" of Conditional Multinomials with Incomplete Data

Likewise, for conditional multinomial variables with incomplete data, we can estimate the parameters as:

$$\Theta_{k|pa}^* = \frac{E\left[N_{k,pa}\right]}{E\left[N_{pa}\right]}$$

where:

- $E\left[N_{k,pa}\right]$ is the expected number of occurrences of state k given parent configuration pa in the data
- $E\left[N_{pa}\right]$ is the total expected number of observations for parent configuration pa

This form of parameter estimation is also often called as **frequentist estimation**. We treat the parameters as fixed but unknown quantities and use the observed data to estimate these parameters without incorporating prior beliefs.

Another form of parameter estimation is **Bayesian estimation**, which incorporates prior beliefs about the parameters into the estimation process.

Bayesian Estimation

The parameters (specifically Θ) are treated as random variables.

- **Prior:** Initial beliefs about the parameters before observing any data ($P(\Theta)$).
- **Posterior:** Updated beliefs using the data we observe using Bayes' theorem

$$P(\Theta|x_1, ..., x_m) = \frac{P(x_1, ..., x_m|\Theta) \cdot P(\Theta)}{P(x_1, ..., x_m)}$$

Importantly, the parameters themselves have **distributions** rather than point estimates. This allows to capture the dynamics of the parameters better by specifying a specific kind of prior distribution (e.g., Dirichlet prior for multinomial variables).

$$\theta_{x_i \mid pa_i} = \frac{\alpha(x_i, pa_i) + N(x_i, pa_i)}{\alpha(pa_i) + N(pa_i)}$$

For Bayesian estimation with a Dirichlet prior, with hyperparameters α

The biggest advantage of Bayesian estimation over frequentist estimation is **smoothing**. When we have small data sets, frequentist estimates can be unreliable, as they may assign zero probability to events that have not been observed in the data. Bayesian estimation, on the other hand, incorporates prior beliefs and can provide non-zero probabilities for unobserved events, leading to more robust estimates. Other advantages include:

- **Incorporation of Prior Knowledge:** Bayesian estimation allows us to incorporate prior knowledge or expertise into the estimation process, which can be particularly useful when data is scarce or noisy.
- **Uncertainty Quantification:** By treating parameters as random variables, Bayesian estimation provides a natural way to quantify uncertainty in the parameter estimates through the posterior distribution.
- **Convergence:** Bayesian methods often have better convergence properties, especially when using Markov Chain Monte Carlo (MCMC) methods to approximate the posterior distribution.

The choice of prior can significantly influence the results, but also how easy it is to compute the posterior distribution. For example, using conjugate priors (e.g., Dirichlet prior for multinomial variables or Beta prior for Bernoulli variables) can simplify the computation of the posterior distribution.

3 - Structure Learning

In structure learning, we want to learn the structure of the Bayesian Network itself from data. This is a more complex task than parameter estimation, as we need to search through the space of possible network structures to find the one that best fits the data.

(a) - Constraint-Based Structure Learning

The main goal of constraint-based structure learning is to find a P-Map (See **Independence Maps**).

In the approach to create a P-Map we've outlined before (**Obtaining P-Maps**), we assumed some independence relations already. But when learning from data, we don't have that knowledge beforehand, making it impossible to directly construct the P-Map.

What we need to do instead is to **test** for independence relations in the data, and then use these tested relations to construct the P-Map.

Testing Independence

- Null Hypothesis H_0: We assume the variables are independent $P(X, Y) = P(X)P(Y)$
- Test: Use procedures like χ^2 or mutual information to test the hypothesis.

Mutual Information

$$\hat{I}(X_i, X_j) = \sum_{x_i, x_j} \hat{P}(x_i, x_j) \log \frac{\hat{P}(x_i, x_j)}{\hat{P}(x_i)\hat{P}(x_j)}$$

If the tests value is below a certain threshold, we accept the null hypothesis and conclude that the variables are independent (Smaller = more likely to be independent).

This approach is not perfect, as statistical tests can be unreliable, especially with small datasets. They can lead to false positives (incorrectly identifying independence) or false negatives (failing to identify independence). Especially when an early test is wrong, it can lead to a completely incorrect structure, as the subsequent steps depend on the previous results.

The choice of significance level α (threshold) is also crucial, as it determines the sensitivity of the tests. A very low threshold may lead to many false positives, while a very high threshold may miss important independence relations.

(b) - Score-Based Structure Learning

Score-Based Structure learning takes a different approach by defining a scoring function that evaluates how well a given network structure fits the observed data. The goal is to find the structure that maximizes this score. This turns the problem into a search problem over the space of possible network structures.

The biggest question here is how to define a good scoring function. A common choice is the **Bayesian Information Criterion (BIC)**, which balances model fit and complexity:

Bayesian Information Criterion (BIC)

$$BIC(G : D) = \underbrace{M \sum_i \left(I\left(X_i, Pa_i^G\right) - H(X_i)\right)}_{\text{Fit (Likelihood)}} - \underbrace{\frac{\log M}{2} \dim(G)}_{\text{Complexity Penalty}}$$

where:

- M is the number of data points
- $I\left(X_i, Pa_i^G\right)$ is the mutual information between variable X_i and its parents in graph G
- $H(X_i)$ is the entropy of variable X_i (If a variable is very predictable, it has low entropy)
- $\dim(G)$ is the number of independent parameters in graph G

As the amount of data M increases, the BIC score increasingly favors simpler models, helping to prevent overfitting.

From then on, Score-based Structure Learning becomes a search problem, where we want to find the network structure G that maximizes the score, which in itself is NP-hard.

Usually we use **Local Search** for that:

Local Search for Structure Learning

- Start with a given network (e.g. empty network, best tree, random network).
- At each iteration:
  - Evaluate all possible changes to the current network (adding, removing, reversing edges).
  - Select the change that results in the highest score improvement.
- Stop when no further improvements can be made.

If data is complete:

- Only rescore families that have changed

If data is incomplete:

- Need to re-evaluate the entire network score (e.g. using EM algorithm for parameter estimation)

As usual, local search can easily get stuck in local optima or plateaus. To mitigate this, we can use standard heuristics like random restarts, simulated annealing, or tabu search to explore the search space more effectively.

Still, especially for incomplete data, this is very computationally expensive, as we need to perferm parameter estimation for each candidate structure during the search process.

In practice we should consider only a few candidate structures based on prior knowledge or constraints to reduce the search space.

Structural EM

We try to mitigate the computational cost doing parameter estimation for each candidate structure during the search process by combining structure learning with the EM algorithm.

- **E-Step:** Estimate the expected sufficient statistics for the missing data using the current structure
- **M-Step:**
  - Search for a structure that increases the expected complete-data log-likelihood using the expected sufficient statistics from the E-step.
  - Update the parameters for the new structure using the expected sufficient statistics.
  - Once we find a better structure, we update the model and repeat the process until convergence.

VI - Approximate Inference

We know that for a general Bayesian Network, computing the a posteriori belief of a variable is NP-hard. Even for simple cases for dynamic Bayesian Networks, exact inference can be infeasible due to the exponential growth of the state space over time.

To solve this we can use two main approaches:

- **Loopy Belief Propagation:** Passing "messages" between variables
- **Sampling:** Simulating a model many times to see what *usually* happens

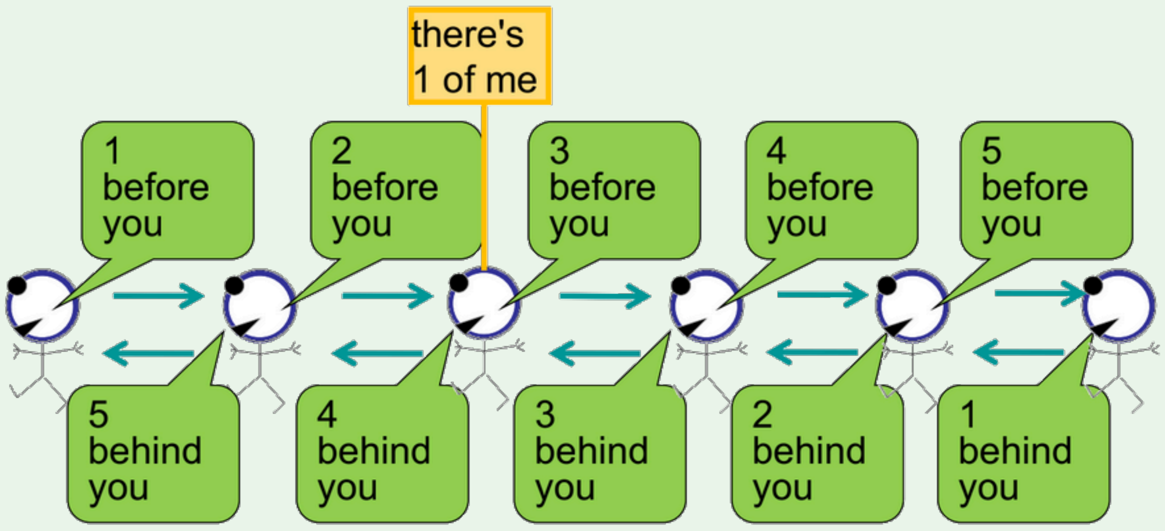
1 - Loopy Belief Propagation

Loopy Belief Propagation relies mainly on the concept of passing messages between variables in the network to update beliefs about their states:

Message Passing: Count the soldiers

An analogy to understand message passing.

- Each node represents a soldier in a line of soldiers.
- They try to count how many soldiers are in the line, while every soldier can only observe its immediate neighbors.
- **Message:** Each soldier tells its neighbors how many soldiers it believes are in front of them, and behind them.
- **Belief Update:** Reasonably only the start and end soldiers know where they stand in the line. They are the only one sure of how many people are in front of or behind them respectively. Based on that sure knowledge, their neighbors can update their beliefs to reflect that information, and so on. With the combined messages from both sides, each soldier can eventually figure out the total number of soldiers in the line. (e.g. one soldier heres "2 before you" and "3 behind you", in can conclude that there are 6 soldiers in total, including itself)



Obviously, real Bayesian Networks are more complex than a line of soldiers, but the same principles apply. Each variable sends messages to its neighbors based on its current belief and the messages it has received from other neighbors.

Though, in networks with loops, messages can circulate indefinitely, potentially leading to oscillations or convergence to incorrect beliefs.

(a) - Sum-Product Belief Propagation

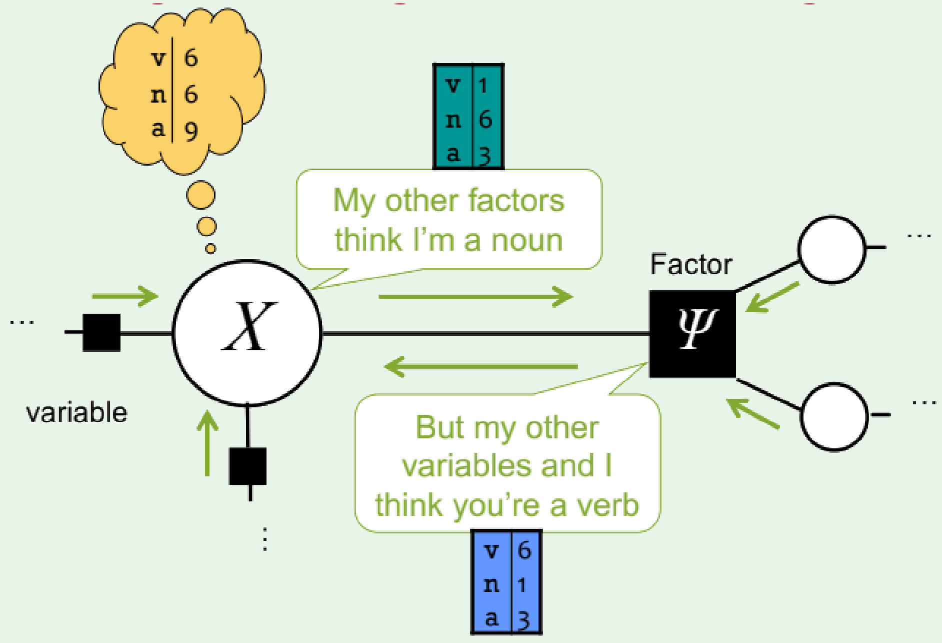
Not to be confused with Sum-Product Networks (SPNs), which in parts inspired Sum-Product Belief Propagation, but is a different concept.

For this, we usually convert the Bayesian Network into a **factor graph** representation. These graphs are made up of **variables** and **factors**.

Factor Graph

- **Variables (Circles):** Act as aggregators. They hold their belief about their own state. Variables do not have any inherent rules or probabilities associated with them, they merely "listen" to all the factors they are connected to and multiplies their opinions to form a consensus.
- **Factors [Squares]:** Act as "local experts". They store potential functions (ψ) that define how the connected variables interact with each other. Factors take the beliefs from the connected variables, apply their potential functions, and send updated messages back to the variables.

The communication between variables and factors happens through **messages**:
- **Variable to Factor:** Sends what the variable currently believes about its state, based on the messages it has received from other connected factors.
- **Factor to Variable:** Based on the potential function and the messages received from other connected variables, the factor computes a new message that reflects how it thinks the variable should update its belief.



v 6
n 6
a 9
v 1
n 6
a 3
My other factors think I'm a noun
Factor
Ψ
X
variable
…
But my other variables and I think you're a verb
v 6
n 1
a 3

Hereby, variables update their beliefs by multiplying all incoming messages from connected factors.

Variable Belief Update

$$b_{i(x_i)} = \prod_{\alpha \in \mathcal{N}(i)} \mu_{\alpha \to i}(x_i)$$

with:

- $b_{i(x_i)}$: Unnormalized belief of variable i being in state x_i
- α: Factor neighbors of variable i
- $\mu_{\alpha \to i}(x_i)$: message from factor α to variable i

Variable to Factor Message

$$\mu_{i \to \alpha}(x_i) = \prod_{\beta \in \mathcal{N}(i) \setminus \alpha} \mu_{\beta \to i}(x_i)$$

with:

- $\mu_{i \to \alpha}(x_i)$: message from variable i to factor α
- β: Factor neighbors of variable i excluding factor α
- $\mu_{\beta \to i}(x_i)$: message from factor β to variable i

Factor Belief

$$b_\alpha(x_\alpha) = \psi_\alpha(x_\alpha) \prod_{i \in \mathcal{N}(\alpha)} \mu_{i \to \alpha}(x_\alpha[i])$$

with:

- $b_\alpha(x_\alpha)$: Unnormalized belief of factor α for the configuration x_α of its connected variables
- $\psi_\alpha(x_\alpha)$: Potential function of factor α for the configuration x_α of its connected variables
- i: Variable neighbors of factor α
- $\mu_{i \to \alpha}(x_\alpha[i])$: message from variable i to factor α for the state of variable i in configuration x_α

Factor to Variable Message

$$\mu_{\alpha \to i}(x_i) = \sum_{x_\alpha : x_\alpha[i] = x_i} \psi_\alpha(x_\alpha) \prod_{j \in \mathcal{N}(\alpha) \setminus i} \mu_{j \to \alpha}(x_\alpha[i])$$

with:

- $\mu_{\alpha \to i}(x_i)$: message from factor α to variable i for state x_i
- $x_\alpha : x_\alpha[i] = x_i$: Summation over all configurations of variables connected to factor α where variable i is in state x_i
- $\psi_\alpha(x_\alpha)$: Potential function of factor α for the configuration x_α of its connected variables
- j: Variable neighbors of factor α excluding variable i
- $\mu_{j \to \alpha}(x_\alpha[i])$: message from variable j to factor α for the state of variable j in configuration x_α

Sum-Product Belief Propagation Algorithm

Input: Factor graph without cycles

Output: Exact marginal for each variable and factor

Algorithm:
1. Initialize all messages to uniform distributions:

$$\boxed{\mu_{i\to\alpha}(x_i) = 1 \qquad \mu_{\alpha\to i}(x_i) = 1}$$

2. Choose an arbitrary node as the root.
3. Send messages from the leaves to the root:
   - For each variable node, send messages to connected factor nodes using the **Variable to Factor Message** equation.
   - For each factor node, send messages to connected variable nodes using the **Factor to Variable Message** equation.
4. Compute beliefs (unnormalized marginals) for each variable and factor using the **Variable Belief Update** and **Factor Belief** equations.
5. Normalize the beliefs to obtain the final marginals.

$$\boxed{p_i \propto b_i(x_i) \qquad p_\alpha(x_\alpha) \propto b_\alpha(x_\alpha)}$$

Of course, a node can only compute its message, once it has received messages from all its other neighbors. This means that the order in which messages are sent is crucial.

Additionally, this means that this does not work for graphs with loops, as messages can circulate indefinitely, potentially leading to oscillations or convergence to incorrect beliefs. Often, it still works well in practice, but there are no guarantees. We can try to mitigate this by:
- Stopping after a fixed number of iterations
- Stopping when messages converge (i.e., change very little between iterations)
- If solution is not oscillating but converging, it *usually* is a good approximation.

2 - Sampling

Sampling methods approximate the desired probabilities by generating samples from the distribution defined by the Bayesian Network. By analyzing these samples, we can estimate marginal and conditional probabilities.

Hereby, we usually use:
- **Input:** Bayesian Network with set of node X
- **Sample:** A tuple with assigned values $s = (X_1 = x_1, X_2 = x_2, ..., X_k = x_k)$. This may include all variables (complete sample) or only a subset (partial sample).
- Ideally, samples are distributed according to $P(X|E)$

(a) - Sampling Basics

Fundamentally, we are given a set of variables that represent a joint probability distribution $\pi(X)$ and some function $g(X)$. We can then compute the expected value of $g(X)$:

$$\boxed{E_\pi g = \int g(x)\pi(X)\,\mathrm{d}x}$$

Further, given a sample S^t as an instantiation: $S^t = \{x_1^t, x_2^t, ..., x_n^t\}$ given the IID assumption and following the **Strong Law of Large Numbers**, we can approximate the expected value as:

$$\bar{g} = \frac{1}{T}\sum_{t=1}^{T} g(S^t)$$

For example:

- Given random variable X over $D(X) = \{0, 1\}$
- Given $P(X) = \{0.3, 0.7\}$
- Generate k samples: $0, 1, 1, 1, 0, 1, 1, 0, 1, 0$
  - We sample $X \leftarrow P(X)$ by:
    - Draw random number $rin[0, 1]$
    - If $r < 0.3$, set $X = 0$, else set $X = 1$
- Approximate P'(X):

$$P'(X = 0) = \frac{\#\text{samples(X=0)}}{\#\text{samples}} = \frac{4}{10} = 0.4$$
$$P'(X = 1) = \frac{\#\text{samples(X=1)}}{\#\text{samples}} = \frac{6}{10} = 0.6$$
$$P'(X) = \{0.4, 0.6\}$$

(b) - Forward Sampling

The basic idea of forward sampling is to generate samples by simulating the Bayesian Network from the root nodes down to the leaf nodes, following the direction of the edges (e.g. working through the network in topological order).

Forward Sampling Algorithm w/o Evidence

Input: Bayesian Network over variables $X = \{X_1, X_2, ..., X_n\}$, N = number of nodes, T = number of samples to generate

Output: Set of T samples $S = \{S^1, S^2, ..., S^T\}$

Algorithm:

- Process the ancestors first before descendants (topological order).

1. For each node X_i, generate a random number $r \in [0, 1]$.
2. Set x_i by mapping r to a state of X_i based on its conditional probability distribution given its parents $P(x_i \mid pa_i)$.
3. Repeat until every node has been assigned a value.

```
1. For $t = 1$ to $T$ :
2.     For $i = 1$ to $N$ :
3.         $X_i \leftarrow$ sample $x_i^t$ from $P(x_i \mid pa_i)$
```

With evidence, we need to ensure that the generated samples are consistent with the observed evidence. If a generated sample does not match the evidence, we discard it and start over.

Forward Sampling Algorithm with Evidence

Input: Bayesian Network over variables $X = \{X_1, X_2, ..., X_n\}$, E = evidence, N = number of nodes, T = number of samples to generate

Output: Set of T samples $S = \{S^1, S^2, ..., S^T\}$ consistent with evidence E

Algorithm:

- Process the ancestors first before descendants (topological order).

1. For each node X_i, generate a random number $r \in [0, 1]$.
2. Set x_i by mapping r to a state of X_i based on its conditional probability distribution given its parents $P(x_i \mid pa_i)$.

- **IF** X_i is in evidence E and $X_i \neq x_i$:
  - Discard the sample, set $i = 1$ and start over.

1. Repeat until every node has been assigned a value.

```
1. For $t = 1$ to $T$ :
2.   For $i = 1$ to $N$ :
3.     $X_i \leftarrow$ sample $x_i^t$ from $P(x_i \mid pa_i)$
4.     IF $X_i \in E$ AND $X_i \neq x_i$ :
5.       Discard $S^t, i = 1$
```

This is of course not very elegant, as we might need to discard many samples before we get a valid one, especially if the evidence is unlikely under the model.

Answering queries stays simple:

$$\boxed{\overline{P}(X_i = x_i) = \frac{\#\text{samples(X\_i = x\_i)}}{T}}$$

(c) - Gibbs Sampling

The core idea of Gibbs sampling is to start with a random assignment of values to all variables in the network, and then iteratively update the value of each variable based on the current values of its Markov blanket (its parents, children, and co-parents of its children).

We essentially sample each variable individually, while keeping the other variables fixed, and repeat this process multiple times to generate a sequence of samples that approximate the joint distribution of the variables in the network.

Gibbs Sampling Algorithm

Input: Bayesian Network over variables X, E = evidence, T = number of samples to generate

Output: Set of T samples $S = \{S^1, S^2, ..., S^T\}$ consistent with evidence E

Algorithm:

1. Fix evidence variables in E to their observed values.
2. Generate an initial assignment for all non-evidence variables (randomly or heuristically).
3. For each non-evidence variable X_i:
   - Compute the conditional distribution $P(X_i \mid mb^t(X_i))$ given its Markov blanket $mb^t(X_i)$ in iteration t.
   - Sample a new value for X_i from this conditional distribution.

4. Repeat 3./Cycle through all non-evidence variables, repeating the sampling process for a total of T iterations.

```
1. Fix $E$ in $S^0$
2. Generate initial $S^0$ from $P(X \mid E)$
3. For $t = 1$ to $T$ :
4.   For $i = 1$ to $N$ :
5.     $X_i$ sample from $P(X_i \mid mb^t(X_i))$
```

Querying can be done as before:

$$\overline{P}(X_i = x_i) = \frac{\#\text{samples(X\_i = x\_i)}}{T}$$

or by taking the average probability over all samples:

$$\overline{P}(X_i = x_i) = \frac{1}{T}\sum_{t=1}^{T} P(X_i = x_i \mid mb^t(X_i) \setminus X_i)$$

The order in which we update the variables is essentially selected arbitrarily, but we usually cycle through them in a fixed order. But since we start with a random assignment, the initial samples may not be representative of the true distribution. To mitigate this, we often discard the first few samples (burn-in period) to allow the Markov chain to converge to its stationary distribution before collecting samples for inference.

Other options to mitigate this is to sample the first samples from an approximate $P(x|e)$, with another technique (e.g. Loopy Belief Propagation or Forward Sampling).

Convergence for Gibbs sampling is generally guaranteed, as long as:
- **Irreducible:** It's possible to get from any state to any other state in a finite number of steps.
- **Aperiodic:** The system does not get stuck in cycles.
- **Ergodic:** The system explores the entire state space over time.

So essentially: If all probablities are > 0, Gibbs sampling will eventually converge to the true distribution.

But convergence can be slow, especially in high-dimensional spaces or when variables are highly correlated.

Improving Gibbs Sampling Convergence Speed

1. Reduce dependence between variables:
   - Skip samples: Only keep every k-th sample to reduce correlation between consecutive samples.
     - Wastes samples, and can increase variance.
     - But can help in reducing autocorrelation.
   - Randomize variable sampling order: Instead of cycling through variables in a fixed order, randomly select the order in each iteration.
     - Can help avoiding getting stuck in predictable patterns.
2. Reduce variance:
   - Blocking Gibbs sampling: Group correlated variables together and sample them jointly.
     - Can improve convergence and reduce variance.
     - But increases computational complexity per iteration.

- Rao-Blackwellization: Instead of sampling all variables, analytically integrate out some variables to reduce variance.
  - Can lead to more accurate estimates.
  - But requires the ability to perform analytical integration for some variables.

3. Run multiple chains in parallel:
   - Generate M chains of size K
   - Each chain produces independent estimate P_m
   - Estimate $P(x_i|e)$ as an average of $P_m(x_i|e)$:

$$\boxed{\overline{P} = \frac{1}{M}\sum_{m=1}^{M} P_m}$$

(d) - Likelihood Weighting

In likelihood weighting, we generate samples by simulating the Bayesian Network from the root nodes down to the leaf nodes, similar to forward sampling. However, when we encounter evidence variables, we do not sample them; instead, we set them to their observed values and adjust the weight of the sample accordingly. Additionally, we assign a weight to each sample based on how likely the evidence is given the sampled values of the other variables.

Likelihood Weighting Algorithm

Input: Bayesian Network over variables X, E = evidence, T = number of samples to generate

Output: Set of weighted samples S = consistent with evidence E

Algorithm:

1. Process nodes in topological order.
2. For each node X_i:
   - Set weight $w_t = 1$ for sample t.
   - **IF** X_i is not in evidence E:
     - Sample x_i from $P(x_i \mid pa_i)$.
   - **ELSE:**
     - Set x_i to the observed evidence value e_i.
     - Update weight $w_t = w_t \cdot P(e_i \mid pa_i)$.

```
1. For $t = 1$ to $T$ :
2.   For each $X_i \in X_{\text{topological}}$ :
3.     $w_t = 1$
4.     IF $X_i \notin E$ :
5.       $X_i \leftarrow$ sample $x_i$ from $P(x_i \mid pa_i)$
6.     ELSE :
7.       $X_i \leftarrow e_i$
8.       $w_t \leftarrow w_t \cdot P(e_i \mid pa_i)$
```

The weight w_t reflects how likely the evidence is given the sampled values of the other variables. This means that samples that are more consistent with the evidence will have higher weights, while those that are less consistent will have lower weights.

When answering queries, we need to take the weights of the samples into account:

$$\boxed{\overline{P}(X_i = x_i) = \frac{\sum_{t:X_i^t = x_i} w_t}{\sum_{t=1}^{T} w_t}}$$

Likelihood weighting converges to exact posterior marginals and generates samples fast. However, it increases variance, which can make convergence slower and cause many samples to be rejected with $P(x^t) = 0$.

Performance also degrades with increasing evidence variables.

VII - Dynamic Bayesian Networks (DBNs)

Normal BNs model static domains. This means that they represent a snapshot of the world at a single point in time. But many real-world domains are dynamic, meaning that they change over time.

Dynamic Bayesian Networks (BNs) extend BNs to model changing domains. In essence, they consist of of a series of "slices" representing the state of the system at different time steps, with connections between slices to represent temporal dependencies.

1 - Hidden Markov Models (HMMs)

A Hidden Markov Model (HMM) is a specific type of Dynamic Bayesian Network that models systems with hidden (unobserved) states that evolve over time. HMMs are widely used in various applications, such as speech recognition and natural language processing.

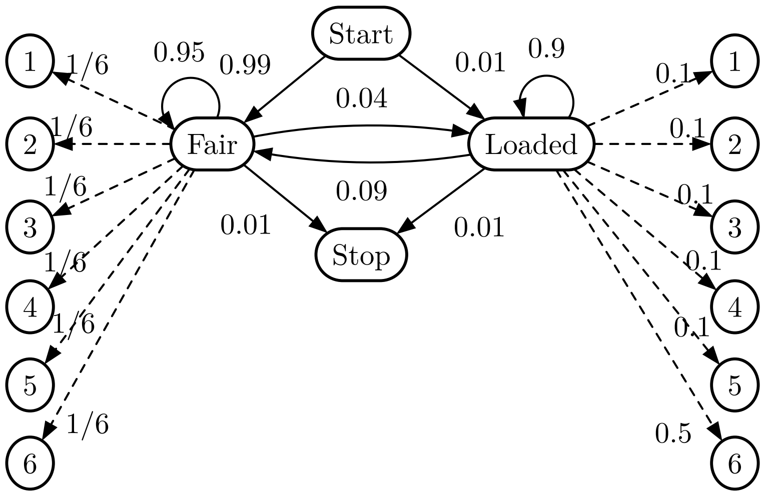
HMM Properties

An HMM is defined by two types of variables:

- **Hidden States X_t:** Variables that represent the underlying state of the system at time t. These states are not directly observable.
- **Observations O_t:** Variables that represent the observable outputs at time t, which are generated based on the hidden states.

Further, HMM has three kinds of probabilities:

- **Initial Probabilites π:** Probability of starting in a particular hidden state at time $t = 1$.
- **Transition Probabilities $a_{x_t x_{t+1}}$** Probability of moving from one hidden state to the next. Follows the **Markov Assumption**: The state at time $t + 1$ only depends on the state at time t.
- **Emission Probabilities $b_{x_t o_t}$:** Probability that a particular hidden state will produce or **emit** a specific observation.



Example HMM: A dice game where a player can use either a fair die or a loaded die, switching between them with certain probabilities

(a) - Decoding HMMs

Computes the probability of a given observation sequence $P(O \mid \mu)$.

The goal here is to see how well the model μ explains the observed data O.

Mathematically, we can sum over all possible hidden state sequences X that could have generated the observation sequence O. Doing this the usual way is computationally infeasible, as the number of possible state sequences grows exponentially with the length of the observation sequence.

Thus, we use the **Forward Procedure** to compute this efficiently using dynamic programming. Instead of summing over all possible state sequences explicitly, we recursively compute the probabilities of partial observation sequences up to each time step, storing intermediate results to avoid redundant calculations.

Forward Procedure

$$\begin{aligned} \alpha_i(t) \quad &= P(o_1, o_2, ..., o_t, x_t = i \mid \mu) \\ \Rightarrow \alpha_i(1) &= P(o_1, x_1 = i \mid \mu) = \pi_i b_{io_1} \end{aligned}$$

$$\alpha_j(t+1) = \sum_{i=1}^{N} \alpha_i(t) a_{ij} b_{jo_{t+1}}$$

$$P(O \mid \mu) = \sum_{i=1}^{N} \alpha_i(T)$$

Another way to do this is the **Backward Procedure**, which works similarly but processes the observation sequence in reverse order, from the last observation back to the first.

Backward Procedure

$$\begin{aligned} \beta_i(t) \quad &= P(o_{t+1}, o_{t+2}, ..., o_T \mid x_t = i, \mu) \\ \Rightarrow \beta_i(T) &= 1 \end{aligned}$$

$$\beta_i(t) = \sum_{j=1}^{N} a_{ij} b_{jo_{t+1}} \beta_j(t+1)$$

$$P(O \mid \mu) = \sum_{i=1}^{N} \pi_i b_{io_1} \beta_i(1)$$

Or combining both:

Combining Forward and Backward

$$P(O \mid \mu) = \sum_{i=1}^{N} \alpha_i(t) \beta_i(t)$$

(b) - Best State Sequence in HMMs

Find the state sequence that best explains the observed data O.

Individually Most Likely States

One approach is to find the **most likely state** $\hat{X}_t$ for each time step independently, using the forward-backward algorithm to compute the posterior probabilities of each state at each time step.

$$\hat{X}_t = \arg\max \gamma_i(t) = \arg\max \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}$$

Viterbi Algorithm

A more sophisticated approach is the Viterbi algorithm, which finds the **most likely sequence of states** $\hat{X}_T = \arg\max_X P(X \mid O)$ that maximizes the joint probability of the entire state sequence given the observations.

$$\delta_j(t+1) = \max_i \delta_i(t) a_{ij} b_{jo_{t+1}}$$
$$\delta_1(i) = \pi_i b_{io_1}$$
$$\psi_j(t+1) = \arg\max_i \delta_i(t) a_{ij}$$
$$\psi_1(i) = 0$$
$$\hat{X}_T = \arg\max_i \delta_i(T)$$
$$\hat{X}_t = \psi_{\hat{X}_{t+1}}(t+1)$$
$$P\left(\hat{X}\right) = \arg\max_i \delta_i(T)$$

(c) - Parameter Estimation in HMMs

Given an observation sequence O, we want to find the model that is most likely to produce the observed data. Since there isn't an analytical solution for this, this problem becomes: Given an initial (random or prior beliefs based) model μ and observation sequence O, update the model parameters to maximize $P(O \mid \mu)$.

Baum-Welch Algorithm

We can use the Baum-Welch algorithm, which is a specific instance of the Expectation-Maximization (EM) algorithm tailored for HMMs. The algorithm iteratively refines the model parameters to maximize the likelihood of the observed data.

Arc Traversal Probability

$$p_t(i,j) = \frac{\alpha_i(t) a_{ij} b_{jo_{t+1}} \beta_j(t+1)}{\sum_{m=1}^{N} \alpha_m(t)\beta_m(t)}$$

State Occupation Probability

$$\gamma_i(t) = \sum_{j=1}^{N} p_t(i,j)$$

Parameter Re-estimation

$$\hat{\pi}_i = \gamma_i(1)$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} p_t(i,j)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

$$\hat{b}_{ik} = \frac{\sum_{t:o_t=k} \gamma_i(t)}{\sum_{t=1}^{T}} \gamma_i(t)$$

For this algorithm it holds that:

$$\boxed{P(O \mid \hat{\mu}) \geq P(O \mid \mu)}$$

It isn't perfect, as it can get stuck in local maxima or saddle points, but is usually effective.

2 - Kalman Filters

While HMMs are great for modeling systems with discrete hidden states, many real-world systems have continuous hidden states. Kalman Filters are designed to handle such scenarios, making them suitable for applications like tracking objects in motion or estimating the state of a dynamic system.

Kalman Filter Properties

A Kalman Filter models the system using two main equations:

- **State Transition Model:** Describes how the hidden state evolves over time, typically represented as a linear transformation of the previous state plus some process noise.

$$\boxed{x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t}$$

- **Observation/Measurement Model:** Describes how the observed measurements relate to the hidden state, also typically represented as a linear transformation of the hidden state plus some measurement noise.

$$\boxed{z_t = C_t x_t + \delta_t}$$

Further, Kalman Filters assume that both the process noise and measurement noise are Gaussian distributed, which allows for efficient computation of the posterior distributions.

Kalman Filter Components

- **State Transition Matrix A_t $(n \times n)$**: Defines how the state evolves from time $t-1$ to time t.
- **Control Input Matrix B_t $(n \times l)$**: Defines how control inputs u_t influence the state transition.

- **Observation Matrix C_t ($k \times n$)**: Defines how the hidden state maps to the observed measurements.
- **Measurement Noise Covariance R_t ($k \times k$)**: Represents the uncertainty in the observation model.
- **Process Noise Covariance Q_t ($n \times n$)**: Represents the uncertainty in the state transition model.
- **Measurement Noise δ_t**: Represents the noise in the observed measurements.
- **Process Noise ε_t**: Represents the noise in the state transition.

Kalman Filter Initialization and Belief Update

Initial belief is represented by:

$$\operatorname{bel}(x_0) = N(x_0; \mu_0, \Sigma_0)$$

with:

- x_0: Initial state
- μ_0: Initial state estimate e.g. starting position and velocity
- Σ_0: Initial estimate covariance e.g. uncertainty in starting position and velocity

The Kalman Filter operates in three main steps: prediction, observation and update.

- **Prediction Step:** Uses the state transition model to predict the next state and its uncertainty.

$$\boxed{\begin{gathered}\overline{\operatorname{bel}}(x_t) = N\left(x_t; \overline{\mu}_t, \overline{\Sigma}_t\right) \\ \overline{\mu}_t = A_t \mu_{t-1} + B_t u_t \\ \overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t\end{gathered}}$$

- **Observation Step:** Observe the measurement z_t at time t.

$$\boxed{z_t = C_t x_t + \delta_t}$$

- **Update Step:** Incorporates the new measurement to refine the state estimate and reduce uncertainty.
  - Compute Kalman Gain:

$$\boxed{K_t = \overline{\Sigma}_t C_t^T \left(C_t \overline{\Sigma}_t C_t^T + Q_t\right)^{-1}}$$

  - Update State Estimate:

$$\boxed{\mu_t = \overline{\mu}_t + K_t (z_t - C_t \overline{\mu}_t)}$$

  - Update Estimate Covariance:

$$\boxed{\Sigma_t = (I - K_t C_t) \overline{\Sigma}_t}$$

Then the posterior belief is:

$$\boxed{\operatorname{bel}(x_t) = N(x_t; \mu_t, \Sigma_t)}$$

This algorithm runs recursively, updating the state estimate and uncertainty at each time step as new measurements are observed. Kalman Filters are optimal for linear systems with Gaussian noise, providing efficient and accurate state estimation in real-time applications.

Kalman Filters, however, assume linearity and Gaussian noise. So they cannot always be used for general Dynamic Bayesian Networks.

3 - General DBNs

Unfortunately, exact inference in general DBNs is intractible, meaning we can generally only use approximate inference methods.

One such method is using **Factored Belief States**. Hereby we intentionally ignore some dependencies between variables (e.g. $P(A, B) \approx P(A)P(B)$) to keep the representation manageable.

VIII - Tractable Probabilistic Models

Bayesian Networks (Graphical views of direct dependencies) and Markov Random Field (MRF) (Graphical views of correlations) are both powerful, but inference in them is often **intractable**. This means that there are essentially too many computations needed to answer queries, making them impractical for large-scale problems.

To address this, we can use **Tractable Probabilistic Models**, (like Sum-Product Networks (SPNs)) which are designed to allow for efficient inference while still capturing important probabilistic relationships.

As such they form a middle ground between "deep learning" models e.g. neural networks, and simple, shallow probabilistic models e.g. Naïve Bayes.

- Deep Learning Models:
  - Represent computations as a series of layers
  - Can capture complex patterns
  - Often no probabilistic semantics
  - Extensive efforts for learning
- Shallow Probabilistic Models:
  - Clear probabilistic semantics
  - Often simpler and more interpretable
  - Inference can be intractable

1 - Sum-Product Networks (SPNs)

Not to be confused with Sum-Product Belief Propagation, which is inspired by SPNs, but works on factor graphs, built from Bayesian Networks or Markov Random Fields.

A Sum-Product Network (SPN) is a type of probabilistic graphical model that represents joint probability distributions as a **rooted directed acyclic graph (DAG)**.

SPN Components

An SPN consists of three types of nodes:

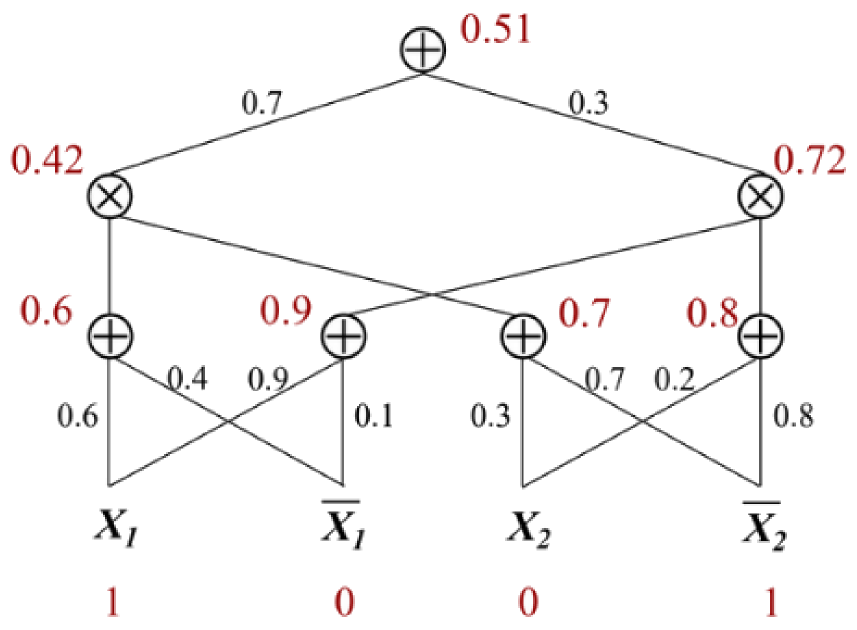
- **Leaf Nodes (Indicators):** Inputs at the bottom of the graph. For each discrete variable X_i with domain $D(X_i)$, there are $|D(X_i)|$ indicator nodes.
- **Sum Nodes ($\times$):** Represent mixtures of features. Mathematically, they compute the weighted sum of their child nodes.
- **Product Nodes ($+$):** Represent conjunctions of features. Mathematically, they multiply the weighted sum of their child nodes.

The structure of an SPN allows for efficient computation of marginal and conditional probabilities through a process of upward and downward passes through the network.

Properties of SPNs

1. **Completeness:** Every sum node has children that depend on the same set of variables.
2. **Decomposability:** For every product node, the sets of variables that its children depend on are disjoint.
3. **Consistency:** For every sum node, the weights of its outgoing edges sum to 1.

These properties ensure that SPNs can represent valid probability distributions and allow for efficient inference.



Inference in SPNs

Inference in SPNs is simple:

- **Marginal Inference ($P(E)$):**
  1. Set Leaf values:
     - For each evidence variable X_i with observed value x_i, set the corresponding indicator node to 1, and all other indicator nodes for X_i to 0.
     - For non-evidence variables, set **all** indicator nodes to 1.
  2. Upward Pass:
     - Starting from the leaf nodes, compute the values of all sum and product nodes up to the root node.
       - Due to the properties that all weights of sum nodes sum to 1, the unobserved variables effectively marginalize out.
  3. The value at the root node represents the marginal probability $P(E)$.
- **Conditional Inference $\left(P(Q|E) = \left(\frac{P(Q,E)}{P(E)}\right)\right)$:**
  1. Find $P(E)$:
     - Perform marginal inference as described above to compute $P(E)$. Ergo set the indicators for evidence E to their observed values, and all others to 1, including query Q.
     - Evaluate the SPN to get $P(E)$.
  2. Find $P(Q, E)$:
     - As with $P(E)$, set the indicators for evidence E to their observed values. Here, additionally set the indicators for query variables Q to their observed values as well.
     - Evaluate the SPN to get $P(Q, E)$.
  3. Compute $P(Q|E)$:
     - Finally, compute the conditional probability using the formula $P(Q|E) = \frac{P(Q,E)}{P(E)}$.

Mind that for soft evidence or continous variables, we can set the indicator nodes to the corresponding probabilities or likelihoods instead of binary values.

Should we require **Maximum a Posteriori (MAP)** inference, we can modify the upward pass:

- Replace sum nodes with max nodes, which select the maximum value among their children instead of summing them.

Thus inference in SPNs is incredibly efficient as marginals can be computed in time linear to the size of the network with simple mathematical operations. This is because the structure of the SPN allows for a systematic way to compute probabilities by traversing the network, combining the contributions of sum and product nodes in a way that avoids redundant calculations.

SPNs are especially useful as they compactly represent partition functions (normalizing constants) which are often intractable in other probabilistic models.

IX - Deep Generative Models

Deep Generative Models are a class of models that learn to generate new data samples that are similar to a given dataset. They are called “deep” because they typically involve multiple layers of neural networks, and “generative” because they can generate new data points.

1 - Supervised vs. Unsupervised Learning

Supervised Learning

- **Input:** Labeled data (input-output pairs) (x, y) with x being the data and y the label
- **Goal:** Learn a mapping from inputs to outputs, $f : x \to y$

As such it focuses on **discriminating** the different classes or outputs based on the input data / finding the decision boundary between them.

This is typically used for tasks like classification, regression, detection, etc.

Unsupervised Learning

- **Input:** Unlabeled data x
- **Goal:** Learn the underlying structure or distribution of the data

As such it focuses on **modeling** the data distribution $P(x)$ or finding patterns in the data. Using these learned patterns it is possible to **generate** new data samples similar to the input data.

This is typically used for tasks like clustering, dimensionality reduction, density estimation, etc.

These provide different statistical model learning paradigms, which are commonly used in:

- **Discriminative Models:** Models that learn the conditional probability distribution $P(y|x)$ directly from the data. They focus on modeling the decision boundary between different classes or outputs. Examples include logistic regression, support vector machines, and conditional random fields.
- **Generative Models:** Models that learn the joint probability distribution $P(x, y)$ of the input data and labels. They focus on modeling the underlying data distribution and can generate new samples. Examples include Gaussian Mixture Models, Hidden Markov Models, and Variational Autoencoders.

We primarily focus on generative models here.

2 - Generative Models

Generative models are a class of models that learn to generate new data samples that are similar to a given dataset. They can be used for various tasks, such as image generation, text generation, and data augmentation.

We can categorize generative models into two main types:

- **Explicit Density Models:** Define a functional form for the probability distribution $p_{\text{model}}(x)$ and try to solve for it. Essentially trying to find the exact mathematical “formula” to describe the data.

- **Implicit Density Models:** Does not try to find an explicit formula for the probability distribution. Instead, they learn to generate samples from the distribution without explicitly modeling it. Essentially trying to learn how to "imitate" the data generation process.

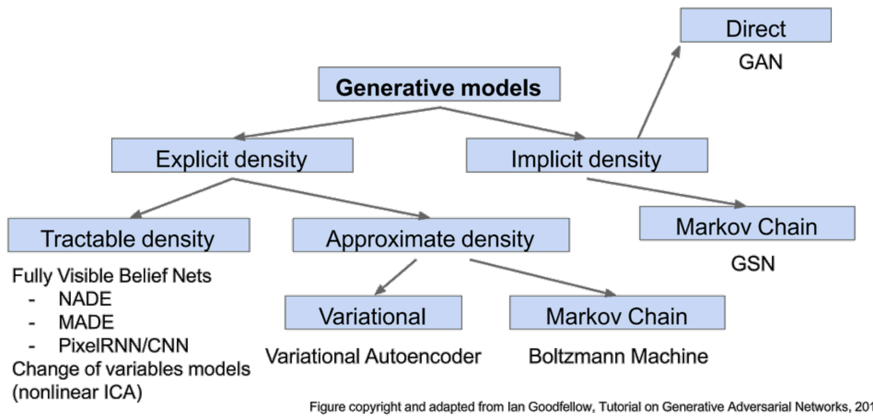
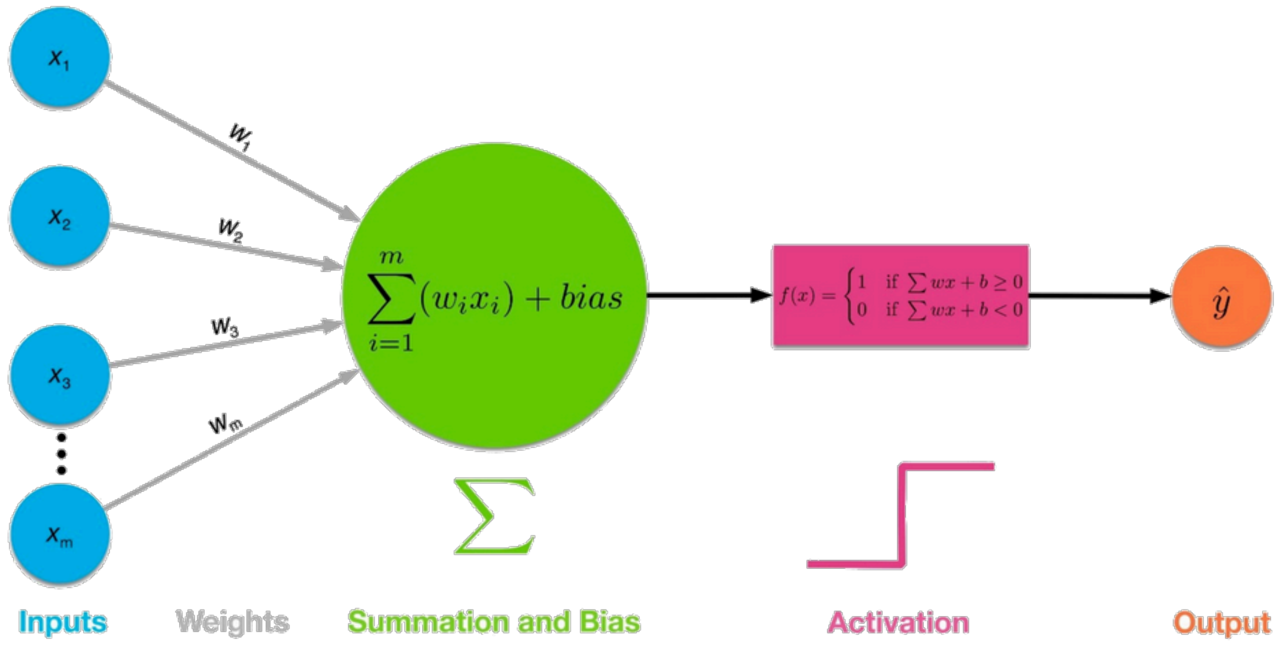


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

3 - Neural Networks

Most generative models we see today are based on neural networks, which are powerful function approximators that can learn complex patterns in data.

A neural network consists of layers of interconnected nodes (**neurons**):


x_1
x_2
x_3
x_m
w_1
w_2
w_3
w_m
$\sum_{i=1}^{m}(w_i x_i) + bias$
$f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases}$
$\hat{y}$
$\sum$
Inputs
Weights
Summation and Bias
Activation
Output

These neurons usually work the same in each nueral network, with the exception of the activation function.

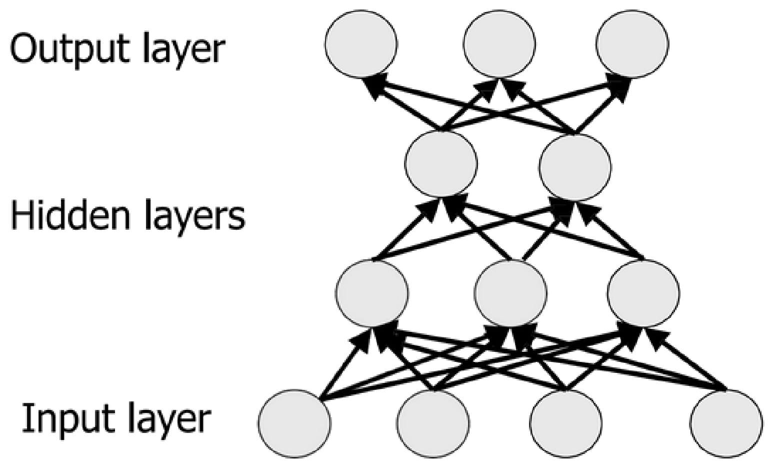
Activation Function

The activation function is a non-linear function applied to the output of each neuron. It allows the network to learn complex, non-linear relationships in the data. Common activation functions include:

- **ReLU (Rectified Linear Unit):** $f(x) = \max(0, x)$
- **Sigmoid:** $f(x) = \frac{1}{1+e^{\{-x\}}}$
- **Tanh (Hyperbolic Tangent):** $f(x) = \frac{e^x - e^{\{-x\}}}{e^x + e^{\{-x\}}}$

In a conventional neural network, we have one input layer, one hidden layer, and one output layer. While this is already able to approximate many functions, its “width” scales exponentially with the complexity of the function we want to learn. Thus, we often use **deep** neural networks, which have multiple hidden layers, allowing them to learn more complex functions with fewer neurons.

For the layers, it essentially refines the data through each layer, so the “higher” we get in the layers, the more refined the data becomes. In essence, the neural network learns simple features first (e.g. edges in an image), and then combines these simple features to learn more complex features (e.g. shapes, objects) in the deeper layers.



Training:

The goal of training a neural network is to find the optimal weights that minimize the difference between the predicted output and the true output (the loss). This is typically done using an optimization algorithm called **gradient descent**.

Gradient Descent

Goal is to find a set of weights W that minimizes the loss function $J(W)$, which represents the difference between the network's prediction $(\hat{y})$ and the actual target (y). The loss function can be defined in various ways, such as mean squared error for regression tasks or cross-entropy loss for classification tasks.

1. **Initialize weights randomly**.
2. **Until convergence**:
   (a) **Compute gradient**: $\frac{\partial J(W)}{\partial W}$
   (b) **Update weights**: $W := W - \alpha \frac{\partial J(W)}{\partial W}$ (with learning rate α)
3. **Return** optimized weights W

Backpropagation

Backpropagation is the algorithm used to compute the necessary gradients for gradient descent. It works by applying the chain rule of calculus to compute the gradient of the loss function with respect to each weight in the network, starting from the output layer and moving backward through the hidden layers to the input layer.

1. **Forward pass**: Compute the output of the network for a given input and calculate the loss.
2. **Backward pass**: Compute the gradient of the loss with respect to each weight by propagating the error backward through the network.
   - For the output layer:

   $$\boxed{\frac{\partial J}{\partial w_L} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_L}}$$

   - For all other layers:

   $$\boxed{\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i} \frac{\partial z_i}{\partial w_i} = \text{error at current neuron} \times \text{output of previous layer}}$$
3. **Update weights**: Use the computed gradients to update the weights using gradient descent.

(a) - PixelRNN

PixelRNN is a generative model designed for image generation. It models the joint distribution of pixel values in an image using a **recurrent neural network (RNN)**, specifically an **Long Short-Term Memory (LSTM)** architecture.

Recurrent Neural Networks (RNNs)

RNNs are an extension of traditional neural networks. In essence, they are **NNs with loops**.

An RNN possesses additional hidden state varaiables that allow for storage of inormation across time steps. This makes them particularly well-suited for sequential data, such as time series or natural language, where the order of the data points matters.

Long Short-Term Memory (LSTM)

LSTMs are a specific type of RNN that are designed to address the vanishing gradient problem, which can occur in traditional RNNs when trying to learn long-term dependencies. LSTMs achieve this by introducing a more complex architecture with:

- **Cell State:** A memory that can carry information across many time steps. The cell state is regulated by three gates:
  1. **Forget Gate:** Determines what information to discard from the cell state.
  2. **Input Gate:** Determines what new information to add to the cell state.
  3. **Output Gate:** Determines what information to output from the cell state.
- **Hidden State:** Represents the output of the LSTM at each time step, which can be used for making predictions or as input to the next time step.

In essence, PixelRNN tries to model:

$$\underbrace{P(x)}_{\substack{\text{Likelihood}\\ \text{of image x}}} = \prod_{i=1}^{n} \underbrace{P(x_i \mid x_1, x_2, ..., x_{i-1})}_{\substack{\text{Probability of } i\text{-th pixel}\\ \text{given all previous pixels}}}$$

Which means, that PixelRNN assumes the probability of an image to be the product of the probabilities of each pixel given all the previous pixels.

Of course, this needs a specific ordering. We usually start from the top-left corner and "spread-out" in a raster scan order (left to right, top to bottom).

Training PixelRNNs is essentially done by maximizing the likelihood: We show the model images and let it predict the next pixel given the previous pixels. We then compare the predicted pixel with the actual pixel and update the model parameters to minimize the difference (loss) between them.

(b) - PixelCNN

PixelCNN is in purpose similar to PixelRNN, as it also models the joint distribution of pixel values in an image. However, instead of using a recurrent architecture, PixelCNN uses a **convolutional neural network (CNN)** architecture.

Convolutional Neural Networks (CNNs)

Unlike traditional neural networks, CNNs are designed to process data with a grid-like topology, such as images.

They use a **filter / kernel** to scan across small regions of the input, allowing for local feature extraction. This is particularly effective for image data, as generally local patterns (e.g. edges, textures) are more important than global patterns for understanding the content of an image.

Additionally, due to the filter being shared across the entire input, CNNs have fewer parameters than fully connected networks, which improves efficiency of computation and reduces the parameter space, making them less prone to overfitting.

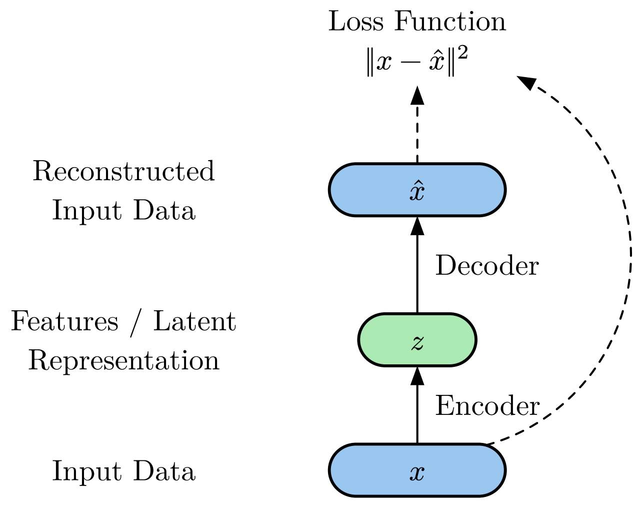
Of course, due to how the chain rule works, we can only ever use information from the "previous" pixels, as we don't know "future" pixels yet. Thus, we use **masked convolutions**. Hereby, we simply set the weights of "future" pixels (right and below) to 0, so that they do not contribute to the output of the convolution.

The rest essentially works the same as PixelRNN, as we train the model by maximizing the likelihood of the training images, and we can generate new images by sampling pixels sequentially from the top-left corner to the bottom-right corner.

(c) - Variational Autoencoders (VAEs)

An **autoencoder** is a generative model that learns to encode data into a lower-dimensional latent space and then decode it back to the original data space. It consists of two main components:

- **Encoder:** A neural network that takes input data and maps it to a latent representation (a lower-dimensional space). The encoder learns to compress the input data while preserving important features.
- **Decoder:** A neural network that takes a sample from the latent space and maps it back to the original data space. The decoder learns to reconstruct the original data from the compressed representation.

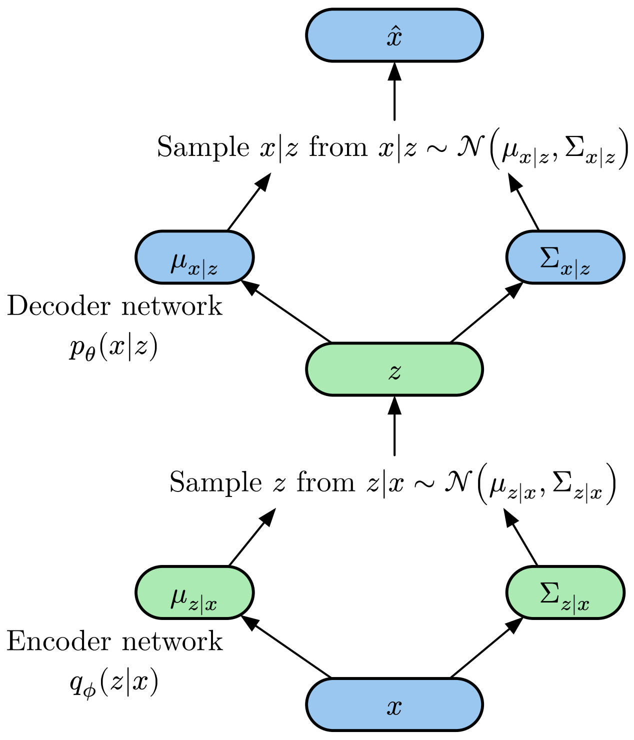


So the encoder learns to compress the input data into a latent representation, while the decoder learns to reconstruct the original data from this latent representation. The goal is to minimize the reconstruction loss, which is typically measured as the mean squared error between the original input and the reconstructed output.

This structure is quite handy, as it can be used for both **generating** using the decoder, and **inferring** using the encoder, after they have been trained.

The encoder can be used like a standard supervised learning model, where we input data and get a latent representation. The decoder can be used to generate new data by sampling from the latent space and passing it through the decoder to get a new data point in the original data space.

A **variational autoencoder** is a specific type of autoencoder that introduces a probabilistic framework to the latent space. While a standard autoencoder learns a deterministic mapping from the input to the latent space and back, a VAE learns a distribution over the latent space (e.g. mean μ and covariance Σ of a Gaussian distribution).


$\hat{x}$
Sample $x|z$ from $x|z \sim \mathcal{N}\left(\mu_{x|z}, \Sigma_{x|z}\right)$
$\mu_{x|z}$
$\Sigma_{x|z}$
Decoder network
$p_\theta(x|z)$
z
Sample z from $z|x \sim \mathcal{N}\left(\mu_{z|x}, \Sigma_{z|x}\right)$
$\mu_{z|x}$
$\Sigma_{z|x}$
Encoder network
$q_\phi(z|x)$
x

The major difference between a VAE and a standard autoencoder is how we “evaluate” it, or what our goal is. In a standard autoencoder we use a simple loss function that measures the reconstruction error.

In a VAE, we cannot directly optimize the likelihood of the data, as it involves an intractable integral over the latent variables:

$$\boxed{\log P(x) = \log \int P(x|z)P(z)\,\mathrm{d}z}$$

Instead, we optimize a lower bound on the log-likelihood, known as the **Evidence Lower Bound (ELBO)**. The ELBO is a term for which:

$$\boxed{\log P(x) \geq ELBO = \mathcal{L}(x^i, \theta, \phi)}$$

The ELBO consists of two main components:

$$\boxed{ELBO = \underbrace{E_z[\log p_\theta(x^i|z)]}_{\text{Reconstruction}} - \underbrace{D_{KL}(q_\phi(z \mid x^i) \parallel p_\theta(z))}_{\text{Divergence}}}$$

- The **reconstruction term** encourages the model to learn to reconstruct the input data accurately from the latent representation. The better the reconstruction, the higher this term will be.
- The **divergence term** encourages the learned latent distribution $q_\phi(z \mid x^i)$ to be close to the prior distribution $p_\theta(z)$. This regularizes the latent space and prevents overfitting by ensuring that the latent representations do not deviate too much from a known distribution (e.g. standard normal distribution).

So for training we essentially follow:

$$\boxed{\theta^*, \phi^* = \arg\max_{\theta,\phi} \sum_{i=1}^{N} \mathcal{L}(x^i, \theta, \phi)}$$

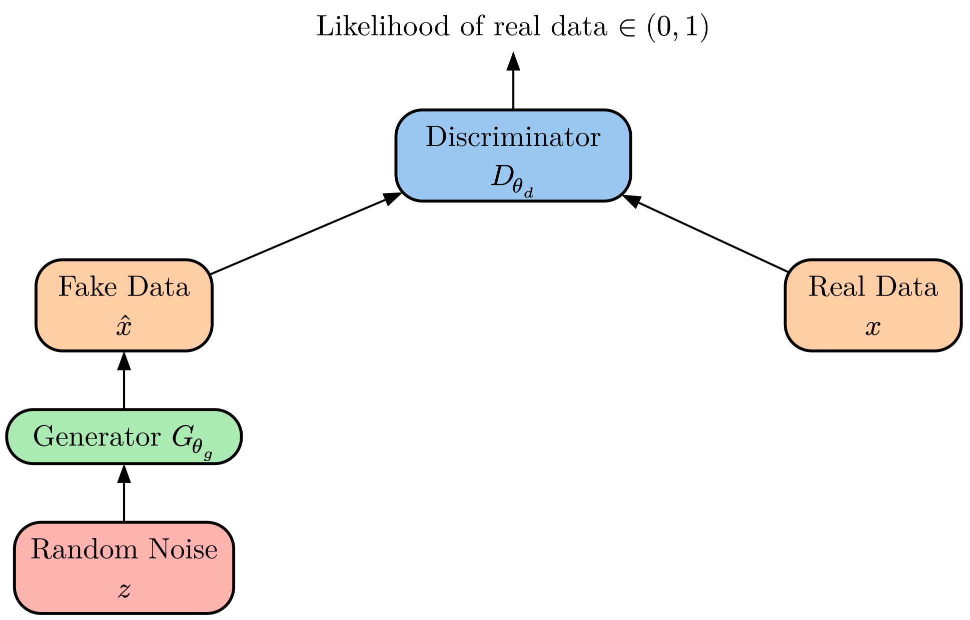
Generation with a VAE is straightforward: We can sample from the prior distribution $p_\theta(z)$ to get a latent representation, and then pass this through the decoder to generate a new data point in the original data space.

(d) - Generative Adversarial Networks (GANs)

GANs, in contrast to the previous models, are **implicit density models**. They simply learn to generate samples from the data distribution without explicitly modeling it.

A GAN takes a game-theoretic approach to generative modeling, where two neural networks, the **generator** and the **discriminator**, are trained simultaneously in a competitive setting. The two “players” have their own objectives:

- **Generator:** Produce fake data samples to fool the discriminator into classifying them as real. The generator learns to map random noise (from a latent space) to the data space, effectively learning to generate new data samples that resemble the training data.
- **Discriminator:** Distinguish between real data samples (from the training set) and fake data samples (produced by the generator). The discriminator learns to output a probability indicating whether a given sample is real or fake.



To train this model, we let the two models compete in a minimax game:

$$\boxed{\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x\sim p_{\text{data}}} \log D_{\theta_d}(x) + \mathbb{E}_{z\sim p(z)} \log\left(1 - D_{\theta_d}\left(G_{\theta_g}(z)\right)\right)\right]}$$

So:

- **Discriminator** wants to maximize the function, such that $D(x)$ is close to 1 (real), and $D(G(z))$ is close to 0 (fake).
- **Generator** wants to minimize the function, such that $D(G(z))$ is close to 1.

GANs Training Algorithm

- **for** number of training iterations **do**:
  1. **for** k steps **do**:
     (a) Sample batch of m noise sample $\{z^1, ..., z^m\}$ from noise prior $p(z)$
     (b) Sample batch of m real data samples $\{x^1, ..., x^m\}$ from data distribution $p_{\text{data}}(x)$
     (c) Update the discriminator by ascending its stochastic gradient:

$$\boxed{\nabla_{\theta_d}\left(\frac{1}{m}\right)\sum_{i=1}^{m}\left[\log D_{\theta_d}(x^i) + \log\left(1 - D_{\theta_d}\left(G_{\theta_g}(z^i)\right)\right)\right]}$$

  2. Sample batch of m noise sample $\{z^1, ..., z^m\}$ from noise prior $p(z)$
  3. Update the generator by ascending its stochastic gradient:

$$\boxed{\nabla_{\theta_g}\left(\frac{1}{m}\right)\sum_{i=1}^{m}\log\left(D_{\theta_d}\left(G_{\theta_g}(z^i)\right)\right)}$$

After we have completed training, we can generate new data samples by sampling from the noise prior $p(z)$ and passing it through the generator to get a new data point in the original data space.

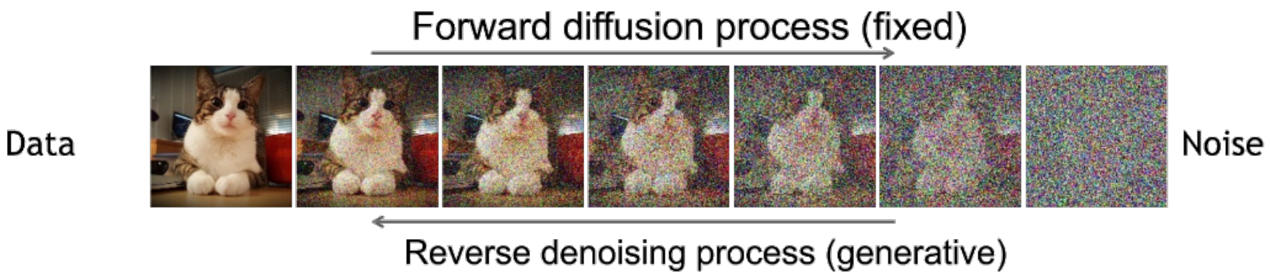
(e) - Diffusion Models

The core idea of diffusion models is to model the data generation process as a gradual transformation from noise to data. This is achieved through a two-step process:

1. **Forward Diffusion Process:** This process gradually adds noise to the data over a series of time steps, effectively transforming the data into pure noise. The forward process is typically defined as a Markov chain, where at each time step, a small amount of Gaussian noise is added

to the data. Thus, the model learns how a small amount of noise affects the data, and how to reverse that small amount of noise to get back to the original data.

2. **Reverse Diffusion Process:** This process learns to reverse the forward diffusion process, effectively denoising the data step by step until it reaches the original data distribution. The reverse process is also defined as a Markov chain, where at each time step, the model learns to predict the noise that was added in the forward process and subtract it from the noisy data to get a less noisy version of the data.


Forward diffusion process (fixed)
Data
Noise
Reverse denoising process (generative)

Forward Diffusion Process

1. Start with a real data sample x_0 from the data distribution $p_{\text{data}}(x)$.
2. For each time step t from 1 to T:
   - Add a small amount of Gaussian noise to the data sample from the previous time step:

$$q(x_t \mid x_{t-1}) = \mathcal{N}\left(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I\right)$$

with:

- β_t: Noise scheduler. Ensures that at the final time step T, the data is transformed into pure noise.

- At the final time step T, the data sample x_T is essentially pure noise, as it has been corrupted by noise at each time step.

Instead of going through it step by step, we can also directly sample from the distribution at any time step t given the original data sample x_0:

$$q(x_t \mid x_0) = \mathcal{N}\left(x_t; \sqrt{\overline{\alpha}_t}x_0, (1-\overline{\alpha}_t)I\right)$$

with:

- $\overline{\alpha}_t$: $\prod_{i=1}^{t}(1-\beta_i)$. This term captures the overall effect of the noise added up to time step t.

Reverse Denoising Process

1. Start with a noise sample x_T from the noise distribution (e.g. standard normal distribution).

$$\boxed{x_T \sim \mathcal{N}(0, I)}$$

2. Iteratively sample x_{t-1}

$$\boxed{x_{t-1} \sim q(x_{t-1} \mid x_t)}$$

3. Repeat this process until we reach x_0, which should be a sample from the original data distribution.

X - Relational Probabilistic Models

Most traditional machine learning models assume that data points are i.i.d. (independent and identically distributed). However, in many real-world scenarios, data points are not independent, but rather have complex relationships with each other.

To capture these relationships, we need models that connect predicate logic with probabilistic reasoning.

1 - Relations

There are several types of relations and different views on what relational learning is.

Relations as Correlations / Regularizations

At a simple level, relations are a signal that two things are connected in a way. For example, if two words appear often together in a context, there must be some semantic relationship between them.

Extensional Relations (Data-Only)

Here, we look at how things are related mathematically. For example:

- **Covariance Functions**
- **Distance Functions**
- **Graphs**
- **Tensors**

Hypergraph Extensional Relations

Here, we look at how things are related mathematically, but with a focus on higher-order relationships. For example:

- **Hypergraphs**: A generalization of graphs where edges can connect more than two vertices, allowing for modeling of complex relationships between multiple entities.
- **Random Walks with Restarts as similarity measure**

Relations as Symmetries / Redundancies

Relations can also be seen as symmetries or redundancies in the data. We say that two data points are symmetrical or redundant if they behave in the same way under certain transformations.

This enables us to use the same model parameters for different data points, which can improve generalization and reduce the number of parameters needed to model the data.

Relations as Logical Dependencies

Looks at multiple (typed) relations between entities, which can be represented using predicate logic. For example:

- **First-order Logic (FOL):** A formal system that allows for the expression of relationships between objects using quantifiers and predicates. It can represent complex relationships and is widely used in knowledge representation and reasoning.

- FOL can express relationships, from which we can derive new relationships using logical inference. For example, if we know that "All humans are mortal" and "Socrates is a human", we can infer that "Socrates is mortal".
- Or more formal:

$$\boxed{\begin{aligned}\forall x : \text{Human}(x) &\Rightarrow \text{Mortal}(x)\\ &\land \text{Human}(\text{Socrates})\\ &\Rightarrow \text{Mortal}(\text{Socrates})\end{aligned}}$$

2 - Markov Logic Networks (MLNs)

In traditional logical systems, such as first-order logic, we only have **hard constraints**, meaning that a statement is either true or false. If a world violates even a single constraint, it is considered impossible.

This can be too rigid for many real-world applications, where we need **soft constraints** that allow for some degree of uncertainty. If a world violates a soft constraint, it does not become impossible, but rather just less likely.

Markov Logic Networks (MLNs) are a framework that combines first-order logic with probabilistic graphical models, specifically Markov networks.

A MLN is defined as a set of pairs (F, w):
- F: A formula in first-order logic, which represents a logical constraint or relationship between entities.
- w: The weight of the formula as a real number. The higher the weight, the stronger the constraint. The value implies:
  - **Hard Constraint** $w \to +\infty$:
    - If the formula is violated, the world becomes impossible.
  - **Soft Constraint** $w > 0$:
    - If the formula is violated, the world becomes less likely, but not impossible.
  - **No Constraint / Prohibition** $w = 0$:
    - The formula has no effect on the probability of the world.
  - **Soft Prohibition** $w < 0$:
    - If the formula is satisfied, the world becomes less likely.
  - **Hard Prohibition** $w \to -\infty$:
    - If the formula is satisfied, the world becomes impossible.

The probability of a world (a specific assignment of truth values to all possible ground atoms) is given by:

$$\boxed{P(X = x) = \frac{1}{Z} \exp\left(\sum_i \underbrace{w_i}_{\substack{\text{weight of formula}\\ i}} \underbrace{n_i(x)}_{\substack{\text{True groundings of}\\ \text{formula } i \text{ in } x}}\right)}$$

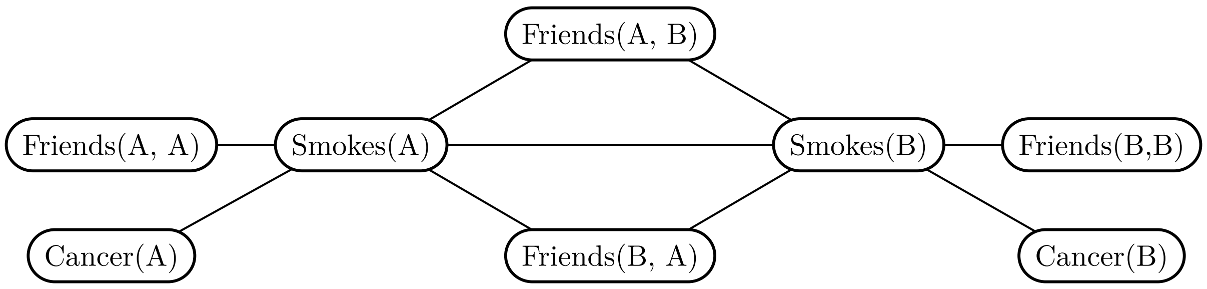
By "true grounding" we mean the number of instances of the formula that are satisfied in the world x. For example, if we have a formula "Smokes(x) => Cancer(x)" and in the world x we have 3 people who smoke and have cancer, then the number of true groundings for this formula would be 3.

The nodes in an MLN represent the ground atoms. So essentially for each combination of constant and predicate, we have one node in the MLN. The edges represent the dependencies between these ground atoms, which are determined by the formulas in the MLN.

Formulas:

$$\forall x : \text{Smokes}(x) \Rightarrow \text{Cancer}(x)$$
$$\forall x, y : \text{Friends}(x, y) \Rightarrow (\text{Smokes}(x) \Leftrightarrow \text{Smokes}(y))$$

Constants: Anna **(A)** and Bob **(B)**



If we add some weights to the formulas, we can compute any probability of interest using graphical model techniques (e.g. use Markov Properties, variable elimination, d-separation, etc.).

Conditional probabilities can be computed using the different rules of probability, such as:
- **Bayes' Theorem:** $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$
- **Chain Rule:** $P(A \mid B) = \frac{P(A,B)}{P(B)}$
- **Marginalization:** $P(A) = \sum_B P(A, B)$

3 - Distribution Semantics

The distribution semantics is a framework for defining the semantics of probabilistic logic programs. It provides a way to assign probabilities to logical statements and to compute the probability of a query given a set of probabilistic facts and rules.

An example: A gambling game with a coin throw and two draws from two urns:

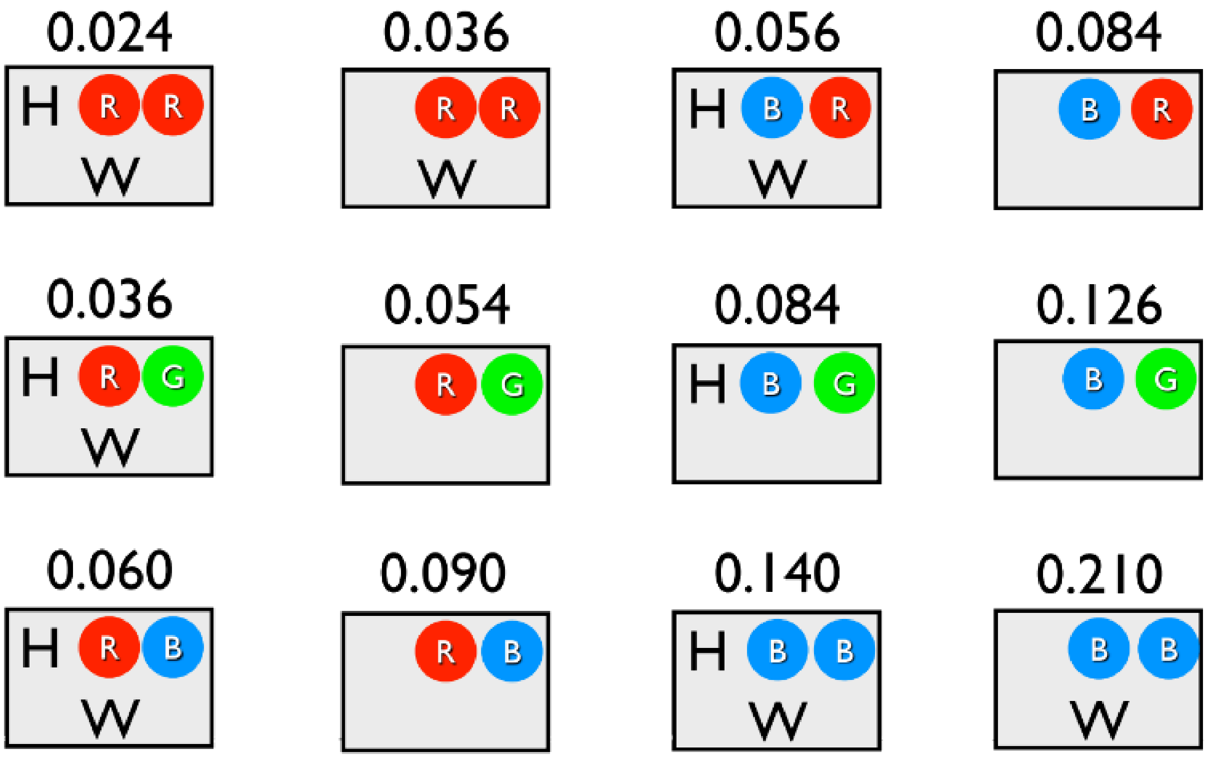
```
0.4::coin(heads).
0.3::col(1, red); 0.7::col(1, blue).
0.2::col(2, red); 0.3::col(2, green); 0.5::col(2, blue).

win :- heads, col(_, red).
win :- col(1, C), col(2, C).
```

Or in natural language:
- Toss a coin, with a 40% chance of getting heads.
- Draw a ball from urn 1, with a 30% chance of getting red and a 70% chance of getting blue.
- Draw a ball from urn 2, with a 20% chance of getting red, a 30% chance of getting green, and a 50% chance of getting blue.
- You win if:
  - You get heads and at least one of the balls is red, or
  - Both balls are the same color.

We can ask multiple questions about this game:
- **Marginal Probabilities:** What is the probability of winning? $P(\text{win})$

- **Conditional Probabilities:** What is the probability of winning given that we got heads? $P(\text{win} \mid \text{heads})$
- **Most Probable Explanation (MPE):** What is the most probable world where we win? $\arg\max_w P(w \mid \text{win})$

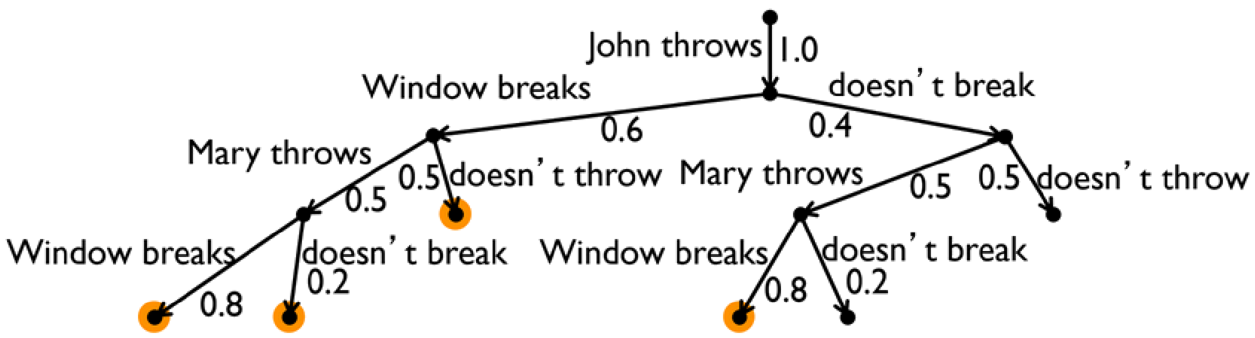


(a) - Causal Probabilistic Logic

Causal probabilistic logic is a framework that combines causal reasoning with probabilistic logic programming. It allows us to model and reason about causal relationships between variables in a probabilistic setting.

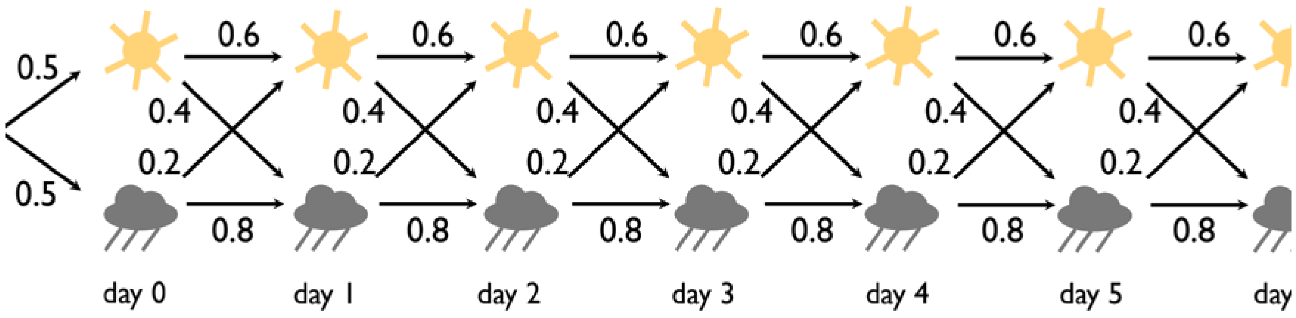
```
throws(john).
0.5::throws(mary)

0.8::break :- throws(mary)
0.6::break :- throws(john)
```



$$P(\text{break})=0.6 \times 0.5 \times 0.8+0.6 \times 0.5 \times 0.2+0.6 \times 0.5+0.4 \times 0.5 \times 0.8$$

```
0.5::weather(sun, 0) ; 0.5::weather(rain, 0).
0.6::weather(sun, T) ; 0.4::weather(rain, T)
    :- T>0, Tprev is T-1, weather(sun, Tprev).
0.2::weather(sun, T) ; 0.8::weather(rain, T)
    :- T>0, Tprev is T-1, weather(rain, Tprev).
```


0.5
0.5
0.6
0.4
0.2
0.8
day 0
day 1
day 2
day 3
day 4
day 5
day