CS 228: Probabilistic Graphical Models

Lecturer: Professor Stefano Ermon

Notes by: Andrew Lin

Winter 2024

1 January 9, 2024

Probabilistic modeling is a branch of machine learning which uses probability distributions to describe complex systems. At the highest level, this course will be about **mathematical modeling**, which is a fundamental tool in science and engineering – in a model, we use mathematical objects to represent a system, variables to describe quantities we care about, and equations to describe their relations. But the problem is that we can rarely make perfect predictions (because of noise, ambiguity, missing data, and so on), so we often want to **assess uncertainty** (e.g. make predictions with some associated confidence), which we can think about as **outputting beliefs**. So most machine learning models are inherently probabilistic models, since they're meant to represent real-world complex systems where we don't have full understanding of relationships between variables:

Example 1

We may want to calculate the probability that an image is assigned a particular label (classification), or that some given speech is speaking some particular string of text, or that a given English text translates to some Chinese text, or that a system transfers from our current state to some new state given some action that we take. Also, in generative AI models, we often want to describe the probability of outputting some particular text or image.

This class will tell us how to deal with uncertainty in a principled way, using computer science and mathematical tools to represent it; this will allow us to derive machine learning algorithms using probability and graph theory. We'll find that there are tradeoffs between computational complexity and the richness of the model, and we'll discuss a process for picking the right model (given some fixed dataset and computational budget), and we'll find principled ways to combine knowledge with observations for our predictions. We'll also describe some rigorous ways to analyze causality and make predictions about what would happen if we intervened in a system. Overall, the theme is that this class will be very broad and applicable to any problem in the general theme of probabilistic forecasting – we'll see applications to various problems in our homework.

The key idea of the course is that we will take the following three steps:

1. **Represent** the world with a collection of random variables $X_1, \cdots, X_n$ with some joint distribution $p(X_1, \cdots, X_n)$,
2. Use prior knowledge and given data to **learn** the distribution,
3. Perform **inference** by computing the conditional distributions of X_i given the values of some X_js.

There will be all kinds of challenges as we go through these steps – for example, we want to **compactly** describe the joint distribution, and we want a way to figure out how much data and computation is required for learning.

Nevertheless, this is the process that we go through when solving any new problem in machine learning. And unlike other AI or ML courses, this is a **top-down** course where we aren't starting with any particular problem and refining our techniques from there – instead, we're describing key building blocks that will be relevant for lots of problems.

Example 2

Localization is a classic application of these techniques – given noisy GPS measurements of a car, we may want to know the actual trajectory that it took.

The two sets of variables that matter are the true positions over time z_i and the measurements y_i. We'll encode their relationship using a joint distribution $p(z_0, z_1, \cdots, z_k, y_0, y_1, \cdots, y_k)$ and using a graph that encodes the transitions $z_{k-1} \to z_k$ and the observations $z_k \to y_k$; our overall goal will be to infer z given y. (A sparser graph means we will have an easier time computationally with the algorithm but also less prediction power.)

Example 3

Generative models for images use neural networks that also involve probabilistic models.

There may be some **latent variables** describing the content and some other variables encoding each of the pixels of the image, and our application is to sample from the distribution to get images that occur with high probability in the model. And if we allow some way of controlling the output (e.g. a text prompt), we can sample from the distribution of images given some caption and solve the text-to-image problem.

Example 4

Many other applications exist as well, such as speech recognition, language models (like ChatGPT), evolutionary biology (constructing a phylogentic tree given the DNA sequence of current species), and error-correcting codes (recovering an image after it goes through a noisy communication channel).

Over the course of this quarter, we'll start off in the simplest case with discrete variables and fully observed models, then generalize to partially observed data (e.g. hidden variables) and approximate inference and learning. By the end of the course, we should be able to understand the different kinds of graphical models out there (and how graphical properties are associated with statistical ones), implement common inference and learning algorithms and analyze their runtime, and tackle real-world problems.

All information about the course can be found on the course website, cs228.stanford.edu; discussion will be done on Ed and homework will be submitted on Gradescope. We'll follow the textbook *Probabilistic Graphical Models: Principles and Techniques* by Koller and Friedman, but there's a lot of material and it's not all required – we can instead consult the lecture notes on the website for a more compact summary. A list of TAs and office hours will also be available soon.

Prereqs for this class are some understanding of probability and statistics, algorithms, programming, multivariable calculus, and basic optimization techniques (like gradient descent and convexity). Grading will be based on 5 assignments (70 percent of the grade, a combination of theory and programming), final exam (30 percent), and participation (3 percent extra credit). The theory questions will require formal proofs, and the programming assignments will be done in Python and will take substantial time. The course will also have weekly ungraded quizzes which may be useful for preparation for the final.

For the rest of today, we'll do a bit of review of probability (random variables, independence, etc.):

Definition 5

In a random experiment, an **outcome space** (often denoted Ω) specifies the possible outcomes that we want to reason about.

For example, when we flip a coin, the outcome space is {heads, tails}, and when we roll a die, the outcome space is $\{1, 2, 3, 4, 5, 6\}$. We specify a probability for each outcome $\omega \in \Omega$, such that $p(\omega) \geq 0$ for all ω and $\sum_\omega p(\omega) = 1$. (For example, in a biased coin, we may have $p(\text{heads}) = 0.6$ and $p(\text{tails}) = 0.4$.

Definition 6

An **event** is a subset of the outcome space.

For example, we may have the set of even die tosses $E = \{2, 4, 6\}$ or odd die tosses $O = \{1, 3, 5\}$, and we can calculate the probability of an event by adding up the probabilities of all outcomes in that event:

$$p(E) = \sum_{\omega \in E} p(\omega).$$

(So if our die is fair, then $p(E) = \frac{1}{2}$.) We also may care about unions and intersections of events, which we can think of as the operations "or" and "and." Intersection is particularly useful because it's how we encode independence:

Definition 7

Two events A and B are **independent** if $p(A \cap B) = p(A)p(B)$.

For example, the events $A = \{1\}$ and $B = \{6\}$ are **not independent** because $p(A \cap B) = 0$ but $p(A)p(B) = \frac{1}{36}$ (intuitively, knowing that we get a 1 changes the probability that we get a 6). However, if we have two different dice, then the probability that the first one rolls a 1 **is independent** from the probability that the second one rolls a 6 (because now our outcome space has 36 equally likely possibilities, one of which results in both events happening).

Definition 8

Given two events A, B with $p(B) > 0$, the **conditional probability of A given B** is

$$p(A|B) = \frac{p(A \cap B)}{p(B)}.$$

Intuitively, we can imagine only taking the fraction of the area inside the part of the outcome space where B occurs (this can be visualized by drawing a Venn diagram). We can think of "conditioning on an event S" as creating a new probability distribution, since $\sum_{\omega \in \omega} p(\omega|S) = 1$. And if A and B are independent, we have $p(A|B) = p(A)$ from the definition of independence – think of this as "knowing B doesn't change our belief about whether A happened." This also means that we can rewrite

$$p(A \cap B) = p(B)p(A|B),$$

which we can think of more generally as leading to the "chain rule" for probabilities

$$p(S_1 \cap \cdots \cap S_n) = p(S_1)p(S_2|S_1)p(S_3|S_1 \cap S_2) \cdots p(S_n|S_1 \cap \cdots \cap S_{n-1});$$

this will be useful in terms of storing probability distributions later on in the course. And we can also use conditional probability to get **Bayes' rule**:

$$p(S_1|S_2) = \frac{p(S_1 \cap S_2)}{p(S_2)} = \frac{p(S_2|S_1)p(S_1)}{p(S_2)}.$$

This essentially provides us with a way to update our belief about S_1 given that some other event S_2 happens: weight our previous belief $p(S_1)$ by this new factor $p(S_2|S_1)$ and then normalize. This turns out to be surprisingly powerful despite the simple statement – unfortunately it's very expensive to calculate these terms exactly, but if we could find these exactly we'd have a principled way to update beliefs and incorporate data into a model.

Definition 9

A **random variable** is a mapping $X : \Omega \to D$, where D is some set (like the integers or real numbers).

Random variables are often easier to work with than events, and we can think of random variables as inducing a partition of outcomes (based on the value of $X(\omega)$). In particular, for any $x \in D$, we can calculate the probability that X takes on that particular value by adding up the probabilities of all ωs mapping to it:

$$p(X = x) = p(\{\omega \in \Omega : X(\Omega) = x\})$$

We'll let $\text{Val}(X)$ denote the set of values X takes on (we'll also call this the set of states).

Example 10

The **Bernoulli distribution** is a simple case where a random variable only takes on two values (say heads and tails). This random variable is then specified by the probability $p = p(X = \text{heads})$ (which means $1-p = p(X = \text{tails})$, and we write it as $X \sim \text{Ber}(p)$. A natural generalization is the **categorical distribution** where $D = \{1, \cdots, m\}$ can take on m different values, and we need to specify the values $p_i = p(Y = i)$. We'll write this as $Y \sim \text{Cat}(p_1, \cdots, p_m)$.

For example, a die roll is distributed as $\text{Cat}(1/6, 1/6, 1/6, 1/6, 1/6, 1/6)$.

Definition 11

Suppose we have a vector of random variables $X(\omega) = (X_1(\omega), \cdots, X_n(\omega))$. The **joint distribution** of these variables is

$$p(X_1 = x_1, \cdots, X_n = x_n) = p(\{X_1 = x_1\} \cap \cdots \cap \{X_n = x_n\}).$$

In shorthand, we'll often write this as $p(x_1, \cdots, x_n)$.

Example 12

Suppose we have a situation where students' intelligence I takes on values i_0 (high) or i_1 (very high), exam difficulty D takes on values d_0 (easy) or d_1 (hard), and students' grades G take on values A, B, or C. Then to specify the joint distribution, we need to specify the probabilities of each of the $2 \cdot 2 \cdot 3$ possible combinations (e.g. $I = i_1$, $D = d_0$, $G = A$).

But we can already see issues here – we need to specify $12 - 1 = 11$ parameters to specify the joint distribution (that's how many degrees of freedom we have), so for a larger system we will get exponential growth. And one of the first challenges we'll have to deal with is compactly representing this kind of distribution when our system size grows.

2 January 11, 2024

Our first homework assignment was posted on Ed yesterday – they'll always be due two weeks after they are first released.

Last time, we introduced the strategy that we'll be using in this course: represent the world using random variables with some joint distribution, learn the values of the distribution from data, and then perform inference by computing conditional distributions. But the first roadblock is often **finding efficient ways to store these distributions** with a computer. When we just have a single simple random variable like a Bernoulli biased coin or categorial distribution, we can completely specify the distribution using a single scalar p or a set of $m-1$ probabilities, respectively. But when we collect random variables together into a random vector and need to capture information about them at once, the number of parameters needed to specify all values $p(x_1, x_2, \cdots, x_n)$ grows exponentially in n, so we'll need a better strategy.

Example 13

Suppose we're building a generative model of images. To specify the color of a single pixel, we can specify the intensity of the red channel R, green channel G, and blue channel B, each of which are categorical and take on 256 different values (between 0 and 255 inclusive). But this means that if we wanted the joint distribution $p(R, G, B)$ yielding all possible pixel states, we would need $256^3 - 1$ parameters just for a single pixel. Similarly, even if we model an image where every pixel is just black or white (so each pixel's color is just a Bernoulli random variable) and are looking at the distribution of images we get for handwritten digits, we would need $2^n - 1$ parameters for an n pixel image.

This means that if we want to generate an small binary pixel image such that images similar to handwritten digits are more likely, we already don't have enough space to store a table of memory values (and it would be statistically very difficult to learn what the distribution should be). So this brute-force approach is not good enough, and that's what today's lecture will teach us how to get around. One strategy would be to assume that our random variables are actually all independent, meaning that we can write the joint distribution as

$$p(x_1, \cdots, x_n) = p(x_1)p(x_2)\cdots p(x_n).$$

We still have the same number of total states (2^n for the binary image), but we now only need to store n parameters to specify the joint distribution (since we just need to specify each marginal distribution $p(X_i = x_i)$, each of which is Bernoulli). This means that we get a huge computational advantage, but unfortunately pure independence is too strong for a useful model – knowing the value of some pixels does tell us information about others (for example, nearby pixels in images often have similar colors). So the strategy will be to instead assume **conditional independence**:

Definition 14

Two events A and B are **conditionally independent given an event C** if

$$p(A \cap B|C) = p(A|C)p(B|C).$$

Similarly, random variables X and Y are **conditionally independent given a random variable Z** if for all x, y, z,

$$p(X = x, Y = y|Z = z) = p(X = x|Z = z)p(Y = y|Z = z).$$

We'll often write this relation in simplified notation as $p(X, Y|Z) = p(X|Z)p(Y|Z)$; it turns out that it's equivalent to check that $p(X|Y, Z) = p(X|Z)$ (in words, this says that given Z, Y doesn't tell us anything about X). We will denote this independence by $X \perp Y|Z$, and we can also make the same definition for collections of random variables.

Example 15

Suppose we want to model the height and reading ability of a person. Then it's plausible that height and reading ability are not independent (since infants cannot read), but once we condition on age they are roughly independent.

A **Bayesian network** will basically specify the conditional independence structure among a set of variables, so that the model can be represented compactly while still giving us reasonable assumptions. The key building block here will be the **chain rule** from last lecture, which tells us that for random variables $X_1, \cdots, X_n$, we have (without any assumptions being placed on the variables)

$$p(X_1 = x_1, \cdots, X_n = x_n) = p(X_1 = x_1)p(X_2 = x_2|X_1 = x_1)\cdots p(X_n = x_n|X_1 = x_1, \cdots, X_{n-1} = x_{n-1}),$$

which can be more compactly written as $p(x_1, \cdots, x_n) = p(x_1)p(x_2|x_1)\cdots p(x_n|x_1, \cdots, x_{n-1})$. This factorization can be done with **any** ordering of our variables, and it's similar to the independence factorization but with some extra conditioning. Some of the conditioning is not too bad – $p(x_1)$ and $p(x_2|x_1)$ are rather manageable to represent – but something like $p(x_n|x_1, \cdots, x_{n-1})$ is not. (Indeed, if the X_i are all binary, we need 1 parameter for $p(x_1)$, 2 for $p(x_2|x_1)$, 2^2 for $p(x_3|x_1, x_2)$, and so on, and in total we end up requiring $2^n - 1$ parameters just like before.) So the key will be to simplify those more "expensive" distributions by assuming conditional independence.

Example 16

Suppose our random variables are indexed by time, so X_i represents some value of interest at time i. We can make the **Markov assumption**

$$X_{i+1} \perp X_1, \cdots, X_{i-1}|X_i,$$

which means that if we know the weather on the ith day, the weather on the $(i+1)$th day is independent from the weather before the ith day (we don't need any other information from the past).

So if we know that $X_3 \perp X_1$ given X_2, and $X_4 \perp X_1, X_2$ given X_3, and so on, then we can simplify our chain rule:

$$p(x_1, \cdots, x_n) = p(x_1)p(x_2|x_1)p(x_3|x_2)\cdots p(x_n|x_{n-1}),$$

and now we only need to condition on a single random variable at a time! In the case of binary random variables, this simplifies the number of parameters needed to represent the distribution from $2^n - 1$ to just $2n - 1$. And in general, the strategy will be similar: for each random variable X_i, we will specify the distribution of X_i given some (typically **small**) set of other random variables X_{A_i}, and then we'll get a joint parameterization of the form

$$p(x_1, \cdots, x_n) = \prod_i p(x_i|X_{A_i}).$$

As long as these sets are small, this representation will be tractable. But we do need to guarantee the distribution is **legal** – specifically, we need to make sure that there is some ordering such that all A_i variables come before i, so that this corresponds to a factorization of the chain rule.

Definition 17

A **Bayesian network** is specified by a directed acyclic graph (DAG) $G = (V, E)$, such that

- the vertices (nodes) $i \in V$ correspond to random variables X_i in the model,
- each node i corresponds to a conditional probability distribution $p(x_i | \vec{x}_{\text{Pa}(i)})$, where $\text{Pa}(i)$ is the set of parents of i.

We call G the **structure** of the Bayesian network, and we can use this to define a joint distribution

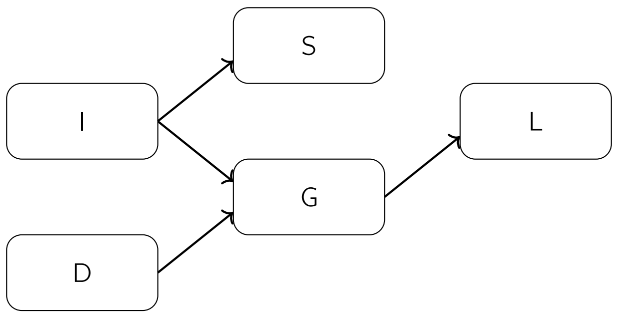
$$p(x_1, \cdots, x_n) = \prod_{i \in V} p(x_i | \vec{x}_{\text{Pa}(i)}).$$

The memory required to store this is now exponential in $|\text{Pa}(i)|$ instead of $|V|$, so this will be economical. And $p(x_1, \cdots, x_n)$ will be a valid probability distribution, since we can always order the graph so that the parents come before the children. Intuitively, the idea is that each random variable only depends on the value of its parents, and then this factorization is just the chain rule plus conditional independence assumptions.

Remark 18. *A* **directed acyclic graph** *(DAG) is a directed graph where there is no directed cycle (meaning that we can never follow the arrows and get back to where we started). A* **topological ordering** *of a graph is an ordering such that for any edge $u \to v$, u comes before v; such an ordering exists exactly when the graph is a DAG.*

Example 19

Suppose we have a Bayesian network involving five random variables for a student's performance in a class (difficulty D, intelligence I, grade G, SAT score S, and letter grade L). Then we could draw the following graph that describes the relationships between them:



Then to specify the joint distribution, we just need the distributions of difficulty and intelligence (with no conditioning), as well as the distribution of grade given difficulty and intelligence, of SAT score given intelligence, and letter grade given grade. In this case, we will have

$$p(d, i, g, s, \ell) = p(d)p(i)p(g|i, d)p(s|i)p(\ell|g).$$

We perform the chain rule in this order to represent that we first sample d and i, then figure out what g is, then figure out what s is, then figure out what ℓ is; (d, i, g, s, ℓ) is a topological ordering. And this distribution **implies** conditional independence: the **ordinary** chain rule would tell us that

$$p(d, i, g, s, \ell) = p(d)p(i|d)p(g|i, d)p(s|i, d, g)p(\ell|g, d, i, s),$$

and so we actually **additionally assume** that $D \perp I$, that $S \perp \{D, G\}|I$, and that $L \perp \{I, D, S\}|G$. There are others as well – for example, S and L are independent given I – and we'll see soon how to use the graph to algorithmically determine all such relations. But we'll demonstrate why this actually holds mathematically first:

Proof that $D \perp I$. As a quick reminder, the distributive law tells us that when we have a lot of variables and constants, we can expand out sums such as

$$\sum_i \sum_j a_i x_i b_j y_j = \sum_i a_i x_i \sum_j b_j y_j.$$

The marginal distribution for (D, I) can be written as

$$p(d, i) = \sum_{g \in \text{Val}(G)} \sum_{s \in \text{Val}(S)} \sum_{\ell \in \text{Val}(L)} p(d, i, g, s, \ell).$$

Plugging in the factorization for our joint distribution, this can then be written as

$$= \sum_{g,s,\ell} p(d)p(i)p(g|i, d)p(s|i)p(\ell|g) = p(d)p(i) \sum_{g,s,\ell} p(g|i, d)p(s|i)p(\ell, g),$$

and then the distributive law allows us to further factor as

$$= p(d)p(i) \sum_{g,s} p(g|i, d)p(s|i) \sum_{\ell} p(\ell|g).$$

The last sum is just 1, since conditioning still gives us a valid probability distribution. Then we can push the sum over s inside to simplify that term and then do the same for g, so we end up finding that $p(d, i) = p(d)p(i)$. □

Putting this in more general terms, we have the following conditional independence assumptions (called **local independencies**):

Proposition 20

For each variable X_v, we have the relation

$$X_v \perp \text{non-descendants of } X_v \mid \text{parents of } X_v.$$

Proof. Let $\vec{X}_N$ denote the vector of non-descendants, $\vec{X}_{\pi_v}$ denote the parents, and $\vec{X}_D$ denote the descendants of v. Then the graph structure means that we can factor as

$$\boxed{p(x_1, \cdots, x_n) = p(\vec{x}_N)p(\vec{x}_{\pi_v}|\vec{x}_N)p(x_v|\vec{x}_{\pi_v})p(\vec{x}_D|x_v, \vec{x}_{\pi_v}, \vec{x}_N)}.$$

Now we can write

$$p(x_v|\vec{x}_{\pi_v}, \vec{x}_N) = \frac{p(x_v, \vec{x}_{\pi_v}, \vec{x}_N)}{p(\vec{x}_{\pi_v}, \vec{x}_N)}$$

and plug in the boxed factorization into the numerator and denominator, meaning that we marginalize over x_D in the numerator and over both x_D and x_v in the denominator. Using the same strategy as before (with the distributive law), this ratio just simplifies to $p(x_v|x_{\pi_v})$ because all other terms cancel out. So we do have the conditional independence statement $p(x_v|x_{\pi_v}, x_N) = p(x_v|x_{\pi_v})$ as desired. □

Proposition 21

Conversely, if this relation $X_v \perp$ non-descendants of X_v | parents of X_v holds for all v, then we have a Bayesian network factorization.

Proof sketch. Use the general chain rule in the order given by topological variable ordering, and then simplify the formula using the conditional independence assumptions; we'll recover the joint distribution in the definition of a Bayesian network. □

So using a Bayesian network is actually **equivalent** to making conditional independence assumptions, and those are the ones you can read off directly from the graph.

Example 22

Suppose we want to build a spam classifier which classifies emails as either spam $Y = 1$ or not spam $Y = 0$. Suppose the words in our vocabulary (like English) are indexed from 1 to n, and we let $X_i = 1$ if the ith word appears in our email and 0 otherwise.

We can think of the emails as being drawn from a distribution $p(Y, X_1, \cdots, X_n)$; in general this is a complicated object which would require exponentially many numbers to store. But we can assume that **all words are conditionally independent given the label** Y, which corresponds to a Bayesian network in which there is an edge from Y to each X_i and no other edges. Under such an assumption, we can factor the joint distribution as

$$p(y, x_1, \cdots, x_n) = p(y) \prod_{i=1}^{n} p(x_i|y),$$

and on the flip side if we can write the distribution in this way we automatically **have** conditional independence. This is probably an incorrect assumption in practice (words do have quite a bit of relation to each other), but this model still turns out to work reasonably well. In other words, we can estimate the parameters using some training data, and we won't need too much data because of the simplicity of our model; we then predict using Bayes' rule

$$p(Y = 1|x_1, \cdots, x_n) = \frac{p(Y = 1)\prod_{i=1}^{n} p(x_i|Y = 1)}{\sum_{y\in\{0,1\}} p(Y = y)\prod_{i=1}^{n} p(x_i|Y = y)} = \frac{p(Y = 1)\prod_{i=1}^{n} p(x_i|Y = 1)}{p(Y = 0)\prod_{i=1}^{n} p(x_i|Y = 0) + p(Y = 1)\prod_{i=1}^{n} p(x_i|Y = 1)}$$

and classify an email as spam if this is higher than some threshold. So the idea is that even if the model is technically wrong, it can still be practically useful!

Fact 23

Bayesian networks are fully general – if we use a **complete** DAG structure where there is an edge from i to j whenever $i < j$, then the general chain rule factorization corresponds to this full Bayesian network. But here we don't make any conditional independence assumptions, so we don't get any computational benefits.

3 January 16, 2024

Last lecture, we saw how Bayesian networks (the simplest graphical model with a directed acyclic graph) specify a model leveraging conditional independence, allowing us to represent probability distributions more compactly. Today's lecture will teach us how to use the graph (as a data structure) to discover statistical properties of the model.

Recall from last time that for every variable X_v, we automatically know (from simplifying the expression for the joint distribution) that X_v is independent from its nondescendants given the values of its parents. It's important as a modeler to understand the assumptions and simplifications of our model, so we're now going to understand what **other** claims we can make about our variables.

Example 24

We'll start with simple cases for intuition where our networks only have three nodes A, B, C.

1.
$$\begin{tikzcd} & B \arrow[dl] \arrow[dr] & \\ A & & C \end{tikzcd}$$

2.
$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

3.
$$\begin{tikzcd} A \arrow[dr] & & B \arrow[dl] \\ & C & \end{tikzcd}$$

1. Suppose A and C both have a **common parent** B. Then knowing B **decouples** A and C, meaning that $A \perp C|B$ even though A and C are not marginally independent. (For example, A could be height, B could be age, and C could be reading ability.)
2. Suppose we have a **cascade** where A tells us information about B, which tells us information about C. Then knowing B again decouples A and C. (For example, A could be age, B could be intelligence, and C could be reading ability.)
3. Suppose we have a **V-structure** where A and B have a common descendant C. Then A and B are marginally independent, but knowing C can actually **couple** A and B together. (For example, A and B could be two independent causes of an event C.) This **explaining away** phenomenon is actually what makes Bayesian networks powerful!

Example 25

To elaborate more on this last case, suppose S is the indicator of going to school, I is the indicator of extreme intelligence, and R is reading ability (where all variables are binary). Assume that S and I are independent with $p = \frac{1}{2}$, and we have the V-structure $S \to R \leftarrow I$; also assume that $R = I \vee S$ deterministically.

Deterministic dependence means that our Bayesian network has all conditional probabilities 0 or 1; for example, the probability that $R = \text{True}$ given that $I = \text{False}$ and $S = \text{True}$ is 1. We can then calculate

$$p(I = \text{True}|R = \text{True}) = \frac{2}{3},$$

since $R = \text{True}$ implies that (I, S) either take the values $(\text{False}, \text{True})$, $(\text{True}, \text{False})$, or $(\text{True}, \text{True})$ with equal probabilities. However,

$$p(I = \text{True}|R = \text{True}, S = \text{True}) = \frac{1}{2},$$

since this means we must either have $(\text{False}, \text{True})$ or $(\text{True}, \text{True})$. So indeed conditioning on R makes I and S no longer independent – going to school already "explains the reading ability away," so our estimated probability of the other cause goes back down. Thus it is **true** that $I \perp S$ but **not true** that $I \perp S|R$.

Generalizing this analysis beyond just 3-node Bayesian networks, we want a way to decide if some set of variables is conditionally independent from some other set, given that some subset is already observed. Doing this check manually from a set of equations is complicated and error-prone, so we want to use a purely graph-based approach involving **graph separation**:

Definition 26

Let G be a Bayesian network and O be a set of observed variables. An undirected path in G is an **active path** if for every consecutive triple of nodes X, Y, Z along the path, we have one of the following:

- (Cascade) $X \leftarrow Y \leftarrow Z$ or $X \rightarrow Y \rightarrow Z$, and Y is **unobserved**.
- (Common parent) $X \leftarrow Y \rightarrow Z$, and Y is **unobserved**.
- (V-structure) $X \rightarrow Y \leftarrow Z$, but Y or one of its descendants is **observed**.

Equivalently, an active path is such that whenever we have a V-structure pointing toward V_i, then either V_i or one of its descendants is in O, and no other nodes along the path are in O.

Definition 27

If there is **no active path** between $\vec{X}$ and $\vec{Y}$ when $\vec{Z}$ is the set of observed variables, then we say that $\vec{X}$ and $\vec{Y}$ are **d-separated** with respect to $\vec{Z}$, and we write d-sep$(\vec{X}; \vec{Y} | \vec{Z})$.

The main result is the following:

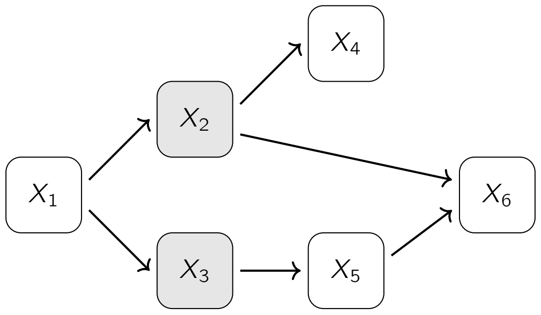
Theorem 28

If $\vec{X}$ and $\vec{Y}$ are d-separated with respect to $\vec{Z}$, then $\vec{X} \perp \vec{Y} | \vec{Z}$. In other words, $\vec{X}$ is conditionally independent from $\vec{Y}$ given $\vec{Z}$ if there is **no active path** between any element $X \in \vec{X}$ and any element $Y \in \vec{Y}$ for observations $\vec{Z}$.

Conditional independence then reduces statistical independencies to evaluating **connectivity in graphs**.

Example 29

Suppose we have the following graph shown below. Then to check whether d-sep$(X_1; X_6 | \{X_2, X_3\})$ holds, we need to check the top and bottom path from X_1 to X_6 – both of those are **not** active paths, so $X_1 \perp X_6 | X_2, X_3$ (knowing X_6 does not tell us anything more about X_1 if we already know X_2 and X_3).



Here are a few more examples from this network G:

- There is an active path from X_5 to X_6 (the single edge from X_5 to X_6 vacuously satisfies the condition), so X_5 and X_6 **are not** d-separated with respect to $\{X_2, X_3\}$. However, that doesn't necessarily tell us that $X_5 \perp X_6 | X_2, X_3$ does not hold (see the theorem below).
- X_4 and X_5 **are** d-separated given $\{X_2, X_3\}$, so we **can** safely claim conditional independence. They are also d-separated given X_1.

- However, X_4 and X_5 **are not** d-separated with respect to $\{X_1, X_6\}$ (the path (X_4, X_2, X_6, X_5) is active), so we cannot say anything about whether X_4 and X_5 are independent given X_1, X_6.

Example 30
We can in fact take the result from last lecture and state it as

$$\text{d-sep}(X_v; \text{non-descendants of } X_v | \text{parents of } X_v);$$

this is indeed true if we think about the different paths that could take us from X_v to one of its non-descendants and check that none of them are active paths.

Fact 31
Breadth-first search can check d-separation automatically and efficiently, since this is basically a check of whether there is a path from one set of vertices to another (except that we must verify that the paths are active paths).

Theorem 32 (Soundness of d-separation)
Suppose $I(p)$ are the set of true conditional independencies that hold for a joint distribution p (for example, if X, Y are independent Bernoulli random variables of parameter $p = \frac{1}{2}$, then $X \perp Y$ would be in $I(p)$). Let $I(G)$ be the set of conditional independencies $\{(\vec{X} \perp \vec{Y} | \vec{Z} : \text{d-sep}(\vec{X}; \vec{Y} | \vec{Z})\}$. Then if p factorizes according to G (in the sense of the Bayesian network), then $I(G) \subseteq I(p)$ (that is, all d-separation assumptions hold).

Unfortunately, **we do not have completeness** – it is not true that $I(G) = I(p)$, so there are additional assumptions that may not be revealed through d-separation in the graph structure. So sometimes we do need to use the actual values of the probability distributions.

Example 33
Suppose we have a simple network $A \to B$ where A and B are both binary. Then we actually have $A \perp B$ if we specify the values $P(B|A = a_0)$ to be identical to $P(B|A = a_1)$. Then in this particular case, $I(p) = \{A \perp B\}$, but $I(G)$ is the empty set (that is, *d*-separation cannot tell us anything).

But the idea is that we have to craft the probability distribution very carefully for this to hold:

Theorem 34
If $\vec{X}$ and $\vec{Y}$ are not d-separated given $\vec{Z}$, then there is **some** distribution p factorizing over G in which $\vec{X}$ and $\vec{Y}$ are dependent. In other words, if $\vec{X} \perp \vec{Y} | \vec{Z}$ in **all** distributions that factor over G, then $\vec{X}$ and $\vec{Y}$ are d-separated.

Proof sketch. Pick some active path between $\vec{X}$ and $\vec{Y}$ and choose the conditional probability tables so that we do actually get dependence (this will be true up to a set of measure zero). Then we can set everything else to be independent so that we don't "cancel the dependency" that we just created. □

Later on in the course, we'll be going from p to G instead of the other way around (trying to learn the structure from some data), and the fact that the graph cannot tell us the whole story has important consequences for what we can and cannot do from sampled data.

Example 35

Hidden Markov models are frequently used for models like speech recognition and Kalman filtering. The idea is that we have a set of hidden variables $\vec{Y}$ and a set of observations $\vec{X}$, and we can model this as a Bayesian network where we have edges $Y_i \to X_i$, as well as edges $Y_i \to Y_{i+1}$, for each i, and all $\vec{X}_i$ are observed.

In other words, we can factor the joint distribution as a product of conditional probabilities

$$p(\vec{y}, \vec{x}) = p(y_1)p(x_1|y_1)\prod_{t=2}^{T} p(y_t|y_{t-1}p(x_t|y_t).$$

Intuitively, $p(y_1)$ is the distribution of initial values Y_1 of a robot, and $p(y_t|y_{t-1})$ is the **transition probability** between states that it can take on. Then $p(x_t|y_t)$ is the **emission probability**, which we can think of as a noisy sensor measurement X_t of the true value Y_t, and $p(y_t|x_1, \cdots, x_t)$ can be thought of as the **filtering probability**.

Looking at the graph now, we can understand the conditional independencies between the variables without looking too carefully at the form of the distribution. For example, we cannot say that $X_1 \perp X_6$, since there is an active path $X_1 \to Y_1 \to \cdots \to Y_6 \to X_6$ – this means that we do not have independence for **most** choices of the conditional distributions. On the other hand, we do have $Y_1 \perp \{Y_3, Y_4, Y_5, Y_6\}|Y_2$, since the only path connecting those sets of vertices is not an active path.

So in summary, we can go back to the question of "to what extent can we construct G so that the independencies $I(G)$ are a reasonable surrogate of the true independencies $I(p)$." This is the question of **structure learning** when we have only partial data from p, and we'll have lots more about this in the coming lectures.

Definition 36

A directed acyclic graph G is an **I-map** (independence map) of a distribution p if $I(G) \subseteq I(p)$.

Since d-separation is sound, we know that G is a valid I-map if p factors over G. But a fully connected DAG G is an I-map for **any** distribution (since we're making no conditional independence assumptions), so our goal is not just to find an I-map – we want $I(G)$ to be as big of a subset as possible, making G as sparse as possible, and capture as many true independencies as we can. We'll discuss this more next time!

4 January 18, 2024

Last time, we discussed how to check for conditional independence in graphical representations using d-separation. At the end, we introduced the concept of an I-map to understand how much a graph tells us about the underlying distribution – an I-map of a distribution p is a DAG G where the set of conditional independencies implied by the graph, $I(G)$, is a subset of the conditional independencies that actually exist in a distribution, $I(p)$. We mentioned that a fully connected DAG always gives us $I(G) = \varnothing$, so that always counts as an I-map, but we'd like something more sparse to give us better efficiency.

Definition 37

A graph G is a **minimal I-map** for a distribution p if it is an I-map and removing any edge makes it no longer an I-map.

Any distribution p always has an I-map – we just start with a fully connected DAG (with no conditional independence assumed) and repeatedly remove edges as long as $I(G) \subseteq I(p)$. There are finitely many edges, so we can just keep doing this until we can't remove any more. But we might get a different result if we remove edges in a different order, so minimal I-maps are **not unique**. (This is a bit of a problem, since it means it will be tricky to estimate a graph G directly from data or establish direction of causation.)

Definition 38

A graph G is a **P-map** (perfect map) of a distribution p if $I(G) = I(p)$.

Unfortunately, not every distribution has a P-map (we'll see an example later on where no matter how we choose the graph, we can't perfectly capture all conditional independence relations). And even if we have a perfect map, it still doesn't need to be unique – for example, the two Bayes structures $X \to Y$ and $Y \to X$ both have the same set of independencies $I(G) = \varnothing$, but they imply very different causal statements. (Directionality of influence is one of the key challenges in causal inference, and we'll see this more later when trying to learn the graph from data.) So there are limitations with understanding relations with just the structure we've introduced here.

Related to this idea, we can think about when different Bayesian networks encode the same conditional independencies:

Definition 39

Two graphs G and G' are **I-equivalent** if $I(G) = I(G')$.

For example, the cascades $X \to Z \to Y$ and $Y \to Z \to X$, as well as the common parent structure $Z \to X, Z \to Y$ are all I-equivalent, but they're not equivalent to the V-structure $X \to Z, Y \to Z$.

Theorem 40

If two Bayesian networks are I-equivalent, then they must have the same skeleton (of undirected edges).

Having the same skeleton is necessary but not sufficient, as we saw in the V-structure example above. But adding in that additional condition is enough for equivalence:

Theorem 41

If G and G' have the same skeleton and the same V-structures, then $I(G) = I(G')$ (but the converse is not true).

Overall, we've seen that some types of dependencies cannot be easily captured by Bayesian networks, but it turns out that certain **undirected networks** will be more equipped to deal with those:

Example 42

Suppose we have four people A, B, C, D, and suppose we want to model their hair color (each red, green, or blue). Also, assume that the pairs A and B, B and C, C and D, and D and A are friends, so each pair never has the same hair color. This defines a distribution p which is uniform over all valid assignments of hair colors.

For example, the probability $p(A = \text{red}, B = \text{red}, C = \text{blue}, D = \text{green})$ is zero, while the probability of some valid assignment is $\frac{1}{\text{number of valid assignments}}$. (We can think of this as a constraint satisfaction problem.) This distribution satisfies $A \perp C | \{B, D\}$ (since A and C must take hair colors different from B and D, but they can choose from the possibilities independently) and similarly $B \perp D | \{A, C\}$. These are in fact the **only** independencies.

It turns out **there is no directed graph which is a P-map** for this distribution. (Indeed, we can go through all possible Bayesian networks with four nodes; no matter how we choose edges and their directions we won't get $I(G) = I(p)$. Natural choices of directed graphs end up encoding one of the independencies but not the other, or they encode additional ones and thus are not I-maps.) So instead we'll suggest using an **undirected graphical model**, which will be a data structure that doesn't obscure the structure of what's going on in this example.

Definition 43

In a **Markov random field**, we have an **undirected** graph in which each node corresponds to a random variable and edges between them represent dependencies. A **clique** of a graph is a complete subgraph (that is, a set of vertices all adjacent to each other); we now define a joint distribution

$$p(x_1, \cdots, x_n) = \frac{1}{Z}\prod_{c \in C} \phi_c(\vec{x}_c),$$

where C is **some subset** of cliques, ϕ_c is a function on the vertices of a clique $c \in C$, and Z is the normalizing constant, also called the **partition function**,

$$Z = \sum_{\hat{x}_1, \cdots, \hat{x}_n} \prod_{c \in C} \phi(c(\hat{\vec{x}}_c)).$$

The key difference is that we do not do a topological ordering, and calculating with cliques can be more computationally intensive than just looking at the parents of each edge. But for example, the pairs of edges in our example above are exactly the maximal cliques, so we are encoding the hat condition in a more natural way.

Just like with conditional probability distributions, we can represent $\phi_c(\vec{x}_c)$ with a table; the key difference is that these functions need not be normalized. (But notice that if we scale the values in a table by some constant, that doesn't change the underlying probability distribution.) These tables will not be too large if we don't have too many variables in a clique – we need 2^n parameters for an n-clique.

Example 44

Consider an undirected network on A, B, C, where the relevant cliques are the pairs of vertices. Suppose we specify that $\phi_{A,B}(a, b) = 10$ if $a = b$ and 1 otherwise, $\phi_{B,C}(b, c) = 10$ if $b = c$ and 1 otherwise, and $\phi_{A,C}(a, c) = 10$ if $a = c$ and 1 otherwise.

Then this Markov random field is assigning higher probabilities to configurations where adjacent variables are equal; in this situation we have

$$p(a, b, c) = \frac{1}{Z}\phi_{A,B}(a, b)\phi_{B,C}(b, c)\phi_{A,C}(a, c),$$

where Z sums $\phi_{A,B}(a, b)\phi_{B,C}(b, c)\phi_{A,C}(a, c)$ over all possible values of (a, b, c); in this case it ends up being 2060, so $p(0, 0, 0) = \frac{1000}{2060}$. These models are often called **energy models** (motivated by constructions in statistical physics).

Example 45

For the hair color example above, we can use a MRF with the joint distribution modeled as

$$p(a, b, c, d) = \frac{1}{Z}\phi_{AB}(a, b)\phi_{BC}(b, c)\phi_{CD}(c, d)\phi_{AD}(a, d)\phi_A(a)\phi_B(b)\phi_C(c)\phi_D(d),$$

where for example $\phi_{AB}(a, b) = 0$ if $a = b$ and 1 otherwise.

The single-node potentials like ϕ_A can be thought of as encoding preferences for a particular color (this is just giving us more control over the underlying distribution), but again we can scale a given ϕ however we'd like without changing the probabilities.

Fact 46
In this model, we cannot represent consistent local conditional or marginal probabilities because we don't have a topological ordering. And it turns out this will cause problems for sampling efficiently, and it can make interpretation more difficult as well.

We've described how d-separation encodes independence in Bayes networks; it turns out there is an analog of local independence in Markov random fields (also called **Markov networks**) which is actually simpler.

Theorem 47
Conditional independence in a Markov network is given by **graph separation**. In other words, if there is no path from any $a \in A$ to any $c \in C$ after removing the nodes for B, then $\vec{X}_A$ is independent from $\vec{X}_C$ given $\vec{X}_B$.

We can think of this being similar to common-parent and cascade-structures; there is no V-structure complication now that we don't have directionality. Intuitively, if variables in X_A and X_C can only interact via X_B, then cutting off those edges means there can be no conditional dependence.

Example 48
Markov networks can easily encode the relation $D \perp B|\{A, C\}$ in our hat example; removing A and C from the graph disconnects B from D. On the other hand, it's not always true that $A \perp C|D$, since there is still a path $A \to B \to C$.

Importantly (just like in Bayesian networks), we cannot claim that the existence of a path necessarily implies dependence, since it ends up depending on the values of the clique functions.

Definition 49
A set of variables $\vec{U}$ is a **Markov blanket** of a variable X if $X \notin \vec{U}$ and $\vec{U}$ is a minimal set of nodes such that

$$X \perp (\text{all variables} - \{X\} - \vec{U})|\vec{U}.$$

In an undirected model, the Markov blanket of X is its **set of neighbors** – knowing the values of X's neighbors makes the value of X independent from the remaining variables. On the other hand, in a Bayesian network, the Markov blanket of X is a little larger: it consists of X's parents and children, but **also** the other parents of X's children.

Example 50
Let's see a simple example of independence through separation. Suppose we have a distribution on three variables

$$p(a, b, c) = \frac{1}{Z}\phi_{AB}(a, b)\phi_{BC}(b, c)$$

and we want to show that $A \perp C|B$.

Notice that we can completely compute the conditional probability

$$\begin{aligned} p(a|b) = \frac{p(a,b)}{p(b)} &= \frac{\sum_{\hat{c}} p(a,b,\hat{c}}{\sum_{\hat{a},\hat{c}} p(\hat{a},b,\hat{c}} \\ &= \frac{\frac{1}{Z}\sum_{\hat{c}} \phi_{AB}(a,b)\phi_{BC}(b,\hat{c})}{\frac{1}{Z}\sum_{\hat{a},\hat{c}} \phi_{AB}(\hat{a},b)\phi_{BC}(b,\hat{c})} \\ &= \frac{\phi_{AB}(a,b)\sum_{\hat{c}}\phi_{BC}(b,\hat{c})}{\sum_{\hat{a}}\phi_{AB}(\hat{a},b)\sum_{\hat{c}}\phi_{BC}(b,\hat{c})} \\ &= \frac{\phi_{AB}(a,b)}{\sum_{\hat{a}}\phi_{AB}(\hat{a},b)} \end{aligned}$$

using only the value of ϕ_{AB} (notice that we basically "locally normalize" this factor to a probability distribution). The general phenomenon is that the value of some X, conditioned on its Markov blanket, will only depend on factors that involve X.

But doing a similar calculation for $p(a,c|b)$ will show that it's actually equal to $p(a|b)p(c|b)$, so we have the desired conditional independence. And doing this in general, we just group together all of the factors in the joint distribution as

$$p(\vec{X}_A, \vec{X}_B, \vec{X}_C) = \frac{1}{Z} f(X_{\vec{A}}, X_{\vec{B}}) g(\vec{X}_B, \vec{X}_C)$$

(there is no factor between A and C because that would imply that there is an edge, which we assumed is not true), and then the above proof works analogously.

Theorem 51 (Soundness of separation)
If a probability distribution p factors over an undirected graph G (we say that it is a **Gibbs distribution** for G), then G is an I-map for p.

The converse also holds with an additional assumption:

Theorem 52 (Hammersley-Clifford, 1971)
Suppose p is a **positive distribution** (meaning that $p(\vec{x}) > 0$ for all $\vec{x}$). If G is an I-map for p, then p factorizes over G.

Such a statement means that interactions are local, so we can write down the distribution in terms of these local factors (this will be useful later on for learning). And results about completeness are analogous to Theorem 34.

Example 53
When we do **image denoising**, we have a true image $\vec{Y}$ and a corrupted model $\vec{X}$, where we want to recover $\vec{Y}$ given $\vec{X}$. We can model this with a Markov random field

$$p(\vec{y}, \vec{x}) = \frac{1}{Z} \prod_i \phi_i(x_i, y_i) \prod_{(i,j)\in E} \phi_{ij}(y_i, y_j).$$

Such a factorization means that the ith corrupted pixel depends on the ith original pixel, and also adjacent pixels in the true image have interactions as well (for example, they are likely to have the same value). We can then infer the image $\vec{Y}$ by sampling from $p(\vec{y}|\vec{x})$; this will be better than separately trying to denoise each pixel.

So far we've only seen pairwise interactions – we can also have higher-order potentials perhaps of the form

$$\phi(x, y, z) = \begin{cases} 1 & x + y + z \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

In such a situation we say that the **scope** of ϕ is (X, Y, Z). The problem we start running into at that point is that we can't really tell from the graph which factors are being counted (for example, a clique of 4 vertices could represent 6 pairwise factors or a single factor for all terms).

Definition 54

A **factor graph** is a bipartite undirected graph between **variable nodes** and **factor nodes**, where edges are only between variable and factor nodes. (Think of this as taking the original set of vertices and connecting each factor to the nodes in its scope.) We still end up with the same distribution as in a Markov random field, but we represent it with a different structure as

$$\frac{1}{Z} \prod_{f \in \text{Factors}} \phi_f(\vec{x}_f).$$

(For an example of what this looks like, see Example 101 from a future lecture!)

5 January 19, 2024 (Section)

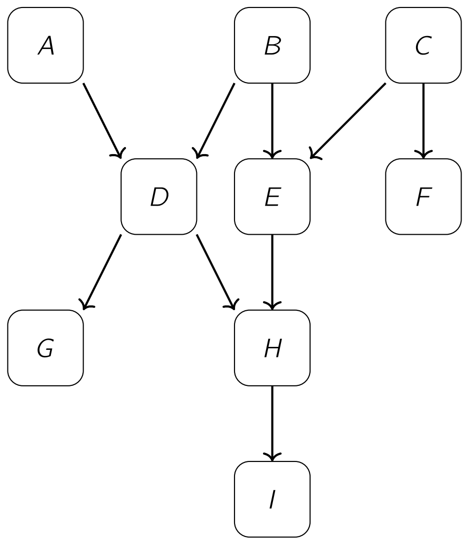
This is the first of five optional sections held by the TAs – today's topic is **d-separation** (and broadly the concept of independencies in graphical models), taught by Devansh Sharma. We'll first think about independence and I-equivalence in Bayesian networks, and then we'll conclude with a bit on independence in MRFs (which will be easier).

The main utility of Bayesian networks is to be **visual** – instead of needing to analyze the probability distribution, we can look at a graph and read off independence using facts like the local Markov property (that each variable is conditionally independent of its nondescendants, if we've observed its parents). Our key test is d-separation – recall that if there is no "flow of influence" between two variables (that is, there is no active path), then we have conditional independence. And we check whether a path is active by looking at consecutive triples – if we have a cascade or common parent, then we need the middle node to be **unobserved**, and if we have a V-structure, we need the middle node or one of its descendants to be **observed**.

Remark 55. *Intuitively (but not precisely), we can think of active paths as "dependency paths" where all nodes along the path are dependent on each other.*

Example 56

Consider the diagram below – we'll consider a few examples of d-separation on this graph.



- F and I are not d-separated, since we have a path (F, C, E, H, I) which is active.
- If B is observed, then F and D are d-separated (there are no active paths, since any path between them has to go through (E, B, D) or (E, H, D).
- If I and B are observed, then G and E are not d-separated, since we have the active path (G, D, H, E).

This was a pretty small graph to visually work with, but there's a general algorithm to check whether two sets X and Y are d-separated given Z. We can't enumerate all paths from X to Y and check if they're active (since there may be exponentially many of them) – instead, the following strategy gives a **linear-time procedure**:

- Traverse the graph bottom-up from Z, and mark all nodes that are in Z or are ancestors of Z; call this set W.
- Now traverse the graph breadth-first from X to Y, stopping when we encounter a **blocked node**, meaning that we are in the middle of a V-structure but not in W or are not in the middle of a V-structure but are in Z.
- If this breadth-first search manages to reach Y without encountering a blocked node, then there is an active path and thus we do not have d-separation. Otherwise, the two sets are d-separated.

(There are some additional details – we need to mark tuples rather than just individual nodes, for example – but we can read more about this in the book if we'd like.)

Recall also that we may want to know if two networks G_1, G_2 contain the same set of independencies (that is, $I(G_1) = I(G_2)$). We've seen that if Z is in the middle of a network of 3 nodes X, Y, Z, then common-parent and cascade structures all give the same independencies, but a V-structure encodes the fact that X and Y are not independent given Z. To prove this, recall that we can write a Bayesian network $X \to Z \to Y$ as

$$p(X, Y, Z) = p(X)p(Z|X)p(Y|Z),$$

the other cascade $Y \to Z \to X$ as

$$p(X, Y, Z) = p(Y)p(Z|Y)p(X|Z),$$

and the common-parent structure $Z \to X, Z \to Y$ as

$$p(X, Y, Z) = p(Z)p(X|Z)p(Y|Z).$$

But we can actually get from any of these expressions to any other one using Bayes' rule, since

$$p(X)p(Z|X)p(Y|Z) = p(Z)p(X|Z)p(Y|Z) = P(Z)P(Y|Z)p(X|Z) = p(Y)p(Z|Y)p(X|Z)$$

(where the first and last steps are Bayes' rule and the middle is just rearranging the factors). On the other hand, we can check that none of these manipulations will get us to the V-structure $X \to Z$, $Y \to Z$.

One fact is that **if two graphs G, G' are I-equivalent, then they must have the same skeleton** – informally, because if we have an edge present in G' but not in G', then we have an extra independence in $I(G)$ but not in $I(G')$. On the other hand, having the same skeleton alone isn't enough (like the V-structure versus a cascade from the previous example); instead, it's true that **if two graphs have the same skeleton and V-structures, then they are I-equivalent**. (Unfortunately, this last statement is not an if and only if statement either; there are graphs G, G' with the same independencies but different V-structures. For example, we can take two fully-connected DAGs.)

Fact 57

We care about I-equivalence for the purposes of structure learning – often we start with a fully-connected Bayesian network and repeatedly remove edges when we test for independence. Unfortunately, we can't really distinguish between $A \to B$ and $B \to A$ with this strategy, and thus some additional assumptions will be introduced later in the course to help with this.

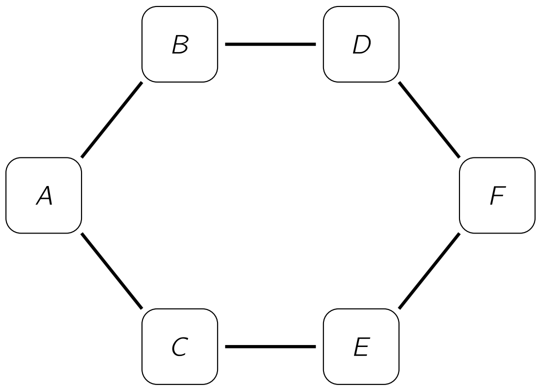
On the other hand, Markov random fields are much easier to reason about because we have no directionality and thus no nuances with V-structures – independence just comes from ordinary graph separation. This gave rise to the concept of a **Markov blanket** U of a variable X, which is the minimal set of nodes that need to be observed to make X independent of everything else. In the case of MRFs, this Markov blanket will just be the set of neighbors to X.

Remark 58. *A fire alarm went off, so the rest of the section was done online – the remaining notes are transcribed from the lecture notes.*

As we saw in lectures, MRFs are able to encode certain independencies that Bayesian networks cannot (for example in the "different hair color" examples); however, they cannot encode **unconditional** independence without disconnecting the graph.

Example 59

Finally, we'll show an example where we go from an MRF to a Bayesian network. Suppose we have the Markov random field shown below, and we want to construct a minimal I-map for it.



The idea is that even though **ordering does not matter for MRFs, it does for Bayesian networks**. So what we can do is pick an ordering, say (A, B, C, D, E, F), of our vertices. Then for each vertex, we choose its parents in a minimal way so that we get the correct set of independencies. In this case, that means we start with A, then have A as a parent of B, then have both A and B as parents of C (since even conditioned on A, C and B are dependent). But then we only need B and C to be parents of D (since A and D are independent given B and C), and similarly only C and D as parents of E, and finally D and E as parents of F.

Fact 60
If we're interested in a general method for building a minimal I-map, the textbook describes an algorithmic procedure for that as well. It's basically the same idea as what we see here – if we've already gone through X_1 through X_n, then X_{n+1}'s parents are some minimal subset S of $\{X_1, \cdots, X_n\}$ such that given S, X_{n+1} is independent of $\{X_1, \cdots, X_n\} \setminus S$.

6 January 23, 2024

Our first topic for today is **converting Bayesian networks to Markov random fields** – this conversion is particularly useful for inference, since a lot of algorithms there are easier to represent in terms of undirected models. It turns out we just need to make a few kinds of changes to do so:

Definition 61
Let $G = (V, E)$ be a Bayesian network. The **moral graph** $\mathcal{M}[G]$ of a Bayesian network is an undirected graph over V, such that we draw an edge between X_i and X_j if **either** there is a directed edge between X_i and X_j, **or** they are both parents of the same node.

This is thought of as "marrying parents" of a node. For example, if we have a V-structure $A \to C, B \to C$, then the moralization has an edge between A and B as well. Note however that we **lose some independence information** by doing this – for example, in such a V-structure we have $A \perp B$, but the moralization does not capture this information. Once we have this moralization, we can introduce a potential function for each conditional probability distribution via

$$\phi_i(x_i, \vec{x}_{\text{Pa}(i)}) = p(x_i | \vec{x}_{\text{Pa}(i)});$$

this explains why we need to moralize the graph, since we need to make x_i and its parents into a single clique to have a valid MRF.

Example 62
In a hidden Markov model (like in Example 35), we don't have any common parents, so we just convert it to a Markov random field by removing the directionality of edges (and the factors are just from individual edges, coming from the conditional probability distributions). But when we have multiple parents, it's often useful to be more explicit and switch to the factor graph notation.

All of this shows us that undirected and directed models are good in different situations – V-structures do not have an undirected representation, but the hair color example does not have a directed representation.

Fact 63
In a Markov random field, conditioning on more random variables always creates more indepencies. But this isn't always true in Bayesian networks because of the explaining away phenomenon.

Now that we've introduced the different kinds of models that we can use, we can figure out **which ones are suitable to which situations**.

Example 64

Suppose we are thinking about a classic machine learning question where we want to go from a set of features $\vec{X}$ and get some labels Y.

There are many different ways to do this – we can have a **generative** model represented by $Y \to \vec{X}$ (where we get features from labels), a **discriminative** model represented by $\vec{X} \to Y$ (where the labels are obtained in terms of features, for example if we're trying to predict them), or an **undirected** model with an undirected edge between $\vec{X}$ and Y. There are tradeoffs between all three representations, and we'll see how they come up with our next example.

Example 65

Consider the email classification example from Example 22, but now not necessarily with all X_is independent. In such a situation, we have a **generative** model where the label Y is a parent of all features X_i.

This Bayesian network does not actually give us the value of $p(Y|\vec{X})$ directly – we can use Bayes' rule to compute it, but it's not explicitly calculated in the conditional probability distributions and we need to do inference to find it. Thus, we could try using a **discriminative** model for this email classifier instead. That would directly give us the value $p(Y|\vec{X})$ needed for classification – we wouldn't need to specify or estimate both the values of $p(Y)$ **and** $p(\vec{X}|Y)$, and we don't even need the value of $p(\vec{X})$ either. However, this model is only useful specifically for discriminating Y's label – we can't use it to generate a new email.

Fact 66

Being a bit more explicit here, the chain rules for the two different models look like

$$p(Y|\vec{X}) = p(Y)p(X_1|Y)p(X_2|Y, X_1)\cdots p(X_n|Y, X_1, \cdots, X_{n-1})$$

for the generative model and

$$p(Y|\vec{X}) = p(X_1)p(X_2|X_1)p(X_3|X_1, X_2)\cdots p(Y|X_1, \cdots, X_{n-1}, X_n)$$

for the discriminative model. And the way we make either of these models computationally tractable is by **simplifying with different assumptions**.

In the generative case, we need to parameterize $p(X_i|\vec{X}_{\text{Pa}(i)}, Y)$ (because X_n has lots of parents), while in the discriminative case, we don't actually care about $p(\vec{X})$ (because $\vec{X}$ is given to us already in a classification problem) but we still need to parameterize $p(Y|\vec{X})$ efficiently.

As we've discussed in a previous lecture, the strategy for the generative case is to use **naive Bayes**, where we say that $X_i \perp \vec{X}_{-i}|Y$ (where $\vec{X}_{-i}$ means all Xs except X_i). But in the discriminative model, we make progress in a different way using **logistic regression**. We make the assumption that the entries have a special structure given by a function

$$p(Y = 1|\vec{x}; \alpha) = f(\vec{x}, \vec{\alpha}),$$

where $\vec{\alpha} = (\alpha_0, \cdots, \alpha_n)$ is some additional set of parameters. So now instead of having a conditional probability table (which would have too many rows), we have a deterministic function parameterized by α. We want this function to depend in a simple way on our variables $\vec{x}$ – the simplest thing we can do is to use a linear dependence

$$z(\vec{\alpha}, \vec{x}) = \alpha_0 + \sum_{i=1}^{n} \alpha_i x_i,$$

but this doesn't get us probabilities between 0 and 1. So the last step is to apply a nonlinearity using the S-shaped **sigmoid function** and define

$$p(Y = 1|\vec{x}; \vec{\alpha}) = \sigma(z(\vec{\alpha}, \vec{x})), \quad \sigma(z) = \frac{1}{1 + e^{-z}}.$$

This is one way to think about a logistic regression classifier – we have some variables $x_1, \cdots, x_n$, and we assume that the probability of $Y = 1$ given these values has a nice deterministic form. Unfortunately, such an assumption has lots of limitations in terms of the shape of the curve: the decision boundary $p(Y = 1|\vec{x}; \vec{\alpha}) = 0.5$ is linear in x, and in general the equiprobable contours are straight lines. But it's just another way of simplifying the joint distribution.

Remark 67. *We'll talk more about regularization when we discuss Bayesian learning and interpretation later, but we can think of it as prior knowledge about how the variables are related – it has to do with the distribution of αs as random variables as well.*

Fact 68

It turns out that under some conditions, the naive Bayes assumption actually **implies** the functional form given by a logistic regression classifier. Indeed, if we assume that Y is Bernoulli with some parameter π and $X_i|Y$ is Gaussian with parameters μ_{iy}, σ_i (in particular, the variance is class-independent for each variable), we'll find the logistic form we described above.

So logistic regression performs **at least** as well as naive Bayes, since any naive Bayes representation (under these conditions) is some particular instance of a logistic model.

Example 69

In practice, it can be important that the logistic model doesn't require conditional independence among the X_i. For example, the occurrence of "bank" and "account" may be very correlated, and thus naive Bayes may be **double counting evidence** even if the correlation does not depend on whether or not the email is spam. A logistic regression classifier would instead just set one of the αs to zero while learning.

However, generative models do still have their use – if some of our features $\vec{X}$ are missing, then just having access of $p(Y|\vec{X})$ does not allow us to make a prediction in the logistic model. In such a situation, we can marginalize over the unseen variables in a generative model. And in general, estimating the generative model using maximum likelihood is more efficient than training the discriminative model. We'll return to this in the later half of the course!

Example 70

In a standard machine learning problem, we want to map a sequence of complex inputs $\vec{X}$ to some relatively simple output $y \in \mathbb{R}$. But we often have structured output problems where the outputs $\vec{Y}$ are also complex and high-dimensional. For example, suppose we input an image, and we want to output a segmentation mask (for example distinguishing which parts of an image correspond to a horse versus the background).

If we are dealing with $M \times N$ color images, this problem has input space $\mathcal{X} = [0, 255]^{3MN}$ and output space $\mathcal{Y} = \{0, 1\}^{MN}$. In such a situation, we don't need to model how the images look like – we just need to discriminate by solving for a **prediction function**

$$f(\vec{x}) = \text{argmax}_{y \in \mathcal{Y}} p(\vec{y}|\vec{x}),$$

to figure out what the most likely segmentation mask is. And there is a complicated dependency between the pixels, but that gets encoded in the conditional distribution.

Example 71

For a more simple example, suppose we are trying to do character recognition: we have a sequence of 5-letter word images, and we want to map them to some ASCII translation. In other words, we have $\vec{x} = (x_1, \cdots, x_5)$ as input, where $x_i \in \{0, 1\}^{300 \cdot 80}$, and $\vec{Y} = (y_1, \cdots, y_5)$ as output, where $y_i \in \{A, B, \cdots, Z\}$.

Again, we don't need to model what handwritten characters look like – we just need a model which tells us which character sequence is most likely. But then we need the relationship between each x_i and y_i, **as well as** relationships between the characters y_i (since some strings are much more likely to occur in English than other ones – if we see a handwriting sample that looks either like "QUEST" or "QVEST," we can guess that it's very likely to be the former).

Definition 72

A **conditional random field** (CRF) is an undirected graphical model specifically designed to model conditional distributions. In a CRF, we have **observed variables** $\vec{X}$ and **target variables** $\vec{Y}$, and we have a graph with both X_i and Y_j variables. Just like in an MRF, we have potentials associated with cliques in the graph (which involve both kinds of variables and tell us how compatible different assignments are), which specifies the **conditional** distribution

$$p(\vec{y}|\vec{x}) = \frac{1}{Z(\vec{x})} \prod_{c \in C} \phi_c(\vec{x}_c, \vec{y}_c),$$

where C is some set of cliques and $Z(\vec{x}) = \sum_{\hat{y}} \prod_{c \in C} \phi_c(\vec{x}_c, \hat{\vec{y}}_c)$ is the normalizing constant to make this a probability distribution.

(Notice that this time the partition function Z depends on $\vec{x}$, and we only sum over the Y variables.) Just like for MRFs, it's convenient that independence is exactly reflected by graph separation.

Example 73

In the handwriting example above, we can associate a potential $\phi_i(x_i, y_i)$ to each clique (x_i, y_i) which is large if the image x_i looks like the character y_i, so that $P(\vec{y}|\vec{x}) = \phi_1(x_1, y_1) \cdots \phi_5(x_5, y_5)/Z(\vec{x})$.

If those were the only potentials, the prediction $f(\vec{x}) = \text{argmax}_{y \in \mathcal{Y}} p(\vec{y}|\vec{x})$ would just be obtained by separately optimizing each term $\phi_i(x_i, y_i)$, since we are making independent assumptions at each location. So it's better to add some other cliques as well, such as adding some terms of the form $\phi_{i,i+1}(y_i, y_{i+1})$ corresponding to frequencies of adjacent character pairs.

We might want to try representing these factors in a table, but for something like $\phi_1(x_1, y_1)$ the variables are too high-dimensional for that to be practical (in this case we'd need $2^{300 \cdot 80} \cdot 26$ terms). So instead we will often **assume a specific functional form**, such as the logistic regression example described above. A popular choice is the log-linear function

$$\phi_c(\vec{x}_c, \vec{y}_c) = \exp\left(\vec{w}_c \cdot \vec{f}_c(\vec{x}_c, \vec{y}_c)\right),$$

where $\vec{w}_c \in \mathbb{R}^d$ is a (learnable) weight vector and $\vec{f}_c(\vec{x}_c, \vec{y}_c) \in \mathbb{R}^d$ is a **fixed** feature vector; here we can choose d to relatively small, and as long as the features aren't too complicated this will work quite well. (In this case, we can let

$\phi_i(x_i, y_i)$ be a convolutional neural network, and we can let $\phi_{i,i+1}(y_i, y_{i+1})$ be represented as a table. And if we have global information, such as an English dictionary, we could make more connections between all of the y_is encoding the validity of words – it's just that learning and inference may become more expensive.) But what's nice is that we can use very complicated features over $\vec{x}$ because we don't need to model a probability distribution for it.

Example 74

Our logistic regression model can be thought of as a conditional random field of the form

$$p(y|x_1, x_2, \cdots, x_n) = \frac{1}{Z(x_1, \cdots, x_n)} r(y) f(x_1, x_2, \cdots, x_n, y).$$

In other words, there are two cliques – the first one is just over the single node y, and the second is over all variables.

We recover the previous functional form by choosing our functions to be

$$r(y) = \exp(\alpha_0 \cdot 1\{y = 1\})$$

(meaning that $r(0) = 1$ and $r(1) = \exp(\alpha_0)$) and

$$f(x_1, x_2, \cdots, x_n, y) = \exp\left(\left(\sum_{i=1}^{n} \alpha_i x_i\right) 1\{y = 1\}\right)$$

(meaning that $f(\vec{x}, 0) = 1$ and $f(\vec{x}, 1) = \exp(\sum_i \alpha_i x_i)$). Computing the conditional distribution does indeed yield

$$p(Y = 0|\vec{x}) = \frac{1}{Z(\vec{x})}, \quad p(Y = 1|\vec{x}) = \frac{1}{Z(\vec{x})} \exp\left(\alpha_0 + \sum_{i=1}^{n} \alpha_i x_i\right),$$

so we do get the correct logistic regression formula. And Z turns out to be the denominator of a softmax if we have a more general categorical (multi-class) classification.

7 January 25, 2024

We've finished the first module of the course now, understanding different models for representing probability distributions. From here, the exciting part begins – we'll figure out how to solve problems, efficiently computing probabilities by exploiting the independence properties we've set up in our representation. So all of this will be a question of **how to infer**.

At a high level, the models we've discussed in this class have two different components – the graph structure and the factors or probability distributions we put on that structure. There's several ways we can acquire such a model: perhaps we have some expert prior knowledge and already know the graph or factors, but often we use data or learning to determine what the graph looks like (we call this **structure learning**) or what the factors look like (we call this **parameter learning**); sometimes we need to do both and that can be particularly difficult.

In practice, though, we often have some **dataset** that we can work with (for example emails that have already been classified as spam or not spam), which we can formalize as a set of samples $\mathcal{D} = \{\vec{x}^{(1)}, \cdots, \vec{x}^{(m)}\}$ taken from some underlying distribution p^* (think of this as an assignment of variables in our model). We usually assume that these data points are **independent and identically distributed** (**iid** for short), meaning that the probability of observing a dataset is given by

$$p^*(\mathcal{D}) = \prod_i p^*(\vec{x}^{(i)}).$$

Furthermore, we are often given a **family of models** $\mathcal{M}$, and we want to learn what the "good model" $\hat{\mathcal{M}} \in \mathcal{M}$ within that family is. (For example, $\mathcal{M}$ may be the set of all Bayesian networks with some **given** graph structure, ranging over the possible choices of conditional probability tables. That would be an example of structure learning, where we just need to learn the model parameters.) Specifically, we want $p_{\hat{\mathcal{M}}}$ to be as close as possible to the true underlying data distribution that we sample from. This is a hard problem in general – we have limited data which is itself an approximation of the true distribution, and it can be too computationally difficult to go through the data.

Example 75

Let's go back to the email classifier from earlier in the course – suppose every email is represented with a vector of 1000 binary features $\vec{X}$, plus a single binary label Y. Then there are more than 10^{300} possible states in our model, so even if we have 10^7 emails to work with, we still have very sparse coverage of the joint probability distribution. So given that we can't get p^* exactly, we resort to making a best approximation.

We do need to quantify what "best" means here, though, and the metric that we use (leading to different loss functions) depends on the problem we're trying to solve:

1. The most general task is **density estimation**, in which we want the full distribution so that we can compute any conditional probability query that comes up.
2. On the other hand, sometimes we care about specific **prediction** tasks (like seeing if an email is spam) where the distribution is used for a particular calculation.
3. Finally, we sometimes are curious mostly about the **structure** of the model itself – perhaps we want to do causal inference or see how different variables (like genes) interact with each other.

In case (1), our main goal is to estimate $p^*(\vec{x})$ for every $\vec{x}$. This is like a regression problem, except we don't actually know what p^* is (just a bunch of random samples from it). This means that our model distribution $p_{\hat{\mathcal{M}}}$ should be close to p^*, and one way to measure the closeness of two distributions is the following:

Definition 76

The **Kullback-Leibler divergence** (KL divergence for short) between two distributions p, q is defined as

$$D(p||q) = \sum_{\vec{x}} p(\vec{x}) \log \frac{p(\vec{x})}{q(\vec{x})}.$$

In other words, the KL divergence is the average value of $\log(p/q)$ with respect to p; if p and q agree everywhere then this quantity will be zero, while if p and q are significantly different this quantity will be much larger. So if we made this our loss function, it would be a reasonable learning goal for incentivizing q to approach p.

Proposition 77

$D(p||q)$ is always nonnegative, with equality only if $p = q$.

Proof. As mentioned above, we can write

$$D(p||q) = \mathbb{E}_{\vec{x}\sim p}\left[-\log \frac{q(\vec{x})}{p(\vec{x})}\right] \geq -\log\left(\mathbb{E}_{\vec{x}\sim p}\left[\frac{q}{(\vec{x})}p(\vec{x})\right]\right),$$

where we use Jensen's inequality on the convex function $-\log$. Then expanding the expectation inside the log, we get $\sum_{\vec{x}} p(\vec{x})\frac{q(\vec{x})}{p(\vec{x})} = \sum_{\vec{x}} q(\vec{x}) = 1$, so the right-hand side is zero and thus we get our result. $\square$

We should be careful not to think of the KL divergence as a distance – in general we have $D(p||q) \neq D(q||p)$. But we can think of it as a **relative entropy** – $D(p||q)$ essentially tells us how many extra bits of information are required to describe samples **from** p, if we're using a code based on q instead of p. For example, Morse code is a system which encodes letters with dots and dashes, where letters like E or A are given much shorter encodings than Q (so that messages are as short as possible on average). If we now wanted to design the equivalent of Morse code for Japanese, it would be useful to know which characters are more frequent than others – KL-divergence measures how efficient our scheme is if we optimize for some distribution q other than the true one p.

Thus, we can evaluate closeness with the formula

$$D(p^*||\hat{p}) = \mathbb{E}_{\vec{x}\sim p^*}\left[\log\left(\frac{p^*(\vec{x})}{\hat{p}(x)}\right)\right] = \sum_{\vec{x}} p^*(\vec{x}) \log\frac{p^*(\vec{x})}{\hat{p}(x)}$$

and try using this as an objective to minimize. We can simplify this further as

$$D(p^*||\hat{p}) = \mathbb{E}_{\vec{x}\sim p^*}[\log p^*(\vec{x})] - \mathbb{E}_{\vec{x}\sim p^*}[\log \hat{p}(\vec{x})]$$

and what's nice is that the first term (this is actually $-H(p^*)$, the negative entropy of the distribution p^*) has **nothing to do with the model**, while the second term can be thought of as **maximizing the log-likelihood**. That is, we want to choose a model which maximizes $\log\hat{p}$ (assigning high probability to our data for instances sampled from p^*), which justifies the choice of learning objective.

Unfortunately, if we just think about this in terms of maximum likelihood, ignoring the $H(*p)$ term means we don't know how **close** we are to the optimal solution – we can only compare two distributions p_1 and p_2 and see which one is better. And because we don't know p^* in general, we approximate the **expected** log-likehood $\mathbb{E}_{\vec{x}\sim p^*}[\log\hat{p}(\vec{x})]$ with the **empirical log-likelihood**

$$\frac{1}{|\mathcal{D}|}\sum_{\vec{x}\in\mathcal{D}} \log\hat{p}(\vec{x});$$

this is the loss function that we'll use, since it only involves data and the model and therefore we can build learning algorithms for it. So this means that when we do **maximum likelihood learning**, we are computing

$$\max_{\hat{p}\in\mathcal{M}} \frac{1}{|\mathcal{D}|}\sum_{\vec{x}\in\mathcal{D}} \log\hat{p}(\vec{x}),$$

which (because the data points are independent) is equivalent to maximizing the likelihood of the data $\hat{p}(\vec{x}^{(1)}, \cdots, \vec{x}^{(m)})$.

Example 78

Suppose we have a biased coin that we want to model – the coin can take on values H or T, and suppose we have the data $\mathcal{D} = \{H, H, T, H, T\}$. Our class of models is just the set of probability distributions over $\{H, T\}$, and thus our learning task is "how to choose $\hat{p}$ given that 3 out of 5 of our tosses came up heads."

For this example, we represent the model with a parameter $\theta = \hat{p}(x = H)$ (meaning that $1 - \theta = \hat{p}(x = T)$). Then given any choice of our model, the probability of generating this particular dataset is $\theta^3(1-\theta)^2$ – we then want to choose θ that maximizes this probability. In this case (perhaps not surprisingly) the answer turns out to be $\theta = \frac{3}{5}$. More generally, if we have a general dataset of h heads and t tails, then the likelihood of our data is

$$L(\theta) = \theta^h(1-\theta)^t \implies \log L(\theta) = h\log\theta + t\log(1-\theta),$$

and taking a derivative of the log-likelihood and setting that equal to 0 shows that $\theta^* = \frac{h}{h+t}$ is the optimal solution. We can do a similar calculation for a die roll or other categorical distribution as well:

Fact 79

For any categorical distribution, the set of models is of the form $\{(\theta_1, \cdots, \theta_n) : \sum_k \theta_k = 1\}$; it turns out that using Lagrange multipliers and doing a similar calculation as before yields

$$\theta_k^* = \frac{\text{number of samples of outcome } k}{\text{total samples}}.$$

However, notice that if we never see some particular variable occur, this process would choose $\theta_k^* = 0$. Thus we often do a process called **Laplace smoothing** (which we'll learn more about when we talk about Bayesian learning) and instead choose

$$\theta_k^* = \frac{\text{number of samples of outcome } k + \alpha}{\text{total samples} + 6\alpha}.$$

for some real number α.

Example 80

So now we understand how to do this process with a single random variable (just one node), and now we must think about how to extend this MLE principle to a Bayesian network.

Our model family is now given by a fixed Bayesian network structure with the factorization

$$\hat{p}(\vec{x}) = \prod_{i=1}^{n} \theta_{x_i|\text{Pa}(x_i)},$$

and again we have some training data $\{\vec{x}^{(1)}, \cdots, \vec{x}^{(m)}\}$. Our goal will be to select probability tables given some dataset of samples, where each sample is a value of all of the variables in our BN. And what's tricky is that we want to do a maximum likelihood estimate of **all parameters** at the same time. The good news is that our BN factorization lets us decompose our likelihood function: we have

$$L(\theta, \mathcal{D}) = \prod_{j=1}^{m} \hat{p}(\vec{x}^{(j)}) = \prod_{j=1}^{m} \prod_{i=1}^{n} \theta_{x_i^{(j)}|\text{Pa}(x_i^{(j)})},$$

so we can swap the order and rewrite this as

$$L(\theta, \mathcal{D}) = \prod_{i=1}^{n} \prod_{j=1}^{m} \theta_{x_i^{(j)}|\text{Pa}(x_i^{(j)})},$$

where our goal is to maximize L (over θ) given that $\theta_{x_i|\text{Pa}(x_i)} \ge 0$, and for each possible choice of values of the parents $\text{Pa}(x_i)$, we have $\sum_{x_i} \theta_{x_i|\text{Pa}(x_i)} = 1$. This is done with the same machinery as in our coin flip example: let $N(x_i, \text{Pa}(x_i))$ be the number of times the tuple of values $(x_i, \text{Pa}(x_i))$ appears in our data set. Grouping those together (so now we don't sum over our different data points j, just like how we grouped together all of the heads and tails), we can now write

$$L(\theta, \mathcal{D}) = \prod_{i=1}^{n} \prod_{\text{Pa}(x_i)} \prod_{x_i} \theta_{x_i|\text{Pa}(x_i)}^{N(x_i, \text{Pa}(x_i))}.$$

But now our optimization problem is

$$\max_{\theta} L(\theta, \mathcal{D}) = \prod_{i=1}^{m} \max_{\theta_i} \prod_{\text{Pa}(x_i)} \prod_{x_i} \theta_{x_i|\text{Pa}(x_i)}^{N(x_i, \text{Pa}(x_i))};$$

notice that the term inside for each i **only depends on the conditional probability distribution for** X_i! So we can go through every variable and separately choose its local probability table to maximize the likelihood, and that will give us the best solution overall. And the only constraint we have on θ now is that for each "row of the conditional probability distribution" (that is, for each choice of the parent values), we need the sum of those entries (the conditional probabilities of x_i) to sum to 1. And thus this ends up being the problem from Fact 79: we fill in the conditional probability tables, choosing large θs for entries that show up often in our data. Summarizing everything, **if we have fully observed data** $\mathcal{D}$, then our optimal choice is

$$\theta^*_{x_i|\text{Pa}(x_i} = \frac{N(x_i, \text{Pa}(x_i)}{N(\text{Pa}(x_i))}.$$

In other words, a potentially complicated problem has been simplified significantly to just doing everything locally – there's nothing mathematically better because of the structure of a BN. (But when we talk about undirected models later, things get more complicated.)

Example 81

Suppose we have a Bayesian network with three true/false variables H, S, E (being health-aware, smoking, and exercising) such that the network structure is given by $H \to S, H \to E$. Also, suppose we have a survey of 16 people, such that we take on the values (T, T, T) twice, (T, F, T) nine times, (T, F, F) once, (F, T, F) once, and (F, F, T) twice, and (F, F, F) once (and (T, T, F) and (F, T, T) zero times).

Our goal is to estimate the conditional probability tables – that is, we want the distribution of H, of S given H, and of E given H. Since 12 of these data points have $H = T$ (health-aware) and the other 4 have $H = F$, we estimate $\theta^{\text{ml}}_H = \frac{3}{4}$. Similarly, out of the 12 health-aware people, 2 smoke and 10 do not, so $\theta^{\text{ml}}_{S|H=T} = \frac{1}{6}$; however out of the 4 non-health-aware people we have 1 smoker, so $\theta^{\text{ml}}_{S|H=F} = \frac{1}{4}$. A similar process gives us the distribution of E.

Remembering now that we don't always want the full density estimation question, we can generalize these ideas:

Definition 82

A **loss function** $\text{loss}(\vec{x}, \mathcal{M})$ is a function that measures some loss made by the model on some instance of the variables $\vec{x}$.

Our goal is then to **minimize loss** (also called **risk**) $\mathbb{E}_{\vec{x}\sim p^*}[\text{loss}(\vec{x}, \mathcal{M})]$ over our class of models – in the cases so far, we've been setting our loss function to be **log-loss** $\text{loss}(\vec{x}, \hat{\mathcal{M}}) = -\log \hat{p}(\vec{x})$. And because we still don't know p^*, we try to approximate by minimizing **empirical risk**

$$\mathbb{E}_{\mathcal{D}}\left[\text{loss}(\vec{x}, \hat{M})\right] = \frac{1}{|\mathcal{D}|}\sum_{\vec{x}\in\mathcal{D}} \text{loss}(\vec{x}, \hat{M}).$$

Example 83

In the spam email classification example (in general, in a discriminative model), our goal has been to predict a set of variables $\vec{Y}$ given some other ones $\vec{X}$. In such a setting, we don't need the whole joint distribution – our loss function is instead the **conditional loss**

$$\text{loss}(\vec{x}, \vec{y}, \hat{\mathcal{M}}) = -\log \hat{p}(\vec{y}|\vec{x}),$$

and thus we only need to estimate the conditional distribution (we don't need to model or estimate $p(\vec{x})$).

This is indeed what we use to train conditional random fields and logistic regression classifiers. Recall that in a logistic regression classifier, we assume the functional form $p(Y = 1|\vec{x}; \alpha) = \sigma(z(\alpha, \vec{x}))$ and we want to pick the model with the smallest loss. Rephrased in our notation, our goal is optimize the coefficients α: we try to solve the problem

$$\min_{\alpha} \mathbb{E}_{(\vec{x},\vec{y})\sim p^*} \left[-\log \hat{p}(\vec{y}|\vec{x}; \alpha)\right].$$

This strategy generalizes to continuous random variables as well – we often represent such a variable X with a probability density function $p_X : \mathbb{R} \to \mathbb{R}^+$, and we do so usually with parameterized densities like a Gaussian $X \sim N(\mu, \sigma) \iff p_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$ or uniform distribution $X \sim U(a, b) \iff p_X(x) = \frac{1}{b-a} 1\{a \le x \le b\}$. The chain rule, Bayes' rule, and other probability laws still apply, and thus we can still use Bayesian networks if we have some or all continuous random variables.

Example 84

Consider a network $Z \to X$, where Z is Bernoulli with parameter p and X is distributed as a Gaussian $N(\mu_0, \sigma_0)$ when $Z = 0$ and as a different Gaussian $N(\mu_1, \sigma_1)$ when $Z = 1$. Then the only parameters we have here are $p, \mu_0, \sigma_0, \mu_1, \sigma_1$.

Example 85

In linear regression, we want to model the relationship between some "covariate" inputs and some "response" outputs. If we have some training data $\mathcal{D} = \{(x_1, y_1), \cdots, (x_m, y_m)\}$ and want to predict Y given X, then it's easier to model $p(Y|X)$ instead of $p(X, Y)$.

We can model this by assuming that $p(Y|X = x)$ is Gaussian with mean $\alpha + \beta x$ and some known **fixed** variance σ^2 (so basically we have Gaussians centered around some best-fit line). More generally if we have multiple covariates, we can assume $p(Y|\vec{X} = \vec{x})$ is Gaussian with some mean $\theta_{\vec{X}}$ and fixed variance σ^2; our conditional likelihood of seeing a particular data point is then

$$p(y_1, \cdots, y_m|\vec{x}_1, \cdots, \vec{x}_m, \theta) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_i - \theta\vec{x}_i)^2\right).$$

Doing some algebra shows that we actually end up reducing problem to the usual **least-squares regression loss**. But even if we have different, more complicated regression models, our machinery here applies to them as well.

8 January 30, 2024

We'll dive fully into inference today – we know everything we need about representation, and our goal is now to solve tasks of interest and compute conditional probabilities. The idea now is that we have some probabilistic model given to us over a set of variables $\mathcal{X}$, and we want to answer conditional probability queries of the form

$$p(\vec{Y}|\vec{E} = \vec{e}) = \frac{p(\vec{Y}, \vec{e})}{p(\vec{e})}.$$

(For example, $\vec{E}$ may be a set of observed votes, and $\vec{Y}$ may be whether a Congressman is a Democrat or Republican.) There will often be some set $\vec{W} = \mathcal{X} \setminus (\vec{Y} \cup \vec{E})$ of other variables which are neither part of the query nor the evidence, so we end up having to marginalize over these other variables:

$$p(\vec{Y}, \vec{e}) = \sum_{\vec{w}} = p(\vec{Y}, \vec{e}, \vec{w}), \quad p(\vec{e}) = \sum_{\vec{y}} p(\vec{y}, \vec{e}),$$

where our sums (or integrals if we have continuous random variables) are over all possible values that the variables $\vec{W}$ or $\vec{Y}$ can take, respectively. Such a sum is expensive but sometimes needed, even though it's not something we explicitly get from a Bayesian network or MRF without computation.

A similar query we may see in inference is the **maximum a posteriori** or most probable explanation, known as **MAP inference**. In such a question, we may have $\vec{Y} = \mathcal{X} \setminus \vec{E}$, and our goal is to calculate

$$\text{MAP}(\vec{Y}|\vec{E} = \vec{e}) = \text{argmax}_{\vec{y}} p(\vec{y}|\vec{e}).$$

So instead of summing, we just find the most likely assignment to the non-evidence variables (this is an optimization problem). Additionally, we may combine these two into a **marginal MAP** problem where we have to calculate both with summation and optimization:

$$\text{MMAP}(\vec{Y}|\vec{E} = \vec{e}) = \text{argmax}_{y} \sum_{\vec{z}} p(\vec{y}, \vec{z}|\vec{e}).$$

In the next couple of lectures, we'll see how these queries can be intractable in general but efficient under some assumptions, and the different situations we'll encounter will require different algorithms.

Proposition 86

Probabilistic inference is hard in the worst-case. Specifically, there does not exist an efficient algorithm in general unless P = NP.

Intuitively, even a simple BN or MRF is "too powerful" in that it can describe a very general set of probability distributions.

Proof. We'll take a canonical NP-complete problem (SAT) and encode it into probabilistic inference in a graphical model. As a reminder, **SAT** (the satisfiability problem) is a situation where we're given a logical formula defined on n binary variables $X_1, \cdots, X_n$

$$(\neg X_3 \vee \neg X_2 \vee X_3) \wedge (X_2 \vee \neg X_4 \vee \neg X_5) \wedge (X_1 \vee X_2 \vee X_3) \wedge \cdots,$$

where each disjunction term is called a **clause** (consisting only of logical ORs) and the disjunctions are combined with logical ANDs. Our goal is to make this formula satisfied, meaning that we want to assign each X_i either "true" or "false" and make the whole statement true.

The simplest case is **3-SAT**, where each clause is defined on at most 3 variables; this is a canonical NP-complete problem. In other words, if we can find an efficient algorithm for this, then we would also have an efficient algorithm for many other problems; we don't expect that this will be the case. (What's expected is that the best way is basically brute-force over all exponentially many assignments of truth values.)

But we can reduce satisfiability to MAP inference in the following way: construct a Markov random field where we have one variable $X_i \in \{0, 1\}$ for each variable in the original SAT formula. Then for each clause C_i, we assign a factor f_i which is 1 if and only if C_i is satisfied. For example, the clause $X_2 \vee \neg X_3$ takes value $f(X_2, X_3) = 1$ if that statement is true, meaning that $f(0, 1) = 0$ but $f(1, 1) = f(1, 0) = f(0, 0) = 1$. Then the joint distribution defined by this graphical model assigns $p(X_1, \cdots, X_n)$ a positive value if and only if all clauses are satisfied (since we multiply indicators of all satisfiability conditions). Thus in summary, the MAP assignment

$$\text{argmax}_{X_1,\cdots,X_n} p(X_1,\cdots,X_n)$$

is at least as hard as 3-SAT for a general probability distribution; in other words, it is NP-hard. A similar reasoning works for Bayesian networks as well (exercise). $\square$

Remark 87. *Notice also that the normalization constant Z in the in the above MRF is the total number of satisfying assignments, so similarly the ability to calculate Z efficiently is hard (because the check of "whether $Z \geq 0$" is what 3-SAT is asking for).*

However, do note that NP-hardness only tells us that there **are** difficult inference problems where these queries are computationally intensive. Often real-world problems will not be as hard as these examples, and in fact doing inference in something like a hidden Markov model is easy – we can use dynamic programming and other techniques to do inference in linear time if there is a **tree-structured** graph.

More generally, the trick we'll use for something like a Bayesian network is to **cache computations** so that we don't have to do them exponentially many times, which will make the queries much more efficient. Recalling the strategy with "pushing the sums inward" using the distributive law from the discussion after Example 19, we can now come up with an algorithm which works for any graphical model but has better running time depending on the **graph structure**.

Example 88

We'll start with the Bayesian network case, understanding how to compute marginal probabilities $p(X_i)$. (From there, we'll generalize to our other graph structures.) Suppose in the simplest case that our BN looks like $A \to B \to C \to D$, and we want to calculate $p(D)$ (that is, the values of $p(D = d)$ for all $d \in \text{Val}(D)$).

Recall that the Bayesian network only gives us $p(A), p(B|A), p(C|B)$, and $p(D|C)$, so we need to marginalize and find

$$p(D) = \sum_{a,b,c} p(A = a, B = b, C = c, D).$$

In general this can take a long time to compute because we have $\text{Val}(A)\text{Val}(B)\text{Val}(C)$ possibilities to sum over just to find a single number, but because of our BN structure we can make use of the distributive law: the above sum simplifies to

$$\sum_{a,b,c} p(a)p(b|a)p(c|b)p(D|c) = \sum_c \sum_b \sum_a p(D|c)p(c|b)p(b|a)p(a),$$

and now pushing in the sums further simplifies this to

$$\sum_c p(D|c) \sum_b p(c|b) \sum_a p(b|a)p(a).$$

So now instead of repeatedly having to do the inner computations, we will store the inner value $p(b|a)p(a)$ as some function $\psi_1(a,b)$, and we'll store the sum over a as some function $\tau_1(b)$. Then we can store $p(c|b)\tau_1(b)$ as a function $\psi_2(b,c)$ and the sum over b as a function $\tau_2(c)$, and so on; this caching is where dynamic programming comes in.

Notice that the size of $\psi_1(A,B)$ only requires $\text{Val}(A)\text{Val}(B)$ entries, and $\tau_1(B)$ similarly only requires $\text{Val}(B)$ entries; thus all of this is more efficient than requiring the full table. Indeed, in a general chain where we have a n variables $X_1 \to X_2 \to \cdots \to X_n$ where each variable has k possible states, our algorithm is as follows. First set

$\tau_1(x_1) = p(X_1 = x_1)$ (which is given to us by the BN), and now for each i we compute and cache

$$\tau_{i+1}(x_{i+1}) = \sum_{x_i} p(X_{i+1}|x_i)\tau_i(x_i).$$

(Here τ_i is the probability distribution of X_i.) Each step takes $O(k^2)$ time here (since we have k values to calculate, and each involves a sum over k terms), so the total running time is $O(nk^2)$. This is much more efficient than doing a naive sum over all variables, which would have complexity $O(k^n)$. And notice that this process didn't ever require us to construct the joint distribution – we used the conditional independence structure to our benefit!

Example 89

Now suppose we have a Bayesian network with n variables $\vec{X} = (\vec{Y}, \vec{Z})$, and our goal is to do a marginalization of the form

$$p(\vec{y}) = \sum_{\vec{z}} p(\vec{y}, \vec{z}) = \sum_{\vec{z}} \prod_{x_i \in \vec{X}} p(x_i|\vec{x}_{\text{Pa}(x_i)})$$

for all $\vec{y} \in \text{Val}(\vec{Y})$.

We'll think of the conditional probability distributions as factors – our algorithm will end up not caring about normalization, so we might as well describe everything in terms of MRFs so that we have more generality. So we can rephrase our goal as completing the **sum-product** inference task

$$\tau(\vec{y}) = \sum_{\vec{z}} \prod_{\phi_i \in \Phi} \phi_i(x_i, \vec{x}_{\text{Pa}(x_i)}),$$

where ϕ_i is exactly the conditional probability distribution for X_i (remembering that this involves "marrying the parents" and creating a clique, so the scope of ϕ_i is X_i and its parents). For example, if we have a Bayes net $A \leftarrow B \rightarrow C \leftarrow D$, then we can think of wanting to compute

$$p(c) = \sum_{a,b,d} \phi_1(b)\phi_2(d)\phi_3(a, b)\phi_4(c, b, d),$$

since we want to marginalize over all variables except C and each factor corresponds to a node and its parents. Our algorithm (**variable elimination**) will do two types of operations – we'll multiply factors and marginalize variables. We think of conditional distributions or factors as tables, where summing over a variable x produces a new factor where the table "doesn't depend on x." If we have a factor ϕ involving $\vec{X}$ and another variable B, then **factor marginalization** of ϕ over B gives us a factor

$$\tau(\vec{X}) = \sum_B \phi(\vec{X}, B).$$

(For example, if we have a table of values for a factor $\phi(a, b, c)$ and want to marginalize over b, then we add up all terms with the same a and c over all possible values of b. This can be thought of as taking a 3D array and "squashing" along one dimension.) Also, if we have two factors, we can also find the **factor product** by creating a new factor where we multiply the values together: for example $\phi_1(a, b)$ and $\phi_2(b, c)$ can be combined as $\phi_3(a, b, c) = \phi_1(a, b)\phi_2(b, c)$. (But for tractability, we want relatively small factors so that these tables stay small.)

These two factor operations are enough for us to describe our algorithm for computing $\tau(\vec{Y}) = \sum_{\vec{Z}} \prod_{\phi_c \in \Phi} \phi_c(\vec{X}_c)$:

- Order the variables $\vec{Z}$ in some order $Z_1, Z_2, \cdots, Z_k$ which we call the **elimination ordering**. (This ordering matters significantly for runtime.)
- For each i, we "eliminate Z_i" by "pushing the sum over Z_i' inward as much as we can." We do this by taking

the **factor product** of all factors with Z_i in their scope, which generates a new product factor (which we hope is relatively small). Then do **factor marginalization** over Z_i.

- Add this new factor to the set of factors and remove all factors that had Z_i in their scope originally.

If we keep repeating this, we'll end up with the output $\tau(\vec{Y})$ that we're looking for.

Example 90

Suppose we have a joint distribution given by a Bayesian network as

$$p(C, D, I, G, S, L, H, J) = p(C)p(D|C)p(I)p(G|D, I)p(L|G)P(S|I)p(J|S, L)p(H|J, G).$$

Our goal is to compute $\tau(J)$, which is the sum of $p(C, D, I, G, S, L, H, J)$ over all variables besides J.

Converting conditional probabilities to factors (for example $p(G|D, I)$ is now $\phi_G(G, D, I)$), we need to compute

$$\sum_{c,d,i,g,s,\ell,h} \phi_C(C)\phi_D(D, C)\phi_I(I)\phi_G(G, D, I)\phi_L(L, G)\phi_S(S, I)\phi_J(J, S, L)\phi_H(H, J, G).$$

We do so by picking some ordering – if we eliminate variable C, we would only need to multiply together the factors $\phi_C(C)$ and $\phi_D(D, C)$ into a single factor and then marginalize to get $\tau_1(D)$. (So we replace those two previous factors with just this one factor τ_1.) Now, we can eliminate variable D by multiplying together $\phi_G(G, D, I)$ and $\tau_1(D)$ and then marginalizing over D; this replaces those two factors with just the new factor $\tau_2(G, I)$. We continue on in this process, repeatedly replacing some set of factors with a new one τ_i and getting rid of one variable at a time, until we just have a factor $\tau_7(J)$ which is our desired answer.

Fact 91

In this case, we can check that (C, D, I, H, G, S, L) is a pretty good elimination ordering, but (G, I, S, L, H, C, D) does a much worse job because we have to multiply together more factors and thus create very large factors τ_i (storing and clearing tables takes lots of time and memory). And if we can't find a good elimination ordering, we won't get computational savings – this is consistent with our observation that this problem is hard in the worst case.

Remember now that we often want to calculate **conditional** probability queries – if our goal is to find $p(\vec{Y}|\vec{E} = \vec{e})$, what we can do is apply variable elimination to the task $p(\vec{Y}, \vec{e})$, but we "clamp" the factors on the observed values of E by only taking the corresponding slices in the table. So instead of having a factor $\phi \in \Phi$, we instead define the factor abstractly as

$$\phi'\left(\vec{x}_{\text{scope}(\phi)\setminus\vec{E}}\right) = \phi\left(\vec{x}_{\text{scope}(\phi)\setminus\vec{E}}, \vec{e}_{\text{Scope}(\phi)\cap\vec{E}}\right)$$

(intuitively, all the variables in $\vec{E}$ are treated as known constants, and all variables not in $\vec{E}$ are still variables in the factors) and then eliminate all variables other than $Y \cup E$ to get $p(\vec{Y}, \vec{E})$. Then finally we do a **second** round of variable elimination to sum over Y and get $p(\vec{e})$, which is the denominator required for our conditional probability.

Example 92

Suppose from the previous example that we want to compute the distribution $p(J|G = 2)$. Just like before, we do variable elimination, but our factors are now simplified by removing the dependence on G:

$$\Phi = \{\phi_C(C)\phi_D(D, C)\phi_I(I)\phi_G(G = 2, D, I)\phi_L(L, G = 2)\phi_S(S, I)\phi_J(J, S, L)\phi_H(H, J, G = 2)\}$$

(In other words, the blue factors come from only taking the rows of the table where G takes the correct value.)

So now we can do variable elimination over everything except J and G (G isn't even showing up in the factors), which allows us to find $p(J, G = 2)$. Then marginalizing yields $p(G = 2)$, and dividing those two quantities gives us the conditional probability we want. (And this algorithm works for undirected models too – we just skip the first step of converting CPDs to factors.) In particular, this is a nice algorithm for computing the normalization constant

$$Z = \sum_{\hat{x}_1, \cdots, \hat{x}_n} \prod_{c \in C} \phi_c(\hat{x}_c)$$

in an MRF, since it's of the sum-product inference form.

To understand the running time of this algorithm, suppose we have n variables and m initial factors in our set Φ. (For a Bayesian network, we have $m = n$, since we get a factor for each node.) At every step of our algorithm, we pick some X_i, multiply all factors together, and get a single factor ψ_i. The complexity is then bottlenecked by the **largest number of variables we ever have in a single factor** ψ_i; if we call this number N_{max}, then we can bound the runtime by $O(nv^{N_{\text{max}}})$, where v is the maximum number of values that any X_i takes on. So if N_{max} is comparable to n, we have an expensive computation, but if we can keep N_{max} small using a good elimination ordering, then this gives us an efficient strategy.

Fact 93

It is actually NP-hard to find the most efficient elimination ordering; we'll talk about this more in the next lecture. Luckily, there are some heuristics that give us good strategies, such as locally avoiding choices that give us big factors.

We can state all of this in terms of the graph structure as well: we start with an undirected graph G_Φ with an edge (X_i, X_j) if the two variables **both appear together in the scope of some factor**. (So if we don't have any evidence to condition on, this is the moralized Bayesian network or the MRF that we started with.) And during the process of variable elimination, creating factors ψ forces us to add edges between different variables that were merged together (called **fill edges**), but then we remove a variable and its incident edges. So the hope is to avoid adding too many of these edges and keeping the graph relatively sparse.

9 February 1, 2024

Today, we'll continue our discussion of exact inference methods – last time, we discussed variable elimination, a method for computing marginal or conditional probabilities or evaluating a normalization constant for an undirected model (or the moralization of a directed one). The idea is that instead of summing over all exponentially many possibilities, we make use of the compact representation as a product of factors and "push in the sums" as much as possible one variable at a time.

But the tricky part is that we need to pick a nice elimination ordering to avoid having our factor products be too large (we end up with larger scopes). With something like a hidden Markov model, there's a natural efficient ordering to use, but in general it's hard to see whether there's a way to keep $N_{\max}$ (the number of factors that arise in a ψ_i) small. At the end of last lecture, we thought about this in terms of graphs – at each step, we eliminate a variable X by creating a new factor containing all other variables $\vec{Y}$ that X was also involved in. So if we remove a leaf from a graph (only connected to one other vertex), we don't have to add any new edges, but if we remove a node connected to 10 other nodes, then our graph now has a clique of 10 vertices, where we may have added in some extra **fill edges**.

To summarize the computation cost, we can thus use a single graph which is the **union of all graphs that were created during the elimination process**; we call this the **induced graph** $\mathcal{I}_{\Phi,\prec}$ (which depends both on the set of factors Φ and on the elimination ordering $\prec$). In other words, there is an edge between X_i and X_j if the two variables appeared in a common factor at some point. Then the complexity of VE (variable elimination) is captured in the following way:

Theorem 94

Every factor generated during variable elimination has a scope which is a clique in $\mathcal{I}_{\Phi,\prec}$; conversely, every **maximal** clique in $\mathcal{I}_{\Phi,\prec}$ is the scope of some intermediate factor formed during VE. Therefore $N_{\max}$ (the maximum number of variables involved in any factor) is exactly the size of the largest clique in $I_{\Phi,\prec}$, and the running time of the algorithm is $O(mv^{N_{\max}})$ (where v is the maximum number of values that a variable can take on).

Definition 95

The **width** of an induced graph is the number of nodes in the largest clique minus 1. For a graph $\mathcal{G}$ and ordering $\prec$, the **induced width** $w_{\mathcal{G},\prec}$ is the width of the induced graph $\mathcal{I}_{\mathcal{G},\prec}$, and the **treewidth** of a graph $\mathcal{G}$ is the minimal possible induced width over all orderings:

$$w_{\mathcal{G}}^* = \min_{\prec} w_{\mathcal{G},\prec}.$$

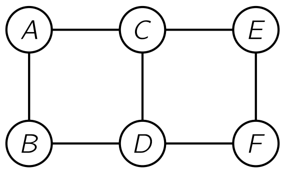
We call this "treewidth" because it tells us "how much the graph looks like a tree," since graphs without cycles are easiest to do inference with. This is a well-defined quantity that just depends on the graph, though it is computationally difficult to compute (finding the treewidth is NP-hard). However, we can still generally aim to find good elimination orderings in practice by doing a **min-neighbor** strategy (picking nodes with few neighbors) or **min-fill** (introducing few new fill edges).

Example 96

The treewidth of a star graph (with a central node and a bunch of other nodes only connected to that central node) is 1. Indeed, we choose an elimination ordering where we remove all nodes besides the central node first (so that we don't create new edges); this never introduces any new fill edges so the largest clique in the induced graph has size 2.

Example 97

Next, consider the following diagram which is "almost like a chain graph:"



We can think of this as almost being a "tree of width 2" (we have a tree on the top and the bottom), and indeed if we remove the variables in the order (A, B, C, D, E, F), we only add two new edges (from B to C, then from D to E). This means the largest clique has size 3 and thus the treewidth is 2. And this process works even if we extend the graph horizontally – we are basically using the same dynamic programming trick.

We'll now generalize to trying to solve **multiple inference tasks** on the same graph, such as computing both $p(X)$ and $p(Y)$. One naive strategy is to do the two tasks separately, but often we can get some savings by caching intermediate computations – we can think about this in terms of **message passing** or **belief propagation**. It turns out that in the special-case of tree-structured graphs, we can efficiently compute **all** single-variable marginals (using the **sum-product algorithm**), and then we'll generalize this in a future lecture to larger cliques using the **junction-tree algorithm**.

Example 98

Consider the simple graphical model $A \to B \to C \to D$ from Example 88. Remember that the calculation in that case corresponded to the elimination ordering (A, B, C, D), but now we can think about marginalizing over A as using the factor $\tau_1(b)$ as "passing on a message" that tells us how we should modify our belief about what states B should take. Then when we marginalize over B, we pass on a message to variable C.

Even when we have a different tree structure, this reasoning holds as well. If we do variable elimination on the graph $A \to C, B \to C, C \to D$, then marginalizing over A passes on $m_{AC}(c)$ and marginalizing over B passes on $m_{BC}(c)$, so that when we eliminate C we need to do the computation

$$m_{CD}(d) = \sum_c \phi_C(c)\phi_{DC}(D, c)m_{AC}(c)m_{BC}(c)$$

where we factor in the local factors and also the information passed on from previous variables. (And we could account for these terms in this way because **conditioned on C, the previous variables A, B are independent of the "new effect" that D has on C** – this wouldn't be exact if we had additional edges that create cycles.) So letting $\mathcal{N}(j)$ denote the neighbors of a node j, we can think of variable elimination as passing on messages of the form

$$m_{ji}(x_i) = \sum_{x_j} \phi_j(x_j)\phi_{ij}(x_i, x_j) \prod_{\ell \in \mathcal{N}(j) \setminus i} m_{\ell j}(x_j).$$

(that is, send a message to the parent which combines the information from your neighbors), and it tells us that we should always eliminate variables from leaves to root – something like $m_{kj}(x_j)$ encodes the entire subtree rooted at k.

Fact 99

A note of warning is that these messages m do not generally encode probability distributions, because they are not normalized in general.

And finally, our final marginal at some node is computed via

$$p(x_f) \propto \phi_f(x_f) \prod_{\ell \in \mathcal{N}(x_f)} m_{\ell f}(x_f)$$

(we just need to figure out the normalization at the end by summing this over all x_f). The reason this is nice is that **if we tried to compute marginal probabilities for each node, we repeatedly generate the same messages.**

Example 100

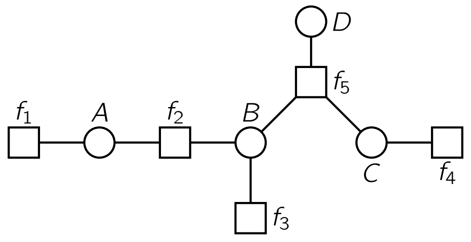
If we have a star graph with central node A and other nodes B, C, D, then the message $m_{BA}(a)$ will be generated when we're trying to find the marginal distribution for any of the nodes A, C, D. So even though we need to compute 3 messages for each of the 4 marginal distributions, we actually only need 6 messages overall ($m_{AB}, m_{BA}, m_{AC}, m_{CA}, m_{AD}, m_{DA}$). In general for a tree with n nodes, this means we only need to compute $2(n-1)$ instead of $n(n-1)$ messages.

In other words, this message passing protocol just requires the set of incoming messages. But if we are trying to compute some message $m_{ji}(x_i)$ from j to i, we can only send this message if we **already** have all of the incoming messages into j. This can be thought of as a distributed parallel algorithm – each node has a processor that polls its neighbors, and it **only sends its message to some neighbor when messages from all other neighbors have been received**. So if we start off with a tree, then the only nodes that can start sending messages are the leaves, and then from there we propagate until all messages have been sent. This process will only take $O(|E|)$ steps, but again remember it's only for trees (because otherwise we don't have conditional independence).

This process works very similarly if we have a factor graph representation as well: for a tree, this means we have a factor node connected to each node, as well as a factor node for each edge of the tree (connected to both vertices). In this representation, we can think of factor nodes as sending messages to variable nodes and variable nodes as sending messages to factor modes; nodes again only send messages when messages have been received from all other neighbors.

Example 101

Consider the following factor graph (where squares denote factors and circles denote variables), and suppose we want to marginalize over the variables A, B, C.



We must do the computation

$$\sum_{a,b,c} f_1(a)f_2(a,b)f_3(b)f_4(c)f_5(b,c,d),$$

and in this new context we can think of $f_1(a)$ as being a message $\mu_{f_1\to A}(a)$ that is "sent from f_1 to A" (and similarly the unary factors f_3, f_4 being ready to send their messages to B and C). So now A can send a message to f_2

$$\nu_{A\to f_2}(a) = \mu_{f_1\to A}(a)$$

and then f_2 sends a message to B by taking all incoming messages

$$\mu_{f_2\to B}(b) = \sum_a \nu_{A\to f_2}(a)f_2(a,b).$$

Now B is ready to send a message by multiplying its incoming messages

$$\nu_{B\to f_5}(b) = \mu_{f_3\to B}(b)\mu_{f_2\to B}(b),$$

and we continue on until we have sent a message up to D of the form

$$\mu_{f_5\to D}(d) = \sum_{b,c} \nu_{C\to f_5}(c)\nu_{B\to f_5}(b) f_5(b, c, d).$$

So there are basically messages from variables to factors (which we denote ν) and messages from factors to variables (which we denote μ), corresponding to different update rules. When we send a message of the first type, we multiply messages together (except from the recipient)

$$\nu_{\text{var}(i)\to\text{fac}(s)}(x_i) = \prod_{t\in\mathcal{N}(i)\setminus s} \mu_{\text{fac}(t)\to\text{var}(i)}(x_i),$$

and when we send a message of the second type, we multiply the incoming messages with the local factor, and then we sum out all variables except the recipient to get

$$\mu_{\text{fac}(s)\to\text{var}(i)}(x_i) = \sum_{\vec{x}_{\mathcal{N}(s)\setminus i}} f_s(\vec{x}_{\mathcal{N}(s)})' \prod_{j\in\mathcal{N}(s)\setminus i} \nu_{\text{var}(j)\to\text{fac}(s)}(x_j).$$

So information is propagating in basically the same way as variable elimination or the previous description we had, but this is just a more general way of describing things at the level of factors and messages. And we'll see how this applies beyond trees next time!

10 February 2, 2024 (Section)

Today's section is on **variable elimination**, taught by Garrett Thomas. We'll first explain broadly what we mean by "inference," and then we'll discuss the algorithm and work through some examples.

In the first few weeks of the class, we've largely discussed **representation** (encoding things as Bayesian networks or Markov networks, understanding the independencies encoded by them). Our next topic is to take a given graph and answer interesting probabilistic queries. For example, if we have a joint distribution between Y and some random variables $\vec{X}$, then we might be interested in **marginal inference**, trying to figure out the probabilities for a single given variable

$$p(y) = \sum_{x_1,\cdots,x_n} p(y, x_1, x_2, \cdots, x_n).$$

We may also be interested in **conditional probability queries** of the form $p(y|x_1, \cdots, x_n)$ (for example, trying to classify into one of a set of labels given some features or evidence), or **maximum a posteriori (MAP) inference**, in which we want to find the values of $x_1, \cdots, x_n$ which maximize that conditional probability. Finally, we may marginalize over some variables and be interested in MAP inference on the remaining ones.

The problem is that (as we saw in lecture) inference is tough in the worst case. We saw this by reducing to 3-SAT (which is NP-complete); this means that in general we need to use useful properties of our graph structure to make progress. Variable elimination is one such method for doing exact inference. (And later on in the course, we'll discuss some approximate methods as well, which may be more efficient but not exact.)

Example 102

Recall from lecture that in a chain Bayesian net with variables $X_1, \cdots, X_n$ where X_i only depends directly on X_{i-1}. Then the naive method for computing the marginal probability $p(x_n)$ (which always works) is to sum over all other variables, which takes $O(d^n)$ time.

The way to improve our calculation is to "push the sums in" as deep as possible after decomposing the joint distribution: algebraically, this looks like

$$p(x_n) = \sum_{x_{n-1}} p(x_n|x_{n-1}) \sum_{x_{n-2}} p(x_{n-1}|x_{n-2}) \cdots \sum_{x_1} p(x_2|x_1)p(x_1).$$

(We can also think of this as "factoring out terms" by pulling them to the left as much as possible.) But now if we compute the factors one by one, we can first calculate

$$\tau(x_2) = \sum_{x_1} p(x_2|x_1)p(x_1)$$

in $O(d^2)$ time (since we need to sum over d products for each value of x_2, and there are d values), and that lets us calculate the "next innermost sum" $\sum_{x_2} p(x_3|x_2)\tau(x_2)$ again in $O(d^2)$ time. Repeating this until we finish computing all sums (that is, eliminating variables one at a time) means that we can finish the whole inference query in $O(nd^2)$ time, which is linear instead of exponential in n.

More generally, we can do this exact algorithm for a Markov network too. The idea is to take our decomposition as a product of factors

$$p(x_1, \cdots, x_n) = \frac{1}{Z} \prod_{c \in C} \phi_c(\vec{x}_c),$$

where a factor is basically a big table assigning values to all possible variables in its scope (which requires $d^{|C|}$ entries). Remember that Bayesian networks can be thought of in this way as well, where we convert each conditional probability table to a factor with normalization constant $Z = 1$. So the process of multiplying terms together generalizes to the idea of a "factor product"

$$\phi_3(\vec{x}_c) = \phi_1(\vec{x}_c^{(1)})\phi_2(\vec{x}_c^{(2)}),$$

and summing over the values for a given variable is the "elimination process" that yields a new factor

$$\tau(\vec{x}) = \sum_y \phi(\vec{x}, y).$$

Example 103

If we imagine actually implementing these two operations (creating new tables out of old factor tables), the only complication is that we need "common variables" between the two cliques that we're multiplying to match up. So if we multiply $\phi_1(A, B)$ with $\phi_2(B, C)$, we only multiply together entries where B takes the same value in the two factor tables.

To specify how the variable elimination process works, we are removing one variable at a time, and we need to describe the order in which the variables are removed. The runtime depends significantly on the ordering, and figuring out the best one is NP-hard. Instead, we may use one of a set of **greedy algorithms** (which have some heuristic function that we want to minimize when we choose the next variable): **min-neighbors** tells us to choose a variable with the fewest number of neighbors, **min-weight** minimizes the product of the cardinalities (number of values) of its dependent variables, and **min-fill** minimizes the size of the new factor that we add to our graph. (Empirically, min-fill turns out to work best on average, but this won't be universally true.)

So to summarize, our algorithm works as follows: given some ordering O, we proceed iteratively through the variables in the ordering and eliminate. To eliminate X_i, we find all factors that contain X_i and multiply them together into an intermediate product factor ϕ, then marginalize over X_i to get τ. This factor τ is what we use to replace the factors that originally contained X_i, so now we have a factor graph on one fewer vertex.

Example 104

One reasonable strategy for VE in a Bayesian network is to use the topological ordering (so that we eliminate all parents of a node before the node itself). However, do note that this can still create additional "fill edges" when a node has multiple children (because we need to draw edges between all pairs of them when we perform the factor product step), and it's not always going to be the most efficient ordering.

In general, the runtime of variable elimination is $O(mv^{N_{\max}})$, where m is the number of variables we need to eliminate (the number of steps we take overall), $N_{\max}$ is the largest size of any factor (at any point during the process), and v is a bound on the number of values that any eliminated variable takes on. So this explains the ideas behind the heuristics we mentioned earlier – each of them is trying to keep this quantity small in a different way.

11 February 6, 2024

Last lecture, we introduced the ideas of belief propagation and message passing on a tree (for answering **multiple inference queries** at once without needing to do the calculation from scratch), and today's goal is to generalize that strategy to structures that aren't just trees. We'll start with loopy belief propagation, a good heuristic (approximation) strategy that works well in practice, and then we'll talk about **junction trees**, which are another useful data structure for exact inference.

Recall that belief propagation is the computation coming from variable elimination, in which we pass messages $m_{kj}(x_j)$ from k to j of the form

$$m_{ji}(x_i) = \sum_{x_j} \psi_j(x_j)\psi_{ij}(x_i, x_j) \prod_{\ell \in \mathcal{N}(j)\setminus i} m_{\ell j}(x_j)$$

once all incoming messages from other nodes have been received. The idea is that if we think of passing information up from the leaves to the root, then the information passed from k to j encodes everything in the subtree of k. (So this is a **parallel protocol** because the nodes send information when prompted.)

We now might ask what happens if we apply belief propagation to a **high treewidth graph with loops**. It's now not clear that the message-passing idea is principled, but we can still try the same idea. We no longer have a natural ordering for the vertices (since we can't define what "leaves" and "roots" are anymore), so instead we try something different in the spirit of "local optimization." **First, we initialize** all messages – for example, we can start by choosing a uniform distribution

$$\delta^0_{i\to j}(X_j) = \frac{1}{|\mathrm{Val}(X_j)|}$$

to send from each node to its neighbors. Then we **repeatedly perform BP until the algorithm converges**, calculating

$$\delta^{t+1}_{i\to j}(X_j) = \sum_{x_i} \phi_{ij}(x_i, X_j)\phi_i(x_i) \prod_{k\in\mathcal{N}(i)\setminus\{j\}} \delta^t_{k\to i}(x_i)$$

until $\delta_{i\to j}^{t+1} - \delta_{i\to j}^{t}$ is sufficiently small for all i, j. (We can think of this as "repeatedly updating our beliefs" until they stop changing significantly.)

Fact 105

In general, this **loopy belief propagation** procedure will not always converge. And even if we do have convergence, we may not necessarily get the correct marginals (depending on our initializations of messages). But this still works extremely well in practice, and it's because these updates are basically solving an optimization problem (we'll learn more about this in a later lecture). For some empirical data, we can see this paper.

Remark 106. *Note that the strategy we introduced at the end of last lecture with factor graphs involved summing over more than one factor at a time, so that corresponds to doing variable elimination and that was an* **exact** *algorithm. Here loopy belief propagation is* **not exact**.

There are some practical issues we may run into when running this algorithm as well. While computing each step is relatively quick (the update is simple and only depends on the neighbors, and we only need to marginalize over a **single** variable), there can be numerical stability issues because of the repeated multiplication of factors. So one thing we can do is to **normalize so our messages always sum to one**: at each step we instead compute

$$\delta_{i\to j}(X_j) = \frac{1}{Z_{i\to j}} \sum_{x_i} \phi_{ij}(x_i, X_j)\phi_i(x_i) \prod_{k\in\mathcal{N}(i)\setminus\{j\}} \delta_{k\to i}^{t}(x_i),$$

which doesn't change the outputs or the estimated marginals because we always normalize at the end anyway. And it can also be sensible to work in the "log-space" $a_i = \log \delta_i$ and use the `logsumexp` function instead of taking sums.

Example 107

Since loopy belief propagation multiplies potentials multiple times, it can often be **overconfident**. For example, suppose we have a graph which is a cycle on 4 vertices A, B, C, D in that order, and the factors between each pair of vertices encourages them to take the same value. Then if we have evidence that B takes value 0 more frequently than 1, then the message $\delta_{B\to A}$ encourages $A = 0$, but then $\delta_{B\to C}$ also encourages $C = 0$, and then when we continue passing messages we will "double-count" the encouragement when we cycle back around, thus believing that the probability of 0 is far larger than the probability of 1 for all variables than in practice.

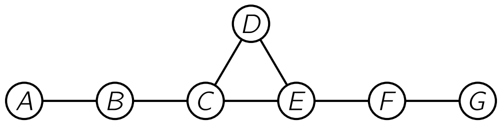
Fact 108

There are certain kinds of graphs where we can prove convergence of loopy belief propagation, such as trees or single-loop graphs or certain other classes of random graphs. But accuracy guarantees often require additional assumptions (such as conditions on the potentials) – this is still an active area of research.

We'll now move to the **junction tree algorithm**, which is again trying to calculate marginal probabilities over a loopy graph but is now yielding an **exact** answer. Recall from variable elimination that the difficulty with cycles is that additional "fill edges" emerge when we eliminate variables. Junction tree basically deals with this by **thinking of a graph as a tree of cliques** – once we have such a structure, we can basically apply message passing in the same way, yielding exact computation of all marginals and guaranteeing "two-pass convergence."

Example 109

For some intuition, imagine a graph which is a path of six vertices A, B, C, E, F, G, but then we have an additional vertex D connected to both C and E, shown below. This is "almost a tree" – if we think of C, D, E as a single "super-variable" $\vec{W}$, then we do indeed have a tree and can think of the equivalent model $p'(A, B, \vec{W}, F, G)$.



Our goal would be to make the cliques as small as possible, since larger cliques take on many more possible values and are thus less computationally efficient. But then the formalism looks similar to what we've seen before for trees: we now have new factors such as

$$\phi'_{BW}(b, \vec{w}) = \phi'_{BW}(b, (c, d, e)),$$

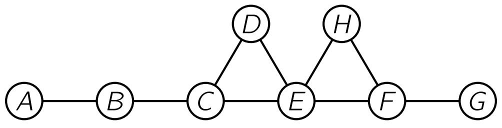
and looking at the original structure of the graph we can choose this to be $\phi_{BC}(b, c)$. Similarly we have $\phi'_{WF}(\vec{w}, f) = \phi'_{WF}((c, d, e), f) = \phi_{EF}(e, f)$, and finally the potential at $\vec{W}$ itself is

$$\phi'_W(\vec{w}) = \phi_{CD}(c, d)\phi_{DE}(d, e)\phi_{CE}(c, e).$$

We can notice that this accounts for all factors that were originally in the MRF definition. (And of course, we could also do things like put the factor $\phi_{CD}(c, d)$ into ϕ_{WF}.)

Example 110

Next, suppose we also add another vertex H that is connected to E and F, as shown below:



One strategy would be to group all of C, D, E, F, H into a single node, but we can also use smaller clusters and define $\vec{W} = (C, D, E)$, $\quad \vec{V} = (E, F, H)$, so that we only have to reason about three variables at a time. The only trouble is that the variable E is "in two different nodes at once," so now we need to make sure that **the values assigned to $\vec{W}$ and $\vec{V}$ are consistent**.

Definition 111

A **cluster tree** T is an undirected tree graph, where each node i is associated to some subset of variables C_i (which we call a **cluster**). Each edge between C_i and C_j is associated with a **sep-set** (separator set) $S_{ij} = C_i \cap C_j$ (used to enforce consistency).

We must choose a cluster tree that is big enough to encode the information about our joint distribution (since we need to be able to account for all factors):

Definition 112

A cluster tree is **family-preserving** for a set of factors Φ if for all $\phi \in \Phi$, there is some cluster C that contains scope$[\phi]$.

(The idea is that if we have a clique on three vertices, we need to be able to jointly reason about them in arbitrary ways, so we need to be able to encode without any additional consistency assumptions.) So now assume we have a model defined by a set of factors Φ of the form

$$p(\vec{x}) = \frac{1}{Z} \prod_{\phi_i \in \Phi} \phi_i(\vec{x}_i),$$

and we have a family-preserving cluster tree T for ϕ. Then on our junction tree, we can define

$$p(\vec{x}) = \frac{1}{Z} \prod_{C \in T} \psi_C(\vec{x}_C),$$

where the factor for a cluster is basically the product of the local factors. We do this by initializing $\psi_C = 1$ for each cluster and then (one by one) multiplying in each potential ϕ_i into some cluster potential ψ_C (such that scope$(\phi_i) \subseteq C$).

Definition 113

A cluster tree T is a **junction tree** if it is a tree, it is family-preserving, and it has the **running intersection** property, meaning that for any clusters V, W, **all clusters on the path between V and W contain $V \cap W$**.

In other words, if a variable x appears in two different clusters V and W we need to be able to "pass the assumption of consistency" along any path connecting them, meaning that x must appear in all clusters between V and W as well.

We can always come up with a (useless) junction tree for any model, since we can always use a single cluster for all variables (it's a tree because there's just one node, it's family-preserving because any scope is contained in that one cluster, and it has the running intersection property because there is only one cluster and no paths). But that doesn't leverage any conditional independence and thus gives us no savings. On the other hand, for a tree, we can easily obtain a small-clique junction tree by defining a cluster for each edge of the original tree.

Fact 114

It's not always immediately obvious how to construct a junction tree, though. If we have a cycle on three vertices A, B, C, then we cannot construct a junction tree by forming pairwise clusters (A, B), (B, C), and (C, A) (since we won't satisfy running intersection). But see Proposition 116 below.

But once we have our junction tree T, we get a natural ordering (with a notion of leaves and roots); since we have no loops, we can combine information in the natural way via message passing. Just like in belief propagation for trees, clusters send a message to each neighbor, and we can only send a message from cluster I to cluster J once I has received messages from all neighbors except J. Then at the end, we can compute the marginal probability over **all** vertices in a cluster I combining the local potential at I with the messages that it received from all of its neighbors. Our message is slightly more complicated, since we need to send it over the **variables in the sep-set**:

$$\delta_{i \to j}(C_i \cap C_j) = \sum_{C_i \setminus C_j} (C_i) \prod_{k \in \mathcal{N}(i) \setminus \{j\}} \delta_{k \to i}(C_k \cap C_i).$$

(We can think of the outside sum as "marginalizing over all variables in C_i that aren't in C_j.) In words, we compute the product of the local potential with the other messages, marginalize everything not in j, and then send the result to j. (Remember that $\delta_{k \to i}(C_k \cap C_i)$ only depends on variables that are already in C_i, so the whole expression inside the sum does indeed only depend on variables in C_i.) As a sanity check, this is indeed the same thing that we did on trees: in a tree-structured model, the intersection between any two clusters is a single variable, so the messages depend only on a single variable and do take the form $\delta_{j \to i}(x_i)$.

Remark 115. *Intuitively,* **this algorithm is just variable elimination** *but now keeping track of a dynamic programming table for each cluster. If we send a message from one cluster C_1 to another C_2 over the sepset S, we're basically passing over the unnormalized marginal distribution where we've eliminated all variables on C_1's side of the junction tree* **except** *those in S.*

We now have a good reasoning for the three assumptions we place on our junction tree: being a **tree** allows us to have a natural order (via choosing a root and passing from leaves to root), being **family-preserving** allows us to represent the same distribution as our original mdoel, and satisfying **running intersection** tells us "when it is safe to eliminate a variable," since the clusters containing a variable form a connected component.

Once this process terminates, beliefs will all be consistent with each other – we say the junction tree is **calibrated**. Then we can do inference easily: we compute **cluster belief** (the marginal probability over the variables in the cluster) by calculating

$$p(C_r) \propto \psi_{C_r} \prod_{k \in \mathcal{N}(r)} \delta_{k \to r}.$$

And if we want the **individual** marginal probabilities after that, we just further marginalize over the other variables in a given cluster. (Notice that variables may appear in multiple clusters, and because our junction tree is calibrated we can compute this marginal with **any** cluster we'd like.)

Proposition 116

We can always get a junction tree "for free" from variable elimination – once we choose an elimination order, variable elimination creates a sequence of cliques coming from combining factors together. So we can choose **one cluster for each product factor produced by variable elimination**, adding an edge between two clusters if the intermediate factor during elimination of one is used when we find the product factor for the other.

This does yield a tree, since every intermediate factor is only used once and thus we cannot have loops. Furthermore, every original factor is multiplied in at some point so this is family-preserving, and running intersection works because variables appear from the moment they are introduced until the moment they are marginalized out. This means that we can choose heuristically good junction trees **using our heuristics for choosing elimination orderings**.

Example 117

Suppose we eliminate a variable X_6 and create a product factor with scope (X_2, X_5, X_6) and an intermediate factor with scope (X_2, X_5). We would then add (X_2, X_5, X_6) to our junction tree. If we eliminate X_5 next and create a new product factor with scope (X_2, X_3, X_5), that makes use of the factor (X_2, X_5) and thus we have an edge between the two clusters (X_2, X_5, X_6) and (X_2, X_3, X_5).

12 February 8, 2024

We'll discuss approximate inference methods, namely **sampling algorithms**, today – instead of variational methods like belief propagation, we'll be thinking about a different class of algorithms to avoid issues with computational intractability. We know that if we have a small treewidth (for example, less than 25) graphical model, then dynamic programming strategies with variable elimination or junction tree methods. But in many applications, maybe approximating a true probability $p(X_i = x_i | \vec{E} = \vec{e}) = 0.29292$ with some approximate inference method and finding that $\hat{p}(X_i = x_i | \vec{E} = \vec{e}) = 0.3$ is good enough. So that opens up the ideas of **Monte-Carlo sampling methods** such as importance sampling or MCMC (Markov Chain Monte Carlo).

The key idea is that instead of working with the full joint distribution of a set of variables, we work with a **representative sample** which allows us to approximate computations with less data. There are many different ways to do this, and in the next few lectures we'll describe some of them.

Definition 118

Let $\vec{X} = (X_1, \cdots, X_n)$ be a set of variables. A **sample** from $\vec{X}$ is an assignment $(x_1, \cdots, x_n)$ of values to all variables.

We also sometimes call samples **instantiations** or **states**. Our goal is to randomly generate a sample $\vec{x}$ according to the probabilities given by the joint distribution P, which we write as $\vec{x} \sim P$.

Example 119

If we have a **univariate** distribution $P(X)$, meaning we have a discrete random variable with domain $\{a_1, \cdots, a_k\}$, then there is an easy algorithm to sample from X. Specifically, divide up $[0, 1]$ into k intervals, such that the ith interval has length $P(X = a_i)$. Then choose a uniform real number r between 0 and 1, and choose a_j such that r lands in the jth region.

(For example, if we have a random variable which takes on the values a, b, c, d with probabilities $0.3, 0.25, 0.27, 0.18$ and our real number r is 0.5209, then our sample is b.) We can do the same process with a **continuous** random variable as well – if X has CDF F_X, then our output is $F_X^{-1}(r)$ (where r is uniform on $[0, 1]$); notice that this is actually the same as the description above for a discrete random variable because the CDF is just a step function.

But sampling from a joint distribution can be more difficult depending on how the variables depend on each other. In a BN, we can do **forward sampling**:

Example 120

A Bayesian network is a generative model in which we can sample the variables one by one in some topological ordering. This way, we can always generate the values of the parents of a node before the values of the node itself, so we can just sample from each conditional probability table using Example 119.

Each sample is then indeed chosen from the distribution P, so the more samples we draw, the more representative of "typical results" our data will look. So the main idea of Monte Carlo estimation is to **express our quantities as expectations**

$$\boxed{\mathbb{E}_{\vec{x}\sim P}[g(\vec{x})] = \sum_{\vec{x}} g(\vec{x})P(\vec{x})}.$$

For example, if we have a joint distribution between variables (A, B, C, D), then we may write the marginal probability for B as

$$P(B = b) = \sum_{a,c,d} P(a, B = b, c, d) = \sum_{a,c,d} \sum_{b} P(a, b, c, d) 1\{B = b\}.$$

Thus, we've written the marginal probability as an expectation

$$\mathbb{E}_{(a,b,c,d)\sim P}[g(a, b, c, d)],$$

where g is the indicator $1\{B = b\}$. And the point is that our $\vec{x}$s are being sampled, so we don't need an explicit formula for $P(\vec{x})$ if we want an estimate of this expectation. Instead, we generate T samples $\vec{x}^{(1)}, \cdots \vec{x}^{(T)}$ from the distribution P, so we can approximate the boxed quantity above as

$$\mathbb{E}_{\vec{x}\sim P}[g(\vec{x})] \approx \hat{g}(\vec{x}^{(1)}, \cdots, \vec{x}^{(T)}) = \frac{1}{T}\sum_{t=1}^{T} g(\vec{x}^{(t)}).$$

Note that $\hat{g}$ is a random variable, so the result that this algorithm will produce will be different every time!

Example 121

If we want to estimate the probability of some event $\vec{E} = \vec{e}$ occurring, we can write that probability as

$$P(\vec{E} = \vec{e}) = \sum_{\vec{x}} \delta_{\vec{e}}(\vec{x})P(\vec{x}) = E_P[\delta_{\vec{e}}(\vec{x})],$$

where $\delta_{\vec{e}}(\vec{x})$ is 1 if $\vec{x}$ is consistent with the evidence and 0 otherwise.

We then estimate this probability with the Monte Carlo estimate

$$\hat{P}(E = e) = \frac{1}{T}\sum_{t=1}^{T} \delta_e(x^{(t)});$$

in words, this is just saying that we take the number of samples with $E = e$ divided by the total number of samples T. Such a computation can be much more efficient, as long as there actually is an efficient sampling algorithm and we're okay with approximate answers (our estimate gets better as the number of samples increases).

Proposition 122

This Monte Carlo estimate is **unbiased**, meaning that

$$\mathbb{E}_{\vec{x}^{(1)},\cdots,\vec{x}^{(T)}\sim P}[\hat{g}(x^{(1)}, \cdots, x^{(T)})] = \mathbb{E}_{\vec{x}\sim P}[g(\vec{x})].$$

Furthermore, by the law of large numbers, we have (almost surely) the **convergence**

$$\hat{g} = \frac{1}{T}\sum_{t=1}^{T} g(\vec{x}^{(t)}) \to \mathbb{E}_P[g(\vec{x})] \text{ as } T \to \infty.$$

In particular, the **variance** of $\hat{g}$ is given by $\text{Var}_P[\hat{g}] = \frac{\text{Var}_P[g(\vec{x})]}{T}$.

In other words, our **average** error is zero – we have the correct expected value. So "in the limit" of infinitely many samples, we do indeed get the right answer for the true marginal probability or other quantity of interest; because our samples are **independent**, we get this $\frac{1}{T}$ behavior for the variance of our estimator.

Refining this variance statement, we can get various **concentration estimates** (understanding the probability that our Monte Carlo estimator is off by at most some error). There are two different probabilistic guarantees we have, one additive and one multiplicative.

Proposition 123 (Hoeffding bound)

Suppose the function g we're trying to estimate only takes on the values $\{0, 1\}$. Then

$$P_{\vec{x}^{(i)}\sim P}\left[\mathbb{E}_P[g(\vec{x})] - \varepsilon \le \hat{g}(\vec{x}^{(1)}, \cdots, \vec{x}^{(T)}) \le \mathbb{E}_P[g(\vec{x})] + \varepsilon\right] \ge 1 - 2e^{-2T\varepsilon^2}.$$

Proposition 124 (Chernoff bound)

Similarly if g only takes on values $\{0,1\}$, then

$$P_{\vec{x}^{(i)}\sim P}\left[\mathbb{E}_P[g(\vec{x})](1-\varepsilon) \le \hat{g}(\vec{x}^{(1)},\cdots,\vec{x}^{(T)}) \le \mathbb{E}_P[g(\vec{x})](1+\varepsilon)\right] \ge 1 - 2e^{-T\frac{\varepsilon^2}{3}\mathbb{E}_P[g(x)]}.$$

In either case, the idea is that for any given error ε, our error probability decreases exponentially fast in T. For example, if $\varepsilon = 0.2$ and $T = 20$, then $1 - 2e^{-2T\varepsilon^2} \approx 0.6$, but with $T = 40$ that quantity grows to 0.92. So we can bound our error with high probability just by taking enough samples, and this means that it's easy to estimate marginals for a BN (by forward sampling).

We often also care about **conditional** queries, though – we may want to know about conditional distributions of the form $p(\vec{Z}|\vec{E} = \vec{e})$. If we have samples from this distribution, we can still use Monte Carlo to estimate the probabilities: for example we can rewrite this conditional probability as an indicator

$$\begin{aligned} P(Z_1 = 0|\vec{E} = \vec{e}) = \sum_{z_2,\cdots,z_k} P(Z_1 = 0, Z_2 = z_2, \cdots, Z_k = z_k|\vec{E} = \vec{e}) &= \sum_{z_1,\cdots,z_k} P(z_1,\cdots,z_k|\vec{E} = \vec{e})1\{Z_1 = 0\} \\ &= \sum_{z_1,\cdots,z_k} P(\vec{Z}|\vec{E} = \vec{e})g(\vec{Z}) \end{aligned}$$

where $g(\vec{Z})$ is the indicator of $Z_1 = 0$. So again we estimate with a sample average **as long as we have access to samples** from the conditional distribution, and the problem now becomes "how to sample."

Definition 125

One simple idea is **rejection sampling**, in which we generate from the original distribution and reject anything not consistent with the evidence.

For example, if we want to sample points uniformly in the unit disk and only have a uniform generator $[-1,1]$, we can generate x, y independently and uniform on $[-1,1]$ and only accept a sample if $x^2 + y^2 \le 1$. So if we have a Bayesian network where we condition on some random variables taking on certain values, we just repeatedly use our topological ordering sampling method and only take the sample if we get the correct values for the conditioned random variables.

This works, but efficiency depends on how frequently we expect to get a sample that is consistent with our evidence – if the probability of the evidence is p, we'll need $\frac{1}{p}$ trials on average to get a **single** sample that is consistent. So conditioning on very unlikely events may not be a good idea – remember that the conditional probability itself can still be quite large even if the event has small probability.

So the other strategy is called **importance sampling**, and it essentially comes down to **fixing** the evidence variables $\vec{E}$ and sampling over the non-evidence variables $\vec{Z}$. What we do is use a different **proposal distribution** Q on $X \setminus E$ that we can efficiently sample from, such that whenever $P(\vec{Z} = \vec{z}, \vec{E} = \vec{e}) > 0$, we have $Q(\vec{Z} = \vec{z}) > 0$. Then

$$\begin{aligned} P(\vec{E} = \vec{e}) &= \sum_{\vec{z}} P(\vec{E} = \vec{e}, \vec{Z} = \vec{z}) \\ &= \sum_{\vec{z}} P(\vec{E} = \vec{e}, \vec{Z} = \vec{z}) \frac{Q(\vec{Z} = z)}{Q(\vec{Z} = z)} \\ &= \mathbb{E}_Q\left[\frac{P(\vec{Z} = \vec{z}, \vec{E} = \vec{e})}{Q(\vec{Z} = \vec{z})}\right] \\ &= \mathbb{E}_Q[w(\vec{z})], \end{aligned}$$

where because the variables in E are fixed, w only depends on the variables in Z, meaning we can now just sample $z^{(1)}, \cdots, z^{(T)}$ and use Monte Carlo to estimate that

$$\hat{P}(\vec{E} = \vec{e}) = \frac{1}{T}\sum_{t=1}^{T} w(\vec{z}^{(t)}).$$

In words, there is a mismatch between samples from P and samples from Q, so we have to **reweight our samples by this factor** w to correct for its effect in the Monte Carlo calculation. And since this is a Monte Carlo estimator, we have all of the same facts as before – the only problem is that our **variance** depends on how good our proposal distribution is.

Example 126

Suppose we have variables (A, B, C, D) and we want to condition on two of the variables $B = b, C = c$. If we let Q be the **uniform proposal distribution** over A and D, then we are generating samples (A, D) uniformly and calculating the sample average of $\frac{P(\text{sample, evidence})}{Q(\text{sample})}$, where this numerator and denominator can be easily computed if we have a BN.

But it's possible that $\frac{P}{Q}$ will be very large or very small for different samples, because even though A and D are uniform, some values will be much more weighted than others for the fixed B and C that we observe. So that would make the variance large – to make the variance small, optimally we would **choose** Q **to be close to** $P(\vec{Z}|\vec{E} = \vec{e})$. If we chose Q to **exactly** be that posterior distribution $P(\vec{Z}|\vec{E} = \vec{e})$, then all importance weights will be constant – in fact the weights will all be

$$w(\vec{z}) = \frac{P(\vec{Z} = \vec{z}, \vec{E} = \vec{e})}{Q(\vec{Z} = \vec{z})} = P(\vec{E} = \vec{e})$$

by the chain rule. Unfortunately, this is just a thought experiment – sampling from a conditional distribution is NP-hard in general, so all it does is tell us what we are aiming for.

Example 127

If we visualize our prior p as being some probability contour, and we think of conditioning on E as restricting to some area, then we should rescale the contour p over the new area. But that's difficult to do, since we would need to know how much total probability mass there is in E.

One strategy we can use here is **likelihood weighting**, in which we do forward sampling like in a normal BN but **clamp the evidence**. In other words, we go in the usual order, and if we already have a value assigned by evidence then we just skip that variable. We then have

$$Q(\vec{x}) = \prod_{v \notin E} P(x_v | \vec{x}_{\text{Pa}(v)});$$

notice that X_v only pays attention to the evidence in its ancestors (**not** its descendants), so this is somewhere in between the prior and posterior distribution. This means that as long as the evidence is relatively towards the top of our BN, this will work pretty well in practice. What's nice is that the importance weight is easy to compute as well: we have

$$w(x) = \frac{\prod_{v \in Z \cup E} P(x_v | \vec{x}_{\text{Pa}(v)})}{\prod_{v \notin E} P(x_v | \vec{x}_{\text{Pa}(v)})} = \prod_{v \in E} P(x_v | \vec{x}_{\text{Pa}(v)}),$$

which can be efficiently computed.

Remark 128. *Notice that if our evidence is only in the "leaves" of a Bayesian network, then likelihood weighting isn't a great idea, since we would be ignoring the evidence altogether. So this weighting performs poorly when our evidence is at the leaves and is also unlikely (we would basically be doing rejection sampling).*

With this, we can go back to our original question and estimate our conditional probability $\mathbb{P}(Z_i = z_i | \vec{E} = \vec{e})$. One option is to separately estimate both terms of the conditional probability using importance sampling and compute

$$\hat{P}(Z_i = z_i | \vec{E} = \vec{e}) = \frac{\hat{P}(Z_i = z_i, \vec{E} = \vec{e})}{\hat{P}(\vec{E} = \vec{e})},$$

but this can go wrong because we're using a ratio of two estimators and the errors can accumulate (for example if the numerator is underestimated while the denominator is overestimated). Instead, we'll **use the same samples** for both the numerator and denominator, and that's called **normalized importance sampling**. Specifically, we can write our conditional probability as summing over all other variables Z_j:

$$P(Z_i = z_i | \vec{E} = \vec{e}) = \frac{\sum_{\vec{z}} \delta_{z_i}(\vec{z}) P(\vec{Z} = \vec{z}, \vec{E} = \vec{e})}{\sum_{\vec{z}} P(\vec{Z} = \vec{z}, \vec{E} = \vec{e})},$$

and now we can use the **same** proposal distribution Q and the **same** samples to estimate

$$\hat{P}(Z_i = z_i | \vec{E} = \vec{e}) = \frac{\frac{1}{T}\sum_{t=1}^{T} \delta_{z_i}(\vec{z}^{(t)}) w(\vec{z}^{(t)})}{\frac{1}{T}\sum_{t=1}^{T} w(\vec{z}^{(t)})} = \boxed{\frac{\sum_{t=1}^{T} \delta_{z_i}(\vec{z}^{(t)}) w(\vec{z}^{(t)})}{\sum_{t=1}^{T} w(\vec{z}^{(t)})}}.$$

And this way, we'll likely overestimate or underestimate the numerator and denominator in the same direction, which will give us a better estimate for the ratio. This actually gives us a **biased** estimator – if $T = 1$, then

$$\mathbb{E}_{\vec{z}^{(1)} \sim Q}[\hat{P}] = E_Q\left[\frac{\delta_{z_i}(\vec{z}^{(1)}) w(\vec{z}^{(1)})}{w(\vec{z}^{(1)})}\right] = \mathbb{E}_Q[\delta_{z_i}(\vec{z}^{(1)})]$$

is the marginal probability on Q – it doesn't even depend on P at all, so it's definitely not the conditional probability $P(Z_i = z_i | \vec{E} = \vec{e})$. But it will be **asymptotically unbiased**, meaning that $\lim_{T \to \infty} \mathbb{E}_Q\left[\hat{P}(X_i = x_i | \vec{E} = \vec{e})\right] = \mathbb{P}(X_i = x_i | \vec{E} = e)$. So we can again trust the result in the limit, and it does behave better than independently performing IS on the numerator and denominator in practice.

Next time, we'll discuss MCMC sampling, which will create a Markov chain that allows us to sample from undirected models as well.

13 February 9, 2024 (Section)

Today's section is on the **junction tree algorithm**, taught by Chaitanya Patel. We'll start with some material from lecture and the syllabus, and then we might say a bit about some additional content as well.

The motivation for caring about this algorithm is that we have an algorithm for calculating marginals on trees – belief propagation – which doesn't work on general loopy graphs (since it will multiply the same potentials multiple times, leading to numerical instability, and it may not even converge). So we need to come up with a new data structure, and that's going to be the **cluster tree** (which is a tree which basically "groups together" nodes of the original graph into clusters). Specifically, we want each factor in our original distribution to have scope contained within some cluster, and we want the **running intersection property**, which ensures that the clusters in which a variable shows up form a connected component – together, these conditions give us a **junction tree**.

There are many ways to find a junction tree, but the idea is that variable elimination always gives us (as a byproduct) a junction tree. Here are the general steps:

1. If we have a Bayesian network, first moralize into an undirected graph.
2. Choose some elimination ordering.
3. Remember that when each variable X is eliminated, we first create a factor with X and all other variables that appear in some factor with X. All of these variables (including X) together form a cluster node.
4. Also, each cluster's factor τ may be used at most once when calculating another intermediate potential ϕ (corresponding to some other cluster) later in the process. (In other words, τ may show up as one of the factors multiplied together to form a later ϕ.) If so, draw an edge between those two clusters.

We can check that this does indeed satisfy running intersection (because once a variable is eliminated, we won't see any more appearance of it in clusters afterward) and that each factor is contained in some cluster (just by the way variable elimination creates its factors). Furthermore, it will be a tree because every factor except the last one is used exactly once.

Fact 129
Notice that different elimination orderings can yield junction trees, and sometimes when we create junction trees we can collapse other clusters into a larger one (because if we have a factor on A, B, C, D, we don't need another factor on A, B if we just incorporate the expression of the latter into the former).

The idea is that once we have a junction tree, we can do variable elimination on it – it's very similar to the process for factor graphs, where messages are sent from a cluster C_i to a cluster C_j if we've already received messages from all other neighboring clusters to C_i. Furthermore, remember that the message is now not just a function of a single variable – it's a function of the sep-set $C_i \cap C_j$. Here's the process:

1. First initialize all potentials based on the potentials from our original MRF. The idea is that our original joint probability distribution is a product of a bunch of local factors, and we choose each local factor to be **arbitrarily** included in **one** of the local ψ_i^0 factors. In this way, we get new ψ "single-cluster" potentials that encode the original potentials. For example, we can think of first initializing all single-cluster potentials to 1, and then if our joint distribution is $P(A)P(B|A)P(C)P(D|A,C)$ from a Bayes net, then $P(B|A)$ can go into any cluster including A and B, $P(C)$ can go into any cluster that includes C, and so on.
2. From here, we send messages of the form $\delta_{i\to j}$ as follows: compute product of our local potential ψ_i^0 with messages from all other neighbors $\mathcal{N}(i)\setminus j$, and then we marginalize over all variables except those in the sep-set $C_i \cap C_j$. Since we have a tree, we will always have a message that is available to send, until all messages in both directions along each edge have been sent.

This process is called the **Shafer-Shenoy clique tree algorithm** – it caches computations so that we can compute **all** marginals at different modes eventually after some pre-computation, and it's exact. Indeed, if we now want to calculate the marginal of some X, we can now pick **any** cluster C_i containing X, do the local computation required to get the joint distribution on C_i (multiplying the potential with all incoming messages), and then marginalize over $C_i \setminus X$.

Fact 130

An alternative approach to directly using VE to find junction trees is to consider **chordal graphs**, which are graphs where all cycles of length 4 or larger have a **chord** between two of its vertices (meaning we have smaller cycles too).

The benefit of not having large cycles is that we won't create huge cliques, and we can create these chordal graphs using triangulation (adding extra edges). In this way, we can form a cluster tree where each cluster has only 3 or fewer vertices. Unfortunately, finding the minimum triangulation is NP-hard, so we often need to observe some structure for this to be useful.

14 February 13, 2024

Today, we'll continue our discussion of sampling methods using the general strategy of **Markov chain Monte Carlo**. The main idea we described last time is to express a quantity $\mathbb{E}_P[g(\vec{x})]$ as the expected value of a random variable and get an estimate from sampled data via

$$\hat{g}(x_{(1)}, \cdots, x_{(T)}) = \frac{1}{T}\sum_{t=1}^{T} g(\vec{x}^{(t)}).$$

The question then becomes "how to get samples from the distribution" especially when we condition on some evidence E. We mentioned that rejection sampling is sometimes a poor strategy if we're conditioning on a very rare event, so the idea is to instead use **importance sampling** by using some other proposal distribution Q on the non-evidence variables and then reweighting using importance weights. The idea is that sampling from posterior distributions is hard in general, but this intermediate Q which can be efficiently sampled can help us get "close" to it.

What we'll do today is some kind of **adaptive proposal** instead of being forced to fix our distribution Q at the beginning – the idea is to adjust the distribution given the quality of our samples. Intuitively, if our true distribution P has two different "peaks" of likely events and our proposal distribution Q only concentrates on the first one, then importance sampling would give us most samples from the first peak and very few from the second one. This means we'd have to weight until we see a sample from the second peak, which may take a very long time. So instead, we'll try to construct some distribution $Q(\vec{x}'|\vec{x})$ which (as a function of $\vec{x}'$) **depends on the previous sample** $\vec{x}$. For example, if Q is a Gaussian, we may choose the mean of the Gaussian to be whatever sample value was drawn last.

Example 131

We'll first see an example of how MCMC is used in practice, and the "why it actually works" will come later. The first class we'll consider is called the **Metropolis-Hastings algorithm**, which is a general-purpose technique for drawing samples from some target distribution.

The idea is that after drawing a sample from $Q(\vec{x}'|\vec{x})$, we have some **acceptance probability**

$$A(\vec{x}'|\vec{x}) = \min\left(1, \frac{P(x')Q(\vec{x}|\vec{x}')}{P(\vec{x})Q(\vec{x}'|\vec{x})}\right)$$

which is meant to represent the quality of our sample. This complicated fraction is the **ratio of the importance weights for $\vec{x}$ and $\vec{x}'$**, so we're saying that if $\vec{x}'$ has a lower importance weight than $\vec{x}$, there is a chance that we reject the sample. And this quantity is tractable to compute because we can calculate the **ratio** of the probabilities $\frac{P(\vec{x}')}{P(\vec{x})}$

and not the actual probabilities explicitly, so we don't need full knowledge of the distribution for this to be doable – for example, this means we do not need knowledge of the partition function Z in a Markov random field.

Remark 132. *We'll learn about this later in the lecture, but it turns out that this "accept/reject" step is what ensures our samples will actually be coming from $P(\vec{x})$ in the limit. (We have to compare our Q to P in some way so that we go from an arbitrary distribution to the correct one.)*

So the algorithm works as follows: choose some starting state $\vec{x}^{(0)}$ at time $t = 0$. Now repeat the following process (which we call the **burn-in phase**): sample from our distribution $x^* \sim Q(\vec{x}^*|\vec{x})$, sample a uniform variable $u \sim \text{Unif}(0, 1)$. Now either transition if $u < A(\vec{x}^*|\vec{x})$, meaning that $\vec{x}^{(t+1)} = \vec{x}^*$, or don't, meaning that $\vec{x}^{(t+1)} = \vec{x}^{(t)}$. Once the burn-in phase is complete (detecting convergence), we continue to repeat this process (including the acceptance/rejection) but actually note down for Monte Carlo averages. (So we don't worry about the first few samples which could be very far off – we wait until the convergence to good samples has already happened.)

Fact 133

Notice that the samples are highly correlated now, and whenever we reject a new sample we are repeating the previous value again. But the point is that the total dataset **will** be from something close to the correct distribution.

Example 134

Continuing on our discussion from before, consider the following situation with one-dimensional distributions. Then suppose $Q(x'|x)$ is a one-dimensional Gaussian centered at x with some fixed variance, and $P(x)$ is a bimodal distribution.

Notice that $Q(x'|x) = Q(x|x')$, so in the expression for the acceptance probability we simplify to just

$$A(x'|x) = \min\left(1, \frac{P(x')}{P(x)}\right).$$

(This is the advantage of a **symmetric** proposal.) So we can initialize $x^{(0)}$ to be any value, and now repeatedly perform the following process. Draw a sample $x^{(t+1)}$ which is $x^{(t)}$ plus some Gaussian noise, and now compare how likely this new sample is to the previous one – accept the new sample if it's an more likely than before (an "uphill move"), otherwise accept it with some smaller probability (where the more downhill we move, the less likely for the transition to happen).

If we start with x^0 inside one of the two peaks of the target distribution P, we'll have a decent chance of rejecting transitions out of the peak, meaning that the algorithm will stay in the first peak for a while. But eventually, our position $x^{(t)}$ may exit the peak by random chance and travel to the other peak, which means it will now sample values within that other peak for a while. What's nice is that even if Q is not a good fit for P (here P is not Gaussian at all), this is still enabling us to sample from both modes!

Example 135

When we work on a graphical model, there is a **generic powerful method** for sampling the random variables one at a time called **Gibbs sampling**.

Gibbs sampling is a special case of Metropolis-Hastings where we can actually make use of the graph structure to choose good proposal distributions; what's nice is that they are often easy to derive and only involve local changes.

The algorithm works as follows: if we have an MRF or Bayesian network with non-evidence variables $(X_1, \cdots, X_n)$, we first initialize (typically random) starting values for all of them. Then **for each i in order**, we sample the conditional distribution

$$x \sim P(x_i | x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n),$$

and now there is no rejection of transitions – we just update x_i to x. Then this new value of x_i is used for the sampling of other variables x_j. Luckily, this conditional distribution only depends on the Markov blanket of X_i (neighbors for an MRF, parents, children, and co-parents for a BN), so it's actually tractable to sample this conditional distribution.

The picture to have in mind is that we visit each variable in turn and (while keeping everything else fixed) resample from its conditional distribution. For example, recall the distribution given from the hair color example, Example 42. Then we could imagine that when we resample from A, we choose a random color among those that are not B and D's colors, while keeping all other hair colors fixed. Then we do the same for B among colors not in their neighbors, and so on. Once we've resampled all of A, B, C, D, we have a **new sample**. Repeating this repeatedly will eventually end the burn-in phase, and then we can use the subsequent samples for Monte Carlo.

Fact 136

There is flexibility in choosing ordering – we can do it in a fixed or random order, and in practice that may change the burn-in time. The point is that we are updating variables to be consistent with the joint ones, so we want to choose the ordering so that information "propagates quickly." For example, the topological order might make sense for BNs, but it's pretty hard to prove how long the burn-in phase is in general.

Notice that this process is kind of a "local search" in the case where our MRF is encoding a satisfiability problem, and in those cases the convergence is in fact very slow – it's a little better than a random walk because we're doing a "coordinated ascent" towards the correct answer, but we may still run into problems.

Proposition 137

Gibbs sampling is a special case of Metropolis-Hastings with proposal distribution

$$Q(x_i', \vec{x}_{-i} | x_i, \vec{x}_{-i}) = P(x_i' | \vec{x}_{-i})$$

(where $\vec{x}_{-i}$ denotes all variables except i). In particular, this probability is zero unless all variables except x_i are kept the same.

We can check that the acceptance probability ends up always being 1 with this proposal (the factors in the complicated fraction all cancel out after using the chain rule). And what's actually going on is that we are sampling from a Markov chain:

Definition 138

A **Markov chain** is a sequence of random variables $X_1, X_2, X_3, \cdots$ which satisfy the **Markov property**

$$P(X_t = \vec{x} | \vec{x}_1, \cdots, \vec{x}_{t-1}) = P(X_t = \vec{x} | \vec{x}_{t-1}).$$

In other words, what happens at any given step only depends on what happened at the previous step, similar to the hidden Markov models we've described before.

We call $P(X_t = \vec{x} | \vec{x}_{t-1})$ the **transition kernel**, and we'll be considering the case where this transition kernel is fixed with time (homogeneous Markov chain) and is therefore of the form $T(\vec{x}' | \vec{x})$ for some **fixed** T. The idea is that **the states of our Markov chain are the possible instantiations of our random variables in the graphical model** – for example if we have an MRF with two binary variables X_1, X_2, then Gibbs sampling is a Markov chain on $(0,0), (0,1), (1,0), (1,1)$, since if we're at $(0,1)$ at one step, the probability of ending up at $(1,1)$ after we've "rerolled all variables" doesn't depend on what the variables looked like at any other previous steps, and in fact the probability of transition is the same (we have a homogeneous chain) since the functional form of Q doesn't change. This Markov structure is what makes it a good idea to only have $Q(\vec{x}'|\vec{x})$ depend on the immediately previous sample, but it also points out why convergence may take a long time – there are exponentially many states and getting to the correct distribution over all of them can take a long time.

In probability language, we can define $\pi^{(t)}(\vec{x})$ to be the distribution over states $\vec{x}$ (possible variable instantiations) at time t. The Markov transition kernel then tells us that

$$\pi^{(t+1)}(\vec{x}') = \sum_x \pi^{(t)}(\vec{x})T(\vec{x}'|\vec{x})$$

(casework over all possible values at time t, weighted by the transition probabilities), so over time we will use the transition kernel T to spread out the mass until it converges to the **stationary distribution** of the Markov chain, which is some distribution π satisfying

$$\pi(\vec{x}') = \sum_{\vec{x}} \pi(\vec{x})T(\vec{x}'|\vec{x}).$$

The point is that π for this chain is actually going to be the distribution $P(\vec{x})$.

Definition 139

A Markov chain is **irreducible** if we can get from any state $\vec{x}$ to any other state $\vec{x}'$ in a finite number of steps, and it is **aperiodic** if for all sufficiently large n, we have $P(x^{(n)} = i|\vec{x}^{(0)} = i) = 0$ for all i (that is, we can eventually always return to our starting point – this avoids cycles or other oscillatory behavior). We call a Markov chain **ergodic** if both of these conditions are satisfied.

(An example of a reducible chain is where we have two states but one of them is a sink which stays at its current position with proabbility 1, and an example of an aperiodic chain is where we have two states and we always transition to the other one with probability 1 – both of these examples are bad because the limiting distribution depends on our initial condition in the first case and doesn't even exist in the second.)

We want these conditions because **ergodic chains always converge to a unique stationary distribution**, so if our MCMC algorithm is ergodic and our joint distribution P is the stationary distribution, we will eventually converge correctly. And now we can see the reason for choosing T as in Metropolis-Hastings:

Proposition 140

For the Metropolis-Hastings the **detailed balance** or **reversibility** equations

$$\pi(\vec{x}')T(\vec{x}|\vec{x}') = \pi(\vec{x})T(\vec{x}'|\vec{x})$$

(meaning that the probability of seeing $\vec{x}'$, then $\vec{x}$ is the same as the probability of seeing $\vec{x}$, then $\vec{x}'$).

Reversibility **automatically implies** that we have a stationary distribution π (if we sum over $\vec{x}$ on both sides, we end up with $\pi(\vec{x}') = \sum_x \pi(\vec{x})T(\vec{x}'|\vec{x})$), so if π is exactly our joint distribution P, this would show that assuming ergodicity assumptions we will indeed converge to the correct P.

Proof. The transition kernel for Metropolis-Hastings is given by

$$T(\vec{x}'|\vec{x}) = Q(\vec{x}'|\vec{x})A(\vec{x}'|\vec{x}),$$

and recall that $A(\vec{x}'|\vec{x}) < 1$ (resp. $= 1$) implies $A(\vec{x}|\vec{x}') = 1$ (resp < 1). In words, if the transition in one direction is uphill, then the transition in the other direction is downhill. So without loss of generality we can assume $A(\vec{x}|\vec{x}') = 1$ and plug in all of our definitions to check that we indeed have

$$P(\vec{x})T(\vec{x}'|\vec{x}) = P(\vec{x}')T(\vec{x}|\vec{x}'),$$

meaning P is indeed the stationary distribution that makes T satisfy detailed balance. □

So the big picture is that the burn-in phase is converting our distribution from $\pi^{(0)}$ to $\pi^{(T)}$, where $\pi^{(T)}$ has gotten sufficiently close to the true $\pi = P$ stationary distribution. And once that's done, we can do (approximate) Markov chain Monte Carlo to estimate sample averages. Unfortunately, the **mixing time** (how long it actually takes to get there) can be very long and also difficult to generally compute. So in practice, knowing when to halt burn-in is an art, and we have heuristics but **no general method** for detecting convergence.

15 February 15, 2024

Today's our last lecture on inference – we'll discuss MPE and MAP inference today, where instead of computing marginal probabilities or sampling from some posterior or marginal distribution, we think about how to find the **most likely** assignment of variables. The inference task looks formally like

$$\text{argmax}_{\vec{x}} p(\vec{x}), \quad p(\vec{x}) = \frac{1}{Z}\prod_{c \in C} \phi_c(\vec{x}_c),$$

where we assume that the evidence variables have already been clamped so we're just looking at the non-evidence ones. What's nice is that the normalization constant is an overall constant, so we can ignore the Z here and just solve the **max-product** inference task

$$\text{argmax}_{\vec{x}} \prod_{c \in C} \phi_c(\vec{x}_c).$$

We can also apply log to everything, since log is a monotonic function, and thus we may instead be trying to solve the **max-sum** problem

$$\text{argmax}_{\vec{x}} \sum_{c \in C} \theta_c(\vec{x}_c), \quad \theta_c(\vec{x}_c) = \log \phi_c(\vec{x}_c).$$

One way to approach this problem is to use the Metropolis-Hastings algorithm from last lecture – we can use it to sample from any distribution, and in particular we can do so from our model $p(\vec{x}) \propto \exp\left(\sum_{c \in C} \theta_c(\vec{x}_c)\right)$. But to bias the model towards sampling likely events, we can instead sample from the rescaled function

$$p_T(\vec{x}) \propto \exp\left(\frac{1}{T}\sum_{c \in C} \theta_c(\vec{x}_c)\right),$$

where T is a parameter that we call **temperature**. In the high-temperature regime where $T \to \infty$, the argument of the exponential goes to 0 regardless of the factors that we began with, so we end up sampling from a distribution close to the **uniform** distribution. But as $T \to 0$, then the argument of the exponential is much larger when $\sum_{c \in C} \theta_c(\vec{x}_c)$ is larger, so the probability mass will **concentrate on assignments with highest probability** (even more than the default $T = 1$).

In general, this kind of problem is provably NP-hard in the worst case, and Metropolis-Hastings can take a long time to converge. But the idea is that if we slowly decrease the temperature T, we can transition from an easier optimization problem to a harder one. This is the idea of **simulated annealing**: start running the algorithm from some initial guess with high T, and **anneal** (slowly decreasing T) as we continue. And in particular, the effect of the temperature is to affect the acceptance probability – the algorithm becomes "greedier" as we decrease the temperature, because

$$A(\vec{x}'|\vec{x}) = \min\left(1, \frac{p_T(\vec{x}')Q(\vec{x}|\vec{x}')}{p_T(\vec{x})Q(\vec{x}'|\vec{x})}\right)$$

and if our proposal distribution Q is symmetric (for example if we randomly pick a variable and pick a new value for it), substituting in the form of our distribution p_T yields

$$\min\left(1, \exp\left(\frac{1}{T}\sum_{c\in C}\theta_c(\vec{x}_c') - \theta_c(\vec{x}_c)\right)\right).$$

In particular, for $T \to \infty$ we will basically always accept a new state $\vec{x}'$, but for $T \to 0$ it will basically only accept uphill moves.

Fact 141

It turns out that for sufficiently slow cooling, this algorithm is guaranteed to find the mode (the most likely outcome). However, it's impractical in general to practically meet this "sufficient slowness" for the theoretical guarantee.

This procedure is very general and can in fact be used for any function – we don't ever use the fact that our distribution is defined by a set of factors. But that means we aren't taking advantage of the structure of conditional independencies, and next we'll see strategies where that structure can actually be exploited. This is often the case if we have a structure prediction problem, such as when we have a conditional random field for the handwriting recognition problem in Example 71.

The key idea is that just like in variable elimination, there is some dynamic programming we can do. The function max distributes over multiplication, just like summation did when we were "pushing in sums:"

$$\text{sum}(ab, ac) = a\text{sum}(b, c), \quad \max(ab, ac) = a\max(b, c)$$

as long as $a \geq 0$ (which is the setting we care about because all entries of factors are nonngative). This distributivity works if we have more than 2 terms as well, and it works for variables or constants. Furthermore, max is also commutative and associative (just like addition), so we can think of max and multiplication and forming a **commutative semi-ring**, and we can translate all of the ideas from variable elimination over directly to this new setting: we can "push max in" or "factor out terms from inner optimizations" to get

$$\max_{x_1\in\{0,1\}}\max_{x_2\in\{0,1\}} f(x_1)g(x_1,x_2) = \max_{x_1\in\{0,1\}} f(x_1) \max_{x_2\in\{0,1\}} g(x_1,x_2).$$

This way, we can again cache intermediate computations (as long as all of our functions are nonnegative), and instead of trying to solve the sum-product problem $\sum_{\vec{x}}\prod_{c\in C}\phi_c(\vec{x}_c)$, we instead solve the max-product problem $\max_{\vec{x}}\prod_{c\in C}\phi_c(\vec{x}_c)$ with the exact same algorithm, just replacing $+$ with max everywhere! So we do have max-product variable elimination, max-product belief propagation, and max-product junction tree – everything comes down to picking an ordering and doing dynamic programming.

Fact 142

The only additional subtlety is that if we care about **argmax**, not **max**, we should also keep track of which assignment is achieving this maximum via back pointers.

Example 143

Suppose we have a simple chain with nodes A, B, C, D in a row, and we want to find the most probable (MPE) assignment

$$\max_{a,b,c,d} \phi_{AB}(a,b)\phi_{BC}(b,c)\phi_{CD}(c,d).$$

Pushing the maximizations in yields

$$\max_{a,b} \phi_{AB}(a,b) \max_{c} \phi_{BC}(b,c) \max_{d} \phi_{CD}(c,d),$$

and now we can do the inner operation first, replacing it with a dynamic programming table or intermediate factor, and repeating this until we have completed the whole task. (And like before, doing things in log space is generally more stable numerically.)

We'll next discuss some **other** optimization techniques coming from combinatorial or convex optimization to deal with this kind of setting; it turns out there are some connections between standard techniques and message-passing or loopy BP.

Definition 144

A **linear program (LP)** is an optimization problem of the form

$$\min \vec{c}\vec{x} \text{ subject to } A\vec{x} \le \vec{b}.$$

For example, we may have a problem like

$$\min(2x_1 + x_2) \text{ such that } x_1 + x_2 \le 6, x_1 - x_2 \le 4, x_1 \ge 0, x_2 \ge 0.$$

Such problems have a linear objective function with linear constraints, and these can always be solved in polynomial time. And if we have linear **equality** constraints, those are okay as well, since we can encode them via $A\vec{x} \le b$ and $A\vec{x} \ge b$.

Definition 145

An **integer linear program (ILP)** is an optimization problem of the same form as a linear program, but we further require integrality constraints $\vec{x} \in \{0,1\}^N$.

Integrality constraints give us something combinatorial, and thus they are generally NP-hard to solve. But there are approximation methods for them, and our goal is going to be **writing MPE as an integer linear program**. For simplicity, let's consider the case with only unary and pairwise potentials. As a discrete optimization problem, our goal is then to compute

$$\text{argmax}_{\vec{x}} \sum_{i \in V} \theta_i(x_i) + \sum_{ij \in E} \theta_{ij}(x_i, x_j).$$

To convert this to an ILP, introduce **indicator variables** $\mu_i(x_i) \in \{0,1\}$ for **each** i and each x_i (For example, if the first variable can take on three states, we have three binary indicators that encode which value it's taking on.) We also introduce indicators $\mu_i(x_i, x_j)$ for each pair of states x_i, x_j for each edge $(i, j) \in E$. We then actually have a linear objective function over the xs given by

$$\text{MPE}(\theta) = \max_{\mu_i,\mu_{ij}} \sum_{i\in V}\sum_{x_i}\theta_i(x_i)\mu_i(x_i) + \sum_{(i,j)\in E}\sum_{x_i,x_j}\theta_{ij}(x_i, x_j)\mu_{ij}(x_i, x_j),$$

which we can write as $\max_{\vec{\mu}} \vec{\theta} \cdot \vec{\mu}$ if we stack all of our variables into single vectors.

Example 146

Suppose we have an MRF with three binary variables all pairwise connected. Then we need 2 variables μ_i for each assignment of the X_is, and we also need 4 variables μ_{ij} for each pairwise assignment (X_i, X_j), so in total we will need 18 indicator variables (not particularly efficient, but still written in a form which fits is for linear optimization).

Of course, there are a bunch of constraints that these variables must satisfy. We can't have $\mu(x_1 = 1) = \mu(x_1 = 0) = 1$ (a variable can only take on one value at a time), and we also can't have $\mu(x_1 = 1) = 1$ and $\mu(x_1 = 0, x_3 = 0) = 1$ (we need consistency between our variable and edge assignments). But we're in luck that we can write these as linear constraints:

$$\mu_i(x_i) \in \{0, 1\} \quad \forall i \in V, x_i, \quad \sum_{x_i}\mu_i(x_i) = 1 \quad \forall i \in V,$$

and similarly

$$\mu_{ij}(x_i, x_j) \in \{0, 1\} \quad \forall (i, j) \in E, x_i, x_j, \quad \sum_{x_i,x_j}\mu_{ij}(x_i, x_j) = 1 \quad \forall (i, j) \in E,$$

and finally we want consistent local assignments

$$\sum_{x_j}\mu_{ij}(x_i, x_j) = \mu_i(x_i) \quad \forall (i, j) \in E, x_i, \quad \sum_{x_i}\mu_{ij}(x_i, x_j) = \mu_j(x_j) \quad \forall (i, j) \in E, x_j.$$

(This behaves a lot like a marginal probaiblity if we think of μ_{ij} as a marginal probability over x_i, x_j.) So all constraints are either linear or enforce integrality, meaning we've indeed written down our inference problem as an ILP. And then there is lots of software we can use (like CPLEX and Gurobi) to try to solve this.

Remark 147. *Geometrically, we have a big space of possible distributions that we're optimizing over – it's a polytope that we can call the* ***marginal polytope****. The corners of this polytope is the set of $\vec{\mu}$ assignments (there may be exponentially many of them), and then* **convex combinations** *of them correspond to the set of* **valid marginal probabilities***. This will show up again when we talk about variational inference later in the course.*

The fact that we require hard choices $\mu_i(x_i), \mu_{ij}(x_i, x_j) \in \{0, 1\}$ in this problem is what is making the problem NP-hard; it would be nice if we can remove those combinatorial constraints. So that's the basic idea of **relaxing integrality constraints** – we can instead ask only for

$$\mu_i(x_i), \mu_{ij}(x_i, x_j) \in \{0, 1\},$$

meaning that we're choosing a probability distribution over values instead of a "hard value." Then we end up with a true LP whose objective function we write as $\text{LP}(\theta)$, which can be solved efficiently (as long as the number of variables is small – it's basically the number of entries in our tables for graphical models). Furthermore, since we are maximizing over a strict superset of our original set, we have

$$\text{MPE}(\theta) \leq \text{LP}(\theta),$$

and in fact **we have equality for tree-structured MRFs**, meaning that LP relaxation is actually **tight**.

Fact 148

We have some polytope M that defines the combinatorial optimization; by relaxing our constraints so that the μs are within $[0, 1]$ we get the **local consistency polytope** M_L, which is an outer bound for M. And what we're saying is that $M = M_L$ for trees (since the local consistencies imply global consistencies).

Remark 149. *Once we have our "fractional assignment" for each μ_i, μ_{ij}, we could choose the μs by "rounding to the nearest integer solution," though we have to be more careful with this to ensure the constraints are still satisfied. In practice this is pretty good (this is a good heuristic and is pretty similar to what we get if we do loopy max propagation), but in general it isn't guaranteed to give the best solution.*

There are other classes of strategies for solving MAP – **local search** similar to Metropolis-Hastings, or **branch-and-bound** where we do an exhaustic search and prune branches of the search space to find bounds (using LP relaxation). But we won't talk much more about those in this class.

16 February 16, 2024 (Section)

Today's section, taught by Sofian Zalouk, will cover MCMC sampling methods (in particular Metropolis-Hastings and Gibbs sampling). The overview is to go over what each "MC" means (Monte Carlo and Markov Chain), why the algorithms work, and some practical aspects that we might need to know about for diagnosis in practice.

The idea of **Monte Carlo** estimation is that we are trying to understand some quantity of interest, and by writing it as the expectation of some random variable we can estimate our quantity via empirical means. For example, we may want to model uncertainty in a model over parameters θ, and we can do this by writing our expression as integrating over θ and modeling that by sampling. What's nice is that linearity means our estimator is unbiased and independence implies that our "error bars" shrink as $\frac{1}{\sqrt{T}}$.

Example 150

If we want to use Monte Carlo to sample π, we can sample (x, y) uniformly from the unit square and approximate

$$\pi \approx \frac{4}{T}\sum_{i=1}^{T} 1\left\{(x^{(i)})^2 + (y^{(i)})^2 < 1\right\},$$

since the probability of each point landing within $x^2 + y^2 \le 1$ is $\frac{\pi}{4}$. However, we will often notice in practice that the convergence rate isn't so good, and often estimating analytically is a better option if it's at all tractable.

This is connected to the idea of **rejection sampling**, in which we estimate the probability of some event E by generating samples from a Bayesian network and estimating the expectation of the function $f(\vec{x}) = 1\{E = e\}$. (So we accept samples that pass in the indicator, and divide by the total number of samples.) Unfortunately, this is not tractable if the probability of E is very small, meaning that we need to sample a very large number of samples to get any acceptance at all.

Instead, we make use of a **proposal distribution** over non-evidence variables, such that Q can be efficiently sampled from and has support matching our original distribution P (meaning that if $P(Z = z, E = e) > 0$, then $Q(Z = z) > 0$). Then we can do some algebraic manipulation to rewrite $P(E = e)$ as an expectation in terms of Q instead of P; this is called **importance sampling** because we're essentially weighting our samples by how "important" they are.

The main limitations with these sampling methods are that forward sampling is bad with small-probability events and is infeasible for MRFs where we don't know the normalization constant Z, and for importance sampling it can be very difficult to construct a good Q similar to P (because if we knew the analytic form of P we wouldn't need Monte Carlo in the first place).

Thus, we want to make our Q adaptive, and we do this by making use of the concept of a **Markov chain**, in which "the state today only depends on yesterday." Specifically, we'll consider a Markov chain on the **set of states of all variables**, and our goal will be that the distribution of our Markov chain at time t converges to the stationary distribution given by the true joint distribution P as $t \to \infty$.

The main problem is that we may have multiple different limiting distributions, so we want to make sure the dynamics we set up on our chain yield an **ergodic** (also called **regular**) Markov chain, which means we can always eventually return to any state for large enough times and that we can reach any state from any other state in a finite number of steps.

The idea with the Metropolis-Hastings and Gibbs sampling algorithms is to set up a Markov chain whose stationary distribution is indeed P. Specifically, one way to do so is to make the chain **reversible** with respect to P, meaning that for any two states x, x', we have the detailed balance equation

$$P(x')T(x|x') = P(x)T(x'|x),$$

where T is the transition kernel. So our T will actually be dictated by our adaptive proposal $Q(x'|x)$, and specifically we'll derive what we want from first principles. We'll specifically add in an acceptance probability $A(x'|x)$, so that

$$T(x'|x) = Q(x'|x)A(x'|x),$$

choosing A so that we have an ergodic, reversible transition kernel. (Ergodicity would ensure that long-enough simulation guarantees convergence to stationary, and reversibility is the easiest way to ensure the stationary distribution actually is P.) If we plug this formula for T into the detailed balance equation, we see that

$$\frac{A(x'|x)}{A(x|x')} = \frac{P(x')Q(x|x')}{P(x)Q(x'|x)},$$

so the **Metropolis choice** is to choose

$$A(x'|x) = \min\left(1, \frac{P(x')Q(x|x')}{P(x)Q(x'|x)}\right),$$

which always works because either $A(x'|x)$ or $A(x|x')$ will be 1 and the other will be smaller and thus their ratio is indeed correct. And to ensure ergodicity, we just make sure $Q(x'|x)$ is never zero (for example, it might be Gaussian).

So in practice, the transition operator does the following: we propose a move from x to x', accept it with some probability $A(x'|x)$, and either go to that state or stay where we are. What's nice is that our expression for acceptance probability only requires the use of unnormalized factor terms in an MRF (since we're dividing two terms of Ps), so it can often actually be computed in practice. And if we choose Q to be symmetric in x and x', there are no terms involving Q in the acceptance probability at all.

Fact 151

The main complication is that Metropolis-Hastings requires a **burn-in** period, since the **mixing time** to reach something close to the stationary distribution can be very long. This means that we throw away the initial samples during this burn-in phase and only start counting at some later point.

Gibbs sampling is another MCMC method similar to Metropolis-Hastings (in fact it's a special case), but it's not written out in terms of an acceptance probability. Instead, we just pick one variable at a time (in turn or randomly) and resample it given the conditional values of all other variables. What's nice is that we only need to sample one random variable at a time, so for graphical models like a BN it is often very efficient to resample from "local information." In words, we can think of this as saying that we pretend x_i was never sampled, and given the other initializations we do it again – this will keep us at the true "equilibrium" stationary distribution $P(x)$ if we were already there. And we can check that if we use $P(x_i|\vec{x}_{-i})$ as our proposal distribution $Q(x'|x)$, plugging this into Metropolis-Hastings indeed makes $A(x'|x)$ always 1.

Fact 152

In practice, we can tell if $Q(x'|x)$ is actually good by looking at the acceptance rate and studying the autocorrelation function, and we can understand how long to make the burn-in period by visualizing our samples and log-likelihood over time.

As in similar problems, there is an "exploitation versus exploration" tradoeff in picking Q. If we choose it to have low variance, we'll sample things close to our current x, which results in a higher acceptance rate but slower exploration. On the other hand, a high variance proposal may have low acceptance rate (because we are likely to sample proposals on the tails with low values of P) but a high capability for exploring P. So our goal is to propose distant samples which still have a decent chance of being accepted. The concept here is that all MCMC chains show **autocorrelation** (meaning that adjacent samples in time are correlated), and since we want to simulate sampling from P our goal should be to choose something with **low** autocorrelation.

And to keep track of burn-in, we can look at whether samples from different MCMC runs look like they are well-mixed. Specifically, we can imagine overlaying the values of x across different runs on the same graph – if they look like they are still taking different paths, we may have a misspecified proposal and aren't converging to the stationary distribution yet, meaning we should continue burn-in. But as mentioned in lecture, it really is still a bit of an art.

Example 153

We'll close with a few words on more advanced techniques which are particularly well-suited for high-dimensional distributions, **Hamiltonian Monte Carlo** and **annealed importance sampling**.

The basic idea of Hamiltonian Monte Carlo is that we can think of our probability distribution as being dictated by a potential energy function

$$P(x) \propto e^{-E(x)}, \quad E(x) = -\log P^*(x).$$

Motivated by physics, we can then think about a velocity v which carries kinetic energy $K(v) = \frac{1}{2}\langle v, v\rangle$, and we imagine having a ball rolling along with velocity. We can then define a joint distribution

$$P(x, v) \propto e^{-E(x)}e^{-K(v)} = e^{-H(x,v)},$$

where $H(x, v)$ is the "sum of the potential and kinetic energy" and velocity is independent of position. What HMC does is **Gibbs sample the velocity** to yield the next direction of movement, simulating the resulting dynamics in a way which is deterministic and reversible. (The idea is that the velocity tells us how to "leapfrog" and accept new positions based on how we travel, and we can "reverse the velocity" and resample if things are not going well in a certain direction.) However, Hamiltonian dynamics can be hard to model, so we usually do a discretization – the details can be found if we search up the idea of **hybrid Monte Carlo**.

Finally, annealed importance sampling is motivated by the following idea: if we're in a boat on a body of water and want to model the volume, perhaps the only information we have is to sample directly below our position at any time. We could try to model our lakebed as being **initially flat**, and over time as we progress we may incorporate the sharper bumps and peaks and some more extreme events as we go on. So in annealing in general, we consider a probability distribution dependent on the **inverse temperature** β given by

$$P(x; \beta) \propto P^*(x)^\beta \pi(x)^{1-\beta)}.$$

As the temperature decreases (that is, as β increases), this will allow us to incorporate sharper details and have features "peak out" more noticeably. What's nice here is that if we start with a low β, mixing is easier, and we get this "chain" between the posterior and prior

$$P(\theta; \beta) = \frac{1}{Z(\beta)} P(\mathcal{D}|\theta)^\beta P(\theta).$$

So the high-level idea is that these are powerful for **high-dimensional** distributions; annealed importance sampling turns a known simple distribution P^T gradually into a target one P^*, and Hamiltonian Monte Carlo lets us travel past local modes by making use of velocity. So these turn out to have connections to **score-based diffusion models** – they can both be used as samplers, and if we're interested, we can take a look at the paper found here.

17 February 20, 2024

Today begins the **learning** portion of the course, in which we develop tools for acquiring the model from data. We saw previously that learning the parameters of a Bayesian network is rather simple, but unfortunately undirected models will give us a harder time computationally (we will see that it requires some challenging inference problems).

Recall that the setup is as follows: our domain is governed by some underlying distribution p^*, we have a dataset of m samples $\{\vec{x}^{(1)}, \cdots, \vec{x}^{(m)}\}$ from p^* which we'll assume to be iid, and we're given some family of candidate models $\mathcal{M}$. Our goal is then to pick a "good" model $\hat{\mathcal{M}}$ within this family that fits p^* well. (For example, $\mathcal{M}$ may be the set of all Bayes nets with some given (fixed) graph structure.) Notably, we can just learn the model parameters for some fixed structure, or we can also try to learn the structure (which we'll talk about in a later lecture).

A first step to solving a learning problem is some kind of loss function which encodes how well we're fitting the data – in a previous lecture, we showed that minimizing KL divergence is equivalent to maximizing the expected log-likelihood $\mathbb{E}_{\vec{x}\sim p^*}[\log \hat{p}(\vec{x})]$. (In information theoretic terms, "achieving perfect compression is like learning the model exactly.") So if we're trying to do learning in this setting, we can do the same thing but **approximate the expected log-likelihood with the empirical one from our data**

$$\mathbb{E}_{\vec{x}\sim p^*}[\log \hat{p}(\vec{x})] \approx \mathbb{E}_{\mathcal{D}}[\log \hat{p}(\vec{x})] = \frac{1}{|\mathcal{D}|} \sum_{\vec{x}\in\mathcal{D}} \log \hat{p}(\vec{x}).$$

If we try to maximize this right-hand side over all $\hat{p}$, we are then doing what we call **maximum likelihood learning** – it's equivalent to saying that we want to maximize the likelihood of our data $\prod_{\vec{x}\in\mathcal{D}} \hat{p}(\vec{x})$, where notably the $\frac{1}{|\mathcal{D}|}$ scaling doesn't change our solution.

In the case of Bayesian networks, we just needed to figure out how to assign values in our conditional probability tables θ. Since the joint distribution is defined as a product of these conditional probabilities, we were able to rewrite the likelihood into an expression that decomposes across variables (see Example 80 for the details of our calculations), meaning that we can estimate CPDs independently because our objective decomposes.

Example 154

Today's task will be to do the same problem in the same setting, except that we want to start now with an MRF model of the form $p(\vec{x}) = \frac{1}{Z}\prod_{c\in C}\phi_c(\vec{x}_c)$, where our factors are now nonnegative functions which only have a probabilistic interpretation because of the normalization constant Z.

It will be convenient to think of $\phi_c(\vec{x}_c)$ in the "log characterization" as $\exp(\theta_c(\vec{x}_c))$, just to make our expression simpler. This means in particular that we have

$$p(\vec{x}) = \exp\left(\sum_{c\in C}\theta_c(\vec{x}_c) - \ln Z\right),$$

where Z can be expressed in the log parameterization as

$$Z = \sum_{\vec{x}}\exp\left(\sum_{c\in C}\theta_c(\vec{x}_c)\right).$$

We know that computing this Z is expensive in general (it's the sum-product inference task we've previously talked about). Our question today is to see if we can **learn the parameters** θ_c if we fix the graph structure (that is, choose our fixed set of cliques C). Formally, imagine defining our family of distributions as parameterized by $\vec{\theta}$ (the set of all entries of our factors in log space)

$$p(\vec{x};\vec{\theta}) = \frac{1}{Z(\vec{\theta})}\prod_{c\in C}\phi_c(\vec{x}_c) = \exp\left(\sum_{c\in C}\theta_c(\vec{x}_c) - \ln Z(\vec{\theta})\right),$$

where the tricky part is now that the partition function Z **also** depends on θ. We can rewrite this (similar to how we casted inference as linear program) as

$$p(\vec{x};\vec{\theta}) = \exp\left(\sum_{c\in C}\sum_{\overline{\vec{x}}_c}\theta_c(\overline{\vec{x}}_c)1\{\vec{x}_c = \overline{\vec{x}_c} - \ln Z(\vec{\theta})\right),$$

which we can write as $\exp\left(\vec{\theta}^T\vec{f}(\vec{x}) - \ln Z(\vec{\theta}\right)$ so that most of the expression is linear in our parameters.

Example 155

Suppose we have a joint distribution over two random variables given by

$$p(X,Y) = \frac{1}{Z}b(X)g(X,Y)h(Y) = \exp(\log b(X) + \log g(X,Y) + \log h(Y) - \ln Z),$$

where X takes values in $\{\text{red, green, blue}\}$ and Y takes values in $\{0,1\}$.

We can then rewrite the expression for h as

$$\begin{aligned} h(Y) = h(0)1\{Y=0\} + h(1)1\{Y=1\} \implies \log h(Y) &= \log h(0)1\{Y=0\} + \log h(1)1\{Y=1\} \\ &= \theta_h(0)1\{Y=0\} + \theta_h(1)1\{Y=1\} \\ &= \begin{bmatrix}\theta_h(0) & \theta_h(1)\end{bmatrix}\begin{bmatrix}1\{Y=0\} \\ 1\{Y=1\}\end{bmatrix}, \end{aligned}$$

where the parameters we are trying to learn are $\theta_h(0)$ and $\theta_h(1)$. Similarly, $\log b(X)$ would be written as a dot product which involves $\theta_b(\text{red}), \theta_b(\text{green}), \theta_b(\text{blue})$, and $\log g(X,Y)$ involves six other θ parameters. So if we concatenate all of the $3 + 6 + 2$ together into a single vector θ, we indeed get an expression

$$p(X, Y; \vec{\theta}) = \exp\left(\vec{\theta}^T \vec{f}(X, Y) - \ln Z(\vec{\theta})\right),$$

where for discrete MRFs $\vec{f}$ is the concatenation of indicator functions much like in the integer LP problem we've set up in previous lectures.

Fact 156

This is called the **overcomplete representation** because we do have redundancy – we'll talk about how to get a more compact representation (using exponential families) in a future lecture. For now, this works and captures what we want in our model.

Returning now to the learning problem, we have some iid data $\mathcal{D}$ and our goal is to maximize the average log-likelihood of our data points. That means our goal is to find

$$\theta^{\text{ML}} = \text{argmax}_\theta \log \prod_{\vec{x} \in \mathcal{D}} p(\vec{x}; \theta) = \text{argmax}_\theta \sum_{\vec{x} \in \mathcal{D}} \left(\vec{\theta}^T \vec{f}(\vec{x}) - \log Z(\vec{\theta})\right).$$

(So for each data point $\vec{x}$, we evaluate $\vec{f}$ on $\vec{x}$, take the dot product with our vector of parameters $\vec{\theta}$, and subtract off the log partition function.) Writing this out explicitly, we wish to find

$$\text{argmax}_\theta \sum_{\vec{x} \in \mathcal{D}} \sum_{c \in C} \sum_{\overline{\vec{x}}_c} \theta_c(\overline{\vec{x}}_c) 1\{\vec{x}_c = \overline{\vec{x}_c}\} - |\mathcal{D}| \log Z(\theta),$$

where unfortunately we cannot decompose the objective into clique potential terms because of the global normalization and thus it is much harder to solve for optimal parameters (since the first term is linear in θ while the second is not – even **computing** our likelihood requires sum-product inference, which is exponential in treewidth). Remembering that the parameters θ are unbounded, we see that the log-normalization constant is there so that we can't just increase our objective function by scaling up θ – our goal is to increase the weight of data points in $\mathcal{D}$, while not increasing the total weight of all $\vec{x}$ (not just those in $\mathcal{D}$) by too much.

Instead, we'll dig deeper into how the partition function term actually depends on θ. We can compute the gradient of $\log Z(\theta) = \log \sum_{\vec{x}} \exp\left(\vec{\theta}^T \vec{f}(\vec{x})\right)$ as follows:

$$\begin{aligned}
\frac{\partial \log Z(\vec{\theta})}{\partial \theta_i} &= \frac{1}{Z(\vec{\theta})} \frac{\partial \sum_{\vec{x}} \exp\left(\vec{theta}^T \vec{f}(\vec{x})\right)}{\partial \theta_i} \\
&= \frac{1}{Z(\vec{\theta})} \sum_{\vec{x}} \frac{\partial \exp\left(\vec{\theta}^T \vec{f}(\vec{x})\right)}{\partial \theta_i} \\
&= \frac{1}{Z(\vec{\theta})} \sum_{\vec{x}} f_i(\vec{x}) \exp\left(\vec{\theta}^T \vec{f}(\vec{x})\right).
\end{aligned}$$

But now we can push the Z back inside the sum and find that this expression is actually

$$\sum_{\vec{x}} f_i(\vec{x}) p(\vec{x}; \vec{\theta}) = \mathbb{E}_{p(\vec{x};\theta)}[f_i],$$

which means that the full gradient takes on a nice form:

$$\boxed{\nabla_{\vec{\theta}} \log Z(\vec{\theta}) = \mathbb{E}_{p(\vec{x};\theta)} \vec{f}(\vec{x})}.$$

So again, we can compute the gradient if we can solve an inference problem. Since the vector $\vec{f}$ is a collection of indicators in this case, we are actually calculating a **vector of marginal probabilities** (for example, the probability that $Y = 1$ is the marginal probability under p). And we can do a similar calculation to show that the second derivative of the log-partition function gives second-order moments – that is, the Hessian is

$$(\nabla^2 \log Z(\vec{\theta}))_{ij} = \left(\text{Cov}(\vec{f}(\vec{x}))_{ij}\right),$$

and since covariance matrices are positive semidefinite, this means $\log Z(\vec{\theta})$ is convex (and thus $-\log Z(\vec{\theta})$ is concave). Furthermore, the first term in our objective function is linear (so in particular concave). This is good because it means our function has only a "single peak" – it's easy to optimize with **gradient ascent**, since local maximizers are global maximizers. Gradient ascent is a simple optimization approach in which we maximize a function h via

$$\vec{x}_{t+1} = \vec{x}_t + \eta_t \nabla h(\vec{x}_t),$$

where η_t is some small stepsize. In this case, this means we initialize our parameters in some way, and then we repeatedly evaluate

$$\nabla_\theta \ell(\theta; \mathcal{D}) = \left(\frac{1}{|\mathcal{D}|}\sum_{\vec{x}\in\mathcal{D}} f(\vec{x})\right) - \nabla_\theta \log Z(\vec{\theta})_{\theta=\theta_t}$$

and take a step in the direction of that gradient. We know that at the maximum likelihood solution, the gradient is zero, and notice also that the expression above is actually the **difference between the empirical data average of $\vec{f}$ and the model expectation**! So there is a natural interpretation of what's actually happening here – we're predicting how frequently we expect $\vec{x}_c = \overline{\vec{x}}_c$ in our data, versus how frequently the model marginals expect this to happen.

Remark 157. *This is a special case of a what happens generally in maximum likelihood learning – we'll see when we study exponential families that moment matching between empirical data and the distribution is a general phenomenon.*

The main problem is that inference is complex and thus doing it repeatedly can be very computationally intensive – instead, we often need to do **learning with approximate inference** instead. All we actually need for learning is a good estimate of marginal distributions, so we can use Gibbs sampling or any other approximate techniques we've established so far to get savings. (So now we contrast the empirical "real samples" to the synthetic "fake samples" we generate via MCMC.)

Example 158

An alternative strategy is to change our learning objective completely – instead of minimizing KL divergence, we use a different estimator which lets us succeed at learning without needing inference. One such approach is called **pseudo-likelihood**, where we approximate the true likelihood with some other expression that's easier to work with in undirected models.

Remember that we can always write down any joint distribution as a product of conditional probabilities via the chain rule. We thus make the approximation

$$p(\vec{x}; \vec{\theta}) \approx p_L(\vec{x}; \vec{\theta}) = \prod_i p(x_i | x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n; \vec{\theta}) = \prod_i p(x_i, \vec{x}_{-i}; \vec{\theta});$$

that is, we condition **not just** on the variables that come before but also the variables that come after. The reason this is a good idea is that for undirected models, this expression can be written as $\prod_i p(x_i | \vec{x}_{N(i)}; \theta)$, where $N(i)$ is the Markov blanket of i. So the new task is to replace the average log-likelihood with the average pseudo-likelihood, and we can now **locally** normalize instead of having a global partition function: our objective function is

$$\ell_{\text{PL}}(\vec{\theta}; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{m=1}^{|\mathcal{D}|} \sum_{i=1}^{n} \log p\left(x_i^{(m)} | \vec{x}_{N(i)}^{(m)}; \vec{\theta}\right).$$

For example, for a pairwise MRF, the conditional probability here is just

$$p\left(x_i^{(m)} | \vec{x}_{N(i)}^{(m)}; \vec{\theta}\right) = \frac{1}{Z_i^{(m)}(\theta)} \exp\left(\sum_{j \in N(i)} \theta_{ij}(x_i^{(m)}, x_j^{(m)})\right),$$

where $Z_i^{(m)}$ only requires summing over a single variable (we now have many small partition functions instead of a large one).

Theorem 159

This new objective function is still concave, meaning we can again optimize it via gradient ascent. Call the maximizer $\vec{\theta}^{\text{PL}}$, and assume that data is drawn from an MRF with parameters $\vec{\theta}^*$. Under some conditions (which are true in practice). Then $\vec{\theta}^{\text{PL}} \to \vec{\theta}^*$ as the dataset gets large.

Intuitively, pseudo-likelihood matches conditional distributions based on local information. And we can imagine that if we run a Gibbs sampler on the model distribution and the data distribution using such local data, then we will get the same invariant distributions in both cases.

Example 160

Finally, we'll see how to extend all of this to conditional models – recall that conditional random fields are like Markov networks but only specify the conditional distribution $p(\vec{y}|\vec{x}) = \frac{1}{Z(\vec{x})} \prod_{c \in C} \phi_c(\vec{x}, \vec{y}_c)$.

This time, our partition function takes the form

$$Z(\vec{x}; \theta) = \sum_{\hat{y}} \prod_{c \in C} \phi_c(\vec{x}; \hat{\vec{y}}_c),$$

and if $\vec{Y}$ is simple (like in logistic regression where each Y_i only takes on values in $\{0, 1\}$) this can indeed be computed easily. Just like in the unconditioned MRF case, we can write this out this joint distribution as

$$p(\vec{y}|\vec{x}) = \frac{1}{Z(\vec{x})} \exp\left(\vec{\theta} \cdot \vec{f}(\vec{x}, \vec{y})\right),$$

just that our normalization constant is different based on the observed value of $\vec{x}$ and our feature functions $\vec{f}$ now depend on $\vec{x}$ (in potentially arbitrarily complicated ways).

Example 161

A setting in which we may want to find $\vec{\theta}$ would be segmentation, where we want to do a binary prediction task to find the "boundary" of some item in an image. Specifically, we may be given some images X along with associated segmentation masks Y, and we want to choose parameters that maximize $\sum_{i=1}^{M} \log p(\vec{y}^{(i)}|\vec{x}^{(i)}; \vec{\theta})$.

By definition, the log-likelihood of a single a data point is given by

$$\log p(\vec{y}^{(i)}|\vec{x}^{(i)}; \vec{\theta}) = \log\left(\frac{1}{Z(\vec{x}^{(i)}; \theta)} \prod_{c \in C} \exp\left(\theta_c \cdot f_c(x^{(i)}, y_c^{(i)})\right)\right),$$

where now instead of having $|\mathcal{D}|$ times the partition function $Z(\vec{\theta})$, we have a different partition function for each data point and have to add them up:

$$\theta^{\text{ML}} = \text{argmax}_\theta \left(\sum_{(\vec{x}^{(i)}, \vec{y}^{(i)}) \in \mathcal{D}} \sum_c \theta_c \cdot f_c(\vec{x}^{(i)}, \vec{y}_c^{(i)}) \right) - \sum_{(\vec{x}^{(i)}, \vec{y}^{(i)}) \in \mathcal{D}} \log Z(\vec{x}^{(i)}; \vec{\theta}).$$

So we can still do gradient ascent because we have a concave function, but now we need to do **one inference per training data point**, not just one per step! So it becomes increasingly important to have a graphical model where inference can be done efficiently while still taking into account some local interactions.

18 February 22, 2024

Today we'll discuss the expectation maximization algorithm and how to learn with missing data – specifically, we'll start on this with the Bayesian network case. Our setting is as follows: we have a joint probability distribution $p(\vec{X}, \vec{Z}; \theta)$, where in our data $\mathcal{D}$ the variables $\vec{X}$ are observed but the $\vec{Z}$ are not. For example, our data might look like $(0, 1, 0, ?, ?, ?)$, $(1, 1, 1, ?, ?, ?)$, and so on, or (in a more concrete example) we might have some emails which are labeled as spam or not spam while others remain unlabeled. These unlabeled emails may still be useful for determining our parameters, though – they may tell us something about how words tend to appear together.

We'll approach this problem by thinking about maximum likelihood again – again our goal is to maximize the objective

$$\ell(\theta; \mathcal{D}) = \log \prod_{\vec{x} \in \mathcal{D}} p(\vec{x}; \theta) = \sum_{\vec{x} \in \mathcal{D}} \log p(\vec{x}; \theta) = \sum_{\vec{x} \in \mathcal{D}} \log \left(\sum_{\vec{z}} p(\vec{x}; \vec{z}; \theta) \right),$$

where we're now summing out over the unobserved variables. The challenge now is that because of this inner sum, we do not have a closed-form solution for θ^* anymore. Even if we have a BN, we can't just count frequencies because marginal probabilities don't factor in a simple way – finding $\sum_{\vec{z}} p(\vec{x}; \vec{z})$ requires inference, and furthermore the likelihood function is **no longer necessarily unimodal** as a function of θ, since we may be adding together different unimodal distributions together.

Thus, we'll need a different strategy which takes an **iterative approach** – we'll connect the partially observed case to the fully observed case by inferring the missing components, and this leads us to the **expectation maximization algorithm**. The algorithm works as follows: we start with some initial guess $\theta^{(0)}$ for our parameters. Then we repeatedly iterate the following two steps until we get convergence:

1. (E-step) Complete the incomplete data with our current guess of parameters $\theta^{(t)}$.
2. (M-step) Update the parameters (finding the best fit) based on the completed (fully observed) dataset that we now have, yielding a new guess $\theta^{(t+1)}$.

Example 162

Recall that in a Bayesian network, our parameters θ are exactly the values appearing in our conditional probability tables. Suppose we have a BN on binary variables with directed edges $A \to C$, $B \to C$, and $C \to D$, and we have a data instance $(a, b, c, d) = (1, ?, ?, 0)$.

In step 1 above, what we do is that **for each possible completion** of values b, c, we can compute how likely the completion is **given** $a = 1$ and $d = 0$ and our current guess of θs. For example, we can calculate $w_1 = p(b = 1, c = 1 | a = 1, d = 0)$ by writing out the definition of conditional probability and using the BN structure to calculate both the numerator and denominator; similarly, we can calculate $w_2 = p(b = 1, c = 0|a = 1, d = 0)$, $w_3 = p(b = 0, c = 1|a = 1, d = 0)$, and $w_4 = p(b = 0, c = 10|a = 1, d = 0)$.

What we do now is think of the single incomplete data point as four separate weighted data samples $(1, 1, 1, 0)$, $(1, 1, 0, 0)$, $(1, 0, 1, 0)$, and $(1, 0, 0, 0)$ of weights w_1, w_2, w_3, w_4 respectively, and we maximize the probability of the parameters on this **larger, weighted** data set using the methods we talked about in previous lectures. And this M-step is pretty simple – it's similar to the unweighted MLE estimate, but now we need to weight the samples instead of just counting the configurations. For example, instead of $\theta^{\text{ML}}_{d=1|c=0}$ just being the number of samples with $d = 1, c = 0$ divided by the number with $c = 0$, we now have

$$\theta^{\text{ML}}_{d=1|c=0} = \frac{\text{weighted count}(d = 1, c = 0)}{\text{weighted count}(c = 0)} = \frac{\sum_{j=1}^m \sum_{\hat{\vec{z}}} 1\{\text{data point consistent with } d = 1, c = 0\} \cdot p(\hat{\vec{z}}|\vec{x}^{(j)}, \theta^{(t)})}{\sum_{j=1}^m \sum_{\hat{\vec{z}}} 1\{\text{data point consistent with } c = 0\} \cdot p(\hat{\vec{z}}|\vec{x}^{(j)}, \theta^{(t)})}.$$

But now we have new θ values, so we will end up with different weights when we do the E-step the next time.

Fact 163

Notice that this still requires us to do the same computation as marginalizing where we sum over all possible values of $\vec{z}$ – the point is just to have an algorithm for actually computing this expectation while avoiding gradient ascent on something that isn't unimodal. There are variants which can be more efficient – we'll see in a future lecture that approximate inference or variational inference gives us estimates of the posterior if we can't afford to compute it exactly, and we can also further simplify by just taking samples.

More formally, in the E-step, we're letting $\vec{x}$ denote our observed data and $\vec{z}$ our hidden data. We then "hallucinate missing values" by computing a distribution over our hidden variables using $\theta^{(t)}$, computing

$$Q^{(t+1)}(\vec{z}|\vec{x}^{(j)}) = p(\vec{z}|\vec{x}^{(j)}, \theta^{(t)})$$

for **each** training example $\vec{x}^{(j)}$ and **each** possible value of $\vec{z}$. Then in the M-step, we know that our marginal likelihood does not decompose when we sum over $\vec{z}$ inside the log, so we'll **use Jensen's inequality** (much like we did with KL divergence) to write

$$\log\left(\sum_{\vec{z}} P(\vec{z})f(\vec{z})\right) \ge \sum_{\vec{z}} P(\vec{z}) \log f(\vec{z}),$$

giving us a lower bound because of the concavity of log. (And the idea is that we're choosing a **particular** choice of P in the E-step so that this bound is quite tight, not just some arbitrary inequality.) So if we do the same trick as in importance sampling to write our sum as an expectation over Q, we have

$$\begin{aligned}
\ell(\theta; \mathcal{D}) &= \sum_{\vec{x}\in\mathcal{D}} \log \sum_{\vec{z}} p(\vec{x}, \vec{z}; \theta) \\
&= \sum_{\vec{x}\in\mathcal{D}} \log \sum_{\vec{z}} Q^{(t+1)}(\vec{z}|\vec{x}) \frac{p(\vec{x}, \vec{z}; \theta)}{Q^{(t+1)}(\vec{z}|\vec{x})} \\
&\ge \sum_{\vec{x}\in\mathcal{D}} \sum_{\vec{z}} Q^{(t+1)}(\vec{z}|\vec{x}) \log \frac{p(\vec{x}, \vec{z}; \theta)}{Q^{(t+1)}(\vec{z}|\vec{x})}.
\end{aligned}$$

Now we're trying to optimize over θ, and $Q \log Q$ doesn't depend on it, so we can write this further as

$$\ell(\theta; \mathcal{D}) \ge \sum_{\vec{x}\in\mathcal{D}} \sum_{\vec{z}} Q^{(t+1)}(\vec{z}|\vec{x}) \log p(\vec{x}, \vec{z}; \theta) + \text{constant}.$$

So in words, we go through our data set, look at all possible completions, and we weight by this weighting term $Q^{(t+1)}$

that we computed in the E-step. (For comparison, the fully observed case has objective function where all data points are equally weighted and we instead have the expression $\sum_{(\vec{x},\vec{z})\in\mathcal{D}} \log p(\vec{x},\vec{z};\theta)$.) So the lower bound that we are optimizing in the M-step can be thought of as the "likelihood of a fully-observed dataset," where we weight each term by the probability of seeing those $\vec{z}$s given the $\vec{x}$s.

Because maximum likelihood maximizes this right-hand side, which is a lower bound for the true objective function, what we're doing is a sensible strategy. And then we're choosing

$$\theta^{(t+1)} = \text{argmax}_\theta \sum_{\vec{x}\in\mathcal{D}} \sum_{\vec{z}} Q^{(t+1)}(\vec{z}|\vec{x}) \log p(\vec{x},\vec{z};\theta)$$

to be our parameters we feed into the next step of our algorithm! And in a BN, just like in the calculations in Example 162, we can optimize each CPT independently. Specifically, we choose each θ to be the ratio of the weighted counts of $(x_i, \vec{x}_{\text{Pa}(i)})$ to $(\vec{x}_{\text{Pa}(i)})$ when we have hidden variables.

Example 164

The whole argument we did above with Jensen's inequality works for **any** choice of Q, just like in importance sampling, so now we're going to explain why our particular choice is a particularly tight lower bound.

The point is that our objective function takes the form $\ell(\theta;\mathcal{D})$, and **at the point** $\theta = \theta^{(t)}$ (the choice of parameters we're currently using), the lower bound is actually tight – we have **exactly**

$$\ell(\theta^{(t)};\mathcal{D}) = \sum_{\vec{x}\in\mathcal{D}} \sum_{\vec{z}} Q^{(t+1)}(\vec{z}|\vec{x}) \log p(\vec{x},\vec{z};\theta^{(t)}) + \text{constant}.$$

(We can see this by plugging in $p(\vec{z}|\vec{x};\theta)$ in for $Q^{(t+1)}(\vec{z}|\vec{x})$; the log term inside just becomes $\log p(\vec{x};\theta)$, which doesn't depend on $\vec{z}$. And that's exactly the equality case for Jensen's inequality.) So while we do the M-step, any increase in this objective function has to actually increase the **true** marginal likelihood, which is what we care about!

For a picture to have in mind, think about having two "curves" in our parameter space, (1) the true marginal likelihood $L(\theta)$ (potentially nonconcave), and (2) its lower bound from Jensen's inequality $\ell(\theta|\theta^{(t)})$ (always concave). Importantly, we've chosen our weighting function so that these two curves touch at $\theta = \theta^{(t)}$. Then in the M step we ignore curve (1) and **globally maximize** the lower bound (2); whatever $\theta^{(t+1)}$ we end up with has a higher value for curve (2), so the true marginal likelihood in curve (1) cannot get worse! (And then using $\theta^{(t+1)}$, we do another E step, which corresponds to finding a different lower bound curve (2) which is below (1) but touching at $\theta = \theta^{(t+1)}$.)

Fact 165

Notice that this procedure may get stuck in local optima – this is related to the idea of confirmation bias and not finding the best possible marginal likelihood. But at least it's guaranteed to make progress at each step.

So this EM algorithm searches for estimates that maximize expected log-likelihood (hence the name), and each iteration we perform always increases the likelihood. However, this algorithm does depend on the estimated parameters we begin with (so it might be good to start with a few guesses for $\theta^{(0)}$ for inferring missing values). And the issue that we've discussed is that this doesn't get around still needing to do inference on a Bayesian network, which may be expensive in general. (The alternative is to do gradient ascent directly on the nonconcave objective function, and EM or some hybrid method tends to do better in practice.)

Example 166

We'll look at an application of this to a particular type of Bayesian network, the **Gaussian mixture model** (where we often use the algorithm for unsupervised learning). Consider a simple Bayesian network $Y \to X$, where Y is a categorical variable that can take k values and X is a continuous random variable such that $p(X|Y)$ is a multivariate Gaussian with mean μ_i and covariance matrix σ_i. (So we first pick some $Y = i$ with probability $P(Y = i)$, and then we generate a datapoint with mean and covariance given by (μ_i, Σ_i).)

Intuitively, multivariate Gaussians can be visualized roughly as ovals in $\mathbb{R}^n$, and a mixture of Gaussians is overlaying these ovals in space. Such a model allows us to **cluster data points** – it's a useful way of defining complex generative models in terms of simple components. (For example, a real dataset which describes (duration of eruption, time since last eruption) for eruptions of Old Faithful may not be fitted well to a 2-dimensional Gaussian – it may fit better to a mixture of two Gaussians corresponding to "small eruptions" and "big eruptions.") Formally, this means we have

$$p(\vec{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\vec{x}|\mu_k, \Sigma_k),$$

where π_k are nonnegative real numbers summing to 1. (Adding together such Gaussians gives us a much more flexible way to represent general distribution shapes.) In unsupervised learning, it's often the case that when modeling a dataset, we have different classes of datasets behaving in different ways (but where the behavior within each class is relatively similar). So we may want to determine where these clusters are without having any labeling of the data in the first place, and we can do so by calculating the **posterior probabilities** $p(Y = i|\vec{x})$ that a point was generated from the ith Gaussian.

The first simple building block is to figure out how to do this in the **fully supervised** setting where we see everything: for a univariate Gaussian, the maximum likelihood estimates are just

$$\mu^{\text{ML}} = \frac{1}{N}\sum_{i=1}^{N} x^{(i)}, \quad \sigma^2_{\text{ML}} = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \hat{\mu})^2,$$

the empirical mean and sample variance, and if we have a mixture of multivariate Gaussians, we similarly get

$$\boxed{\mu^{k,\text{ML}} = \frac{1}{n}\sum_{j=1}^{n} \vec{x}^{(j)}, \quad \Sigma^{k,\text{ML}} = \frac{1}{n}\sum_{j=1}^{n}(\vec{x}^{(j)} - \mu^{k,\text{ML}})(\vec{x}^{(j)} - \mu^{k,\text{ML}})^T},$$

where we're **only** summing over the points where we label with color k. What we're curious about is now the **unobserved** case, where we have missing data (the label of which Gaussian we're from), and this is where we can use EM. Our goal is now to find $\text{argmax}_\theta \prod_j p(y_j, \vec{x}_j)$ even though we don't know the y_js, meaning we want to instead maximize the **marginal likelihood**

$$\text{argmax}_\theta \prod_j p(\vec{x}_j) = \text{argmax}_\theta \prod_j \sum_{k=1}^{k} p(Y_j = k, \vec{x}^{(j)}).$$

So let's translate what the EM algorithm means in this case. We know how to do the M-step – if we have both the values of $\vec{X}$ and Y, then we do know how to compute the (weighted) MLE estimators explicitly. So in the E-step, we fill in the missing y-values, calculating the conditional probabilities that a point $\vec{X}^{(j)}$ comes from each of the Gaussians given our parameters θ. And in this case, both the E and M steps are simple and have closed form solutions because of the simple form of the Gaussian.

Put in different words, at each step, we have a guess for the means and covariance matrices for each of the k Gaussians, and the E-step computes the expected class of each datapoint by looking at how far we are from the means. Then given these class expectations – thinking of a data point as now being 80 percent of a data point in the first Gaussian and 20 percent of one in the second Gaussian – we adjust our mean and covariance to best fit these weighted points like in the boxed equation above, except we now take a weighted average weighted by $P(Y^{(j)} = k|X^{(j)})$ instead of a simple one.

Remark 167. *If we want to "push the means apart" because we have some prior knowledge, we can do so by adjusting our Bayesian network, treating the means themselves as random variables with certain priors. But we can still apply the same machinery and approach the problem in a principled way.*

19 February 23, 2024 (Section)

Today's section, taught by Charlie Marx, will cover **expectation maximization**. Specifically, we'll contextualize the algorithm relative to other optimization algorithms in ML like gradient descent.

We'll think about inference in latent variable models like we did in lecture – suppose we have two sets of variables $\vec{X}$ and $\vec{Z}$ where we never actually get to see $\vec{Z}$, and we want to estimate our parameters θ going into the model. (For example, we may have a Gaussian mixture model where the joint distribution is a mix of Gaussians each with their own mean and variance, and the latent variable is which Gaussian our data was drawn from.)

Our goal overall will be to maximize the marginal likelihood, but the issue here is that our parameters specify a distribution over both $\vec{X}$ and $\vec{Z}$. So we basically marginalize out over $\vec{Z}$, and what's tricky is that even with a friendly density this marginalization may not give us something with nice global structure (like concavity) – we're now taking the log of a sum of probabilities. So the philosophy we go with is one that comes up a lot in modern optimization methods: we use the local structure to get a hint for where the location might be (via a surrogate), and then we perform better on the surrogate in hopes of performing better on the original optimization problem too.

Usually we do **Taylor methods** for optimizing nonconvex functions like this: we may use gradient descent (thinking about the linear approximation, or equivalently the first order approximation), but this requires the inner sum $\sum_{\vec{z}_i} p(\vec{x}_i, \vec{z}_i|\vec{\theta})$ to be tractable, and it requires us to be able to compute the gradient efficiently too. Another method is **Newton's method**, where we do a parabolic (second order approximation) approximation to hope that we have a slightly better surrogate. The main problem is that we then need a way to compute the Hessian instead of just the gradient.

Fact 168

If we think about the landscape of optimization problems we may want to solve, some problems have a lot of structure (like a concave objection function), meaning that we can even find the global optima in closed form or at least nice solvers that approach at some proven rate. However, there are some more complex, non-concave objective functions, and for those there often isn't much we can do besides Taylor methods.

The idea of EM is that we do something between these two extremes – we can get more guarantee than just hoping gradient descent will work. The EM algorithm finds a surrogate function that is easy to maximize, and then it maximizes that instead of our original marginal log-likelihood objective. We saw in lecture that we do an approximation using Jensen's inequality

$$\ell(\theta) \geq \sum_i \sum_{\vec{z}_i} q_i(\vec{z}_i) \log \frac{p(\vec{x}_i, \vec{z}_i|\vec{\theta}}{q_i(\vec{z}_l)},$$

so that we can move the sum past the log, and then we can more tractably optimize this new expression instead. In this case we actually have a global relation between our approximator and the original objective – we get a global **lower bound** using our new function, which is better than what we get with Taylor methods. And if $p(\vec{x}_i, \vec{z}_i|\vec{\theta})$ takes some nice structure (like an exponential family or linear function), then the Jensen lower bound is actually nice as well (because perhaps we have concavity or something else that we can work with). So that gives us more guarantees when working with EM than we would from a Taylor approximation.

But we didn't get this for free – this only works because we're in some particular settings where the Jensen lower bound is actually easier to optimize. So the requirements for using EM are that the inner sum over z should again be tractable so that $p(\vec{z}|\vec{x})$ can be computed, and we should be in a setting like latent variable models where the Jensen bound is tractable.

As a review, the EM algorithm has two steps: we have a Jensen lower bound which is basically what we get if we plug in our proposed distribution over latents, and this is actually a **tight bound** (exactly equal to the objective function) at our current value of $\vec{\theta}$. The point is that $\vec{X}, \vec{\theta}$ give us $\vec{Z}$, and $\vec{X}, \vec{Z}$ give us $\vec{\theta}$ via inference, but we don't actually have **either** the values of $\vec{\theta}$ or $\vec{Z}$. So we'll "flip back and forth" between our guesses for the two and successively optimize in that way. Our E step fixes $\vec{\theta}$ and determines a distribution over $\vec{Z}$, and once we've **fixed** $\vec{Z}$ to be distributed from our guess $\vec{\theta}$, we can optimize over $\vec{\theta}$ to increase the log-likelihood from there.

Example 169
We can draw an analogy to Newton's method – in Newton's method we have an "E-step" where we compute the coefficients of a parabolic approximation to the function (so we fit a quadratic locally), and then we have an "M-step" where we maximize the parabola. But the key difference is that the parabola isn't always underneath the true function, so it's possible that we get unlucky and by moving to the new maximum of the surrogate we've actually decreased the function we care about.

20 February 27, 2024

Today's topic is **Bayesian learning** – we'll be formulating learning as an inference problem by thinking of parameters as random variables themselves, which will allow us to reduce **learning to inference**. This will have several advantages, such as incorporating prior knowledge and modeling uncertainty on the estimates that we've made (we don't just care about the weights but also some distribution for them).

Example 170
We'll start with some motivation: let X be a Bernoulli random variable with some single unknown parameter $\theta = p(X = \text{Heads})$.

We might imagine three scenarios: one with a dataset of 1 head and 2 tails, another with 10 heads and 20 tails, and a third with 100 heads and 200 tails. In all three cases, our MLE estimate is $\hat{\theta} = \frac{1}{3}$, but somehow the situation should be different in that we're a lot more confident about our answer in the third case versus the first one. So what we really want is some distribution of the form $p(\theta|\mathcal{D})$ rather than just this single number, and this is useful because we can use this to give more detailed predictions without being too confident in a particular parameter.

Example 171

Suppose we want to predict whether or not a swan can be black. If we've seen $N = 10$ white swans so far, then we may ask about the probability that the 11th swan X_{N+1} is black. Using the maximum likelihood estimate, we would say that the probability $\hat{\theta}$ of seeing a black swan is 0. But this might just be because we don't have enough data, and we should be a little more careful with these predictions.

What we'll do is treat θ as a random variable, and we do so with the **beta-Bernoulli model**. Specifically, we can imagine the following Bayesian network: let θ be the probability of heads, and let $X_1, \cdots, X_N$ be the observations in our dataset. We then have a (naive Bayes) Bayesian network where we have directed edges $\theta \to X_i$ for each i, where the conditional probability table for X_i is that given θ, X_i is Bernoulli with that parameter θ. In such a setting, we have

$$P(\mathcal{D}|\theta) = P(X_1, \cdots, X_N|\theta) = \prod_{j=1}^{N} \theta^{X_j}(1-\theta)^{1-X_j}$$

(since the X_is are independent **given** the value of θ), which can be equivalently written as $\theta^{N_H}(1-\theta)^{N_T}$, where N_H is the number of heads and N_T is the number of tails. Thus, we can use Bayes' rule to update our belief about θ:

$$P(\theta|\mathcal{D}) = \frac{P(\theta)P(\mathcal{D}(\theta)}{P(\mathcal{D})} \propto P(\theta)P(\mathcal{D}|\theta).$$

So all we need to calculate this is to specify some prior $P(\theta)$, and then we can plug in the expression for $P(\mathcal{D}|\theta)$ from above. The idea is that $P(\theta|\mathcal{D})$ will generally concentrate as our dataset gets larger, becoming more similar to the maximum likelihood learning we previously performed.

Given that θ is continuous in this problem, we'll want to think about how best to specify the prior $P(\theta)$. We could discretize it over $[0, 1]$ and use a table, or we can use some "computationally simple" continuous prior that is flexible enough to be updated and yet work well with the inference algorithms we've discussed so far (since most of what we've said so far only necessarily works with **discrete random variables** – things like variable elimination may not be so sensible anymore in general for the continuous case).

Definition 172

In the beta-Bernoulli model, we choose the prior to be the **beta distribution**

$$P(\theta|\alpha_H, \alpha_T) = \text{Beta}(\theta|\alpha_H, \alpha_T) = \frac{1}{B(\alpha_H, \alpha_T)}\theta^{\alpha_H - 1}(1-\theta)^{\alpha_T - 1}$$

on $[0, 1]$, where the normalizing constant $B(\alpha_H, \alpha_T) = \int_0^1 \theta^{\alpha_H-1}(1-\theta)^{\alpha_T-1}d\theta$ is called the **beta function**.

We call the parameters α **hyperparameters** (also **shape parameters**), and this is a nice form because having the prior take the same form as the conditional probability $P(\mathcal{D}|\theta)$ means that their product also has this same form. For example, a $\text{Beta}(1, 1)$ random variable is just the uniform distribution on $[0, 1]$, $\text{Beta}(30, 20)$ is quite sharply peaked around 0.6, and $\text{Beta}(0.3, 0.2)$ is peaked near $x = 0$ and $x = 1$ (so we do have flexibility in the shape of the distribution that can be modeled). It turns out that for the beta distribution, we have

$$\mathbb{E}[\theta] = \frac{\alpha_H}{\alpha_H + \alpha_T},$$

so this is very reminiscent of the MLE estimate from before, but we can now think of $\alpha_H + \alpha_T$ as a **measure of confidence** for our expectation (since the larger it is, the more peaked our distribution is around that mean). One way to interpret α_H, α_T is as some "number of virtual heads and tails we see before looking at our current data;" the larger they are, the more strongly we believe the initial shape of θ values.

Looking now at the distribution of the posterior, if our likelihood is $P(\mathcal{D}|\theta) = \theta^{N_H}(1-\theta)^{N_T}$ and the prior is $P(\theta|\alpha_H, \alpha_T) = \text{Beta}(\theta|\alpha_H, \alpha_T) = \frac{1}{B(\alpha_H, \alpha_T)}\theta^{\alpha_H - 1}(1-\theta)^{\alpha_T - 1}$, then we basically **add together real and virtual heads and tails**: the posterior is **also Beta** with the updated counts

$$P(\theta|\mathcal{D}, \alpha_H, \alpha_T) \propto \theta^{\alpha_H + N_H - 1}(1-\theta)^{\alpha_T + N_T - 1}.$$

We call this situation where the posterior is in the same family as the prior **conjugacy**.

Fact 173

We'll notice that this looks very similar to the heuristic of Laplace smoothing – in general, we can justify Laplace smoothing by using a beta prior over θ. If we assume that the prior takes the form $\text{Beta}(\alpha_H, \alpha_T)$, then the **prior** mean of θ is $\frac{\alpha_H}{\alpha_H + \alpha_T}$, and the **posterior** mean is $\frac{\alpha_H + N_H}{\alpha_H + N_H + \alpha_T + N_T}$, which is a convex combination of the prior mean and the max-likelihood mean $\frac{N_H}{N_H + N_T}$; specifically we can write

$$\mathbb{E}[\theta|\mathcal{D}] = \frac{M}{N+M} \cdot \frac{\alpha_H}{\alpha_H + \alpha_T} + \frac{N}{N+M} \cdot \frac{N_H}{N_H + N_T},$$

where the weights depend on the relative number of virtual data points M and real data points N. So we are shrinking our estimate away from the MLE towards prior, which can be thought of as a smoothing effect, but as $N \to \infty$ that effect goes away.

Previously in maximum likelihood learning, we would just give a single best guess for θ^*, meaning that in our model the probability of our next coin flip being heads would just be θ^*. The question in the Bayesian prediction setting is to find the probability that the next flip X is heads, given the observed data $\mathcal{D}$, where θ is now a random variable. So we'll compute the probability is heads by **assuming** X is drawn independently from the same distribution as the dataset so far, given θ (so imagine adding a new node and edge in our Bayesian network $\theta \to X$). We then have to marginalize out over θ by integrating and accordingly weighting:

$$p(X = H|\mathcal{D}) = \int_0^1 P(X = H, \theta|\mathcal{D})d\theta = \int_0^1 P(X = H|\theta)P(\theta|\mathcal{D})d\theta.$$

So this is very similar to the sum-product computations we've previously had to do, except now we integrate instead of summing. (We can think of this as averaging over a big ensemble of different models with different values of θ, instead of using a single one with a particular θ – if it were discrete, this would look something like $0.5P(X = H|\theta_1) + 0.2P(X = H|\theta_2) + 0.3P(X = H|\theta)$.) Such a task can be very expensive, but making forecasts in this case is pretty easy, because in beta-Bernoulli we have

$$\int_0^1 P(X = H|\theta)P(\theta|\mathcal{D})d\theta = \int_0^1 \theta P(\theta|\mathcal{D})d\theta$$

and this is exactly the expression for the posterior mean $\mathbb{E}[\theta|\mathcal{D}]$ (which we know how to compute **because** we have a Beta posterior distribution).

Example 174

This solves the black swan event problem – if we start with a Beta$(1, 1)$ uniform prior for θ, then our posterior prediction is of the form

$$P(X_{N+1} = \text{black}|N_{\text{white}}, N_{\text{black}}) = \frac{N_{\text{black}} + 1}{N_{\text{white}} + N_{\text{black}} + 2},$$

which will always be strictly between 0 and 1 even if we haven't observed any swans of one type or the other.

Fact 175

It's often easier to compute the **posterior mode** than to compute the posterior mean (since the latter requires us to integrate over all possible models, but the former is just an optimization problem). So often in practice we will compute $\hat{\theta} = \text{argmax}_\theta P(\theta|\mathcal{D})$, where we still look at the posterior but instead of integrating we just take the most likely value.

In such a situation we are trying to find

$$\text{argmax}_\theta P(\theta|\mathcal{D}) = \text{argmax}_\theta \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})} = \text{argmax}_\theta P(\mathcal{D}|\theta)P(\theta),$$

and we call this **maximum a posteriori estimation**. We can think of this as doing a **regularized MLE**

$$\hat{\theta} = \text{argmax}_\theta \log P(\mathcal{D}|\theta) + \log P(\theta),$$

where we now have an extra term coming from the prior distribution which acts as a regularizer. This is often done in practice, and we can often think about regularizers in problems like this as being priors even if they're not usually phrased in terms of Bayesian learning! For example, when doing linear regression, we may view it as a conditional model where $P(Y|\vec{X} = \vec{x})$ is Gaussian with mean $\theta\vec{x}$ and some fixed variance 1. Then the MLE estimate corresponds to a least-squares objective function, **but** if we assume a Gaussian prior of the form

$$P(\theta|\lambda) \propto \exp\left(-\frac{\lambda}{2}\theta^T\theta\right),$$

which corresponds to assuming that the weight vector has mean 0 and covariance matrix $\lambda^{-1}I$ (saying that big coefficients are unlikely to occur a priori), then the maximum a posteriori estimation now has an extra term $\log P(\theta|\lambda) = -\frac{\lambda}{2}||\theta||^2$. So this yields the **ridge regression** model, encouraging us to find solutions with small coefficient vectors.

Example 176

Suppose now that we want to model a categorical random variable with k possible outcomes instead of just a Bernoulli random variable with 2 outcomes. Now we have a vector $\theta = (p_1, \cdots, p_K)$ encoding the probabilities of the various outcomes, where $\sum_k p_k = 1$.

In this case, the likelihood of our data is given by

$$P(\mathcal{D}|\theta) = \prod_{j=1}^{N}\prod_{k=1}^{K} p_k^{1\{X_j=k\}} = \prod_{k=1}^{K} p_k^{\#\{X_j=k\}}.$$

In this case, the conjugate prior to use is the **Dirichlet distribution**

$$P(\theta|\vec{\alpha}) = \text{Dirichlet}(p_1, \cdots, p_K|\alpha) = \frac{1}{B(\alpha)}\prod_{k=1}^{K} p_k^{\alpha_k - 1},$$

supported on the set of points $(p_1, \cdots, p_K)$ where $p_k > 0$ for all k and $\sum_k p_k = 1$. Multiplying those expressions gives us something with the same functional form (just that the exponent $\alpha_k - 1$ becomes $\alpha_k + \#\{X_j = k\} - 1$), which makes this setting tractable – we just end up with a Dirichlet distribution with α'_k equal to the number of virtual counts α_k plus the number of real data counts $\#\{X_j = k\}$.

We may now want to think about more **general Bayesian networks** – so far, we've only considered the case with a single categorical random variable (that is, a single vector of parameters), but we may want to apply this to a more complicated graph structure with many more conditional probability tables. It turns out the machinery is very similar – if we have a fully observed dataset, we can again reduce to an inference problem.

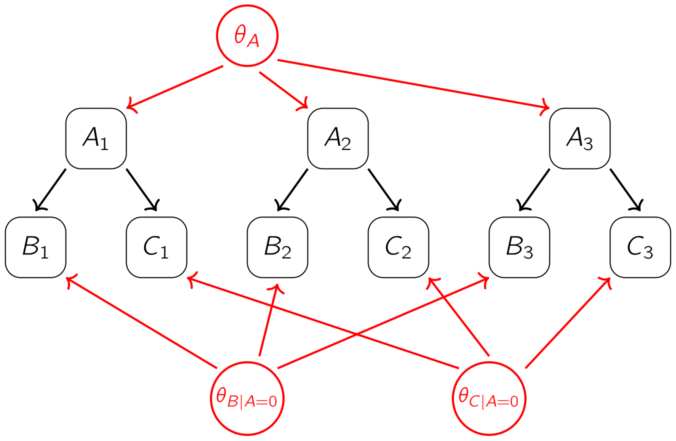
Example 177

Suppose we have binary random variables A, B, C with the BN structure $A \to B$, $A \to C$ (call this BN G), and we have a dataset of five data points for (A, B, C):

$$\mathcal{D} = \{(0, 0, 1), (1, 0, 1), (1, 0, 1), (0, 0, 0), (0, 1, 0)\}.$$

In this setting, we need to estimate five different parameters, namely $\theta_A, \theta_{B|A=0}, \theta_{B|A=1}, \theta_{C|A=0}$, and $\theta_{C|A=1}$. In a Bayesian framework, we need to treat all of these as random variables, and we do so by first specifying some prior for **all of them**. For simplicity, we can discretize θ, and suppose we have a prior of the following form: we know **for sure** that $\theta_{B|A=1} = 0.1$, $\theta_{C|A=1} = 0.8$, but a priori we think θ_A is either 0.75 or 0.9 with equal probability, and similarly we have a distribution for $\theta_{B|A=0}$ (it's either 0.25 or 0.5) and $\theta_{C|A=0}$ (it's either 0.5 or 0.75).

In such a setting, we can follow the structure of the naive Bayes BN $\theta \to X_i$ from above, thinking of each X_i as being itself a collection of random variables in a Bayesian network and θ as the collection of all CPTs. We call this a **meta network** – we have a joint distribution over data and network parameters, where each training data point gives us a copy of the original Bayesian network G (so we would have A_i, B_i, C_i nodes corresponding to the instantiations of A, B, C in the ith data point). We then have random variables $\theta_A, \theta_{B|A=0}, \theta_{C|A=0}$ which represent unknown parameters – these have no parents, and their CPTs are where we specify the priors from the previous paragraph. (So θ_A would have probability 0.5 of being 0.75 and probability 0.5 of being 0.9.) But once we condition on these θs, we make each individual BN behave like G but with the specified parameters θ. (So for example, the distribution of B_3 is specified conditioned on the value of A_3 and also the value of $\theta_{B|A=0}$ in the meta network.) The diagram below shows what the meta-network looks like if we have three data points:



Thus **conditioned on some realization of the parameters θ, each model behaves like the original G with that particular realization**. We'll then let the dataset be the evidence on our meta network, and then we can use this to compute the **posterior distribution** on the network parameters: that is, in this setting we would compute

$$P(\theta_A, \theta_{B|A=0}, \theta_{C|A=0}|\mathcal{D}).$$

So everything falls out automatically to the extent that we can do inference – since all θs are d-separated when we condition on data, we can do the inference separately for each one. So Bayes' rule has given us a way to update beliefs, **reducing learning to a process of computing posterior distributions**.

Fact 178
The setting above assumed that our random variables $\theta_A, \theta_{B|A=0}, \theta_{C|A=0}$ were discretized, so that we can use inference algorithms from our discrete BNs from earlier in the course. But we can also use continuous priors and posteriors, and a natural way to do this is to use a Beta or Dirichlet prior for these θs (depending on how many outcomes we're dealing with); we'll again end up in the same family of variables. For example, we can encode the belief that $\theta_A \approx 0.75$ by starting with a Beta(75, 25) distribution, and then our predictions will integrate over all BNs over choices of our paramteres.

In particular, we estimate probabilities via equations like

$$P(B = b, C = c) = \sum_{\theta} P_\theta(b, c) P(\theta|\mathcal{D})$$

or an integration if we have a continuous ensemble of possible distributions. And just like in the black swan example, this is **more robust than MLE but also more computationally expensive**.

21 February 29, 2024

We'll discuss **structure learning today** – so far, we've assumed a graphical model with some fixed set of conditional independencies and computed the parameters θ from that, but today we'll see how to use data to infer what independencies exist (that is, how to construct our network for the graphical model). We'll learn the Chow-Liu algorithm for structure search, and then we'll think about how to view structure learning as a search problem (via optimization).

Remark 179. *Network structure gives us a lot of information about how our random variables are related, and in many situations with scientific data this relation is actually what we care about (for example understanding if some variables are correlated or seeing the "directions of causality" (is smoking causing cancer, or is cancer causing smoking).*

There are two main methods that we'll discuss today for structure learning: the first is **constraint-based** (where we check for independence and add edges accordingly), and the other is **score-based** (search for network structures that maximize the probability of our particular data set via a loss function).

First of all, recall from earlier in the course that for a directed acyclic graph G we defined $I(G)$ to be the set of independencies implies by G (via d-separation), and we defined $I(p)$ to be the set of conditional independencies that actually occur from the joint distribution. Our goal is to find some G such that $I(G) = I(p)$ (a P-map), but this is not always possible and we can only find a minimal I-map (where $I(G) \subseteq I(p)$ and removing any edges in G would make this no longer true). The algorithm we discussed for finding minimal I-maps was to pick an ordering of variables $X_1, \cdots, X_n$ (corresponding to a chain factorization for the joint distribution) and then repeatedly simplifying each term as much as possible, pruning parents that are not necessary. In other words, for each X_i, we find the **minimal** subset $\vec{A}_i \subseteq \{X_1, \cdots, X_{i-1}\}$ such that

$$p(X_i|X_1, \cdots, X_{i-1}) = p(X_i|\vec{A}_i).$$

Then once we have this subset of variables, we can use standard learning methods to learn the relevant conditional probability table. So this will give us a minimal I-map, but it's not unique (it depends on the order that we choose).

The idea is that **we can use this same algorithm to learn the BN structure from data**, even if we **don't have a method** for testing conditional independence like we did before! Supposing now that we only have data $\mathcal{D}$ (rather than the distribution specified via probability tables), we can try using empirical probabilities and doing **hypothesis testing**. We know that if X and Y **are** independent, then $p(X, Y) = p(X)p(Y)$, which implies that in our empirical distribution we will have $\hat{p}(X, Y) \approx \hat{p}(X)\hat{p}(Y)$. So for every possible value of X and Y, we can see how frequently those occur together and how close the independence equation is to holding – we don't expect perfect equality, but the larger our data set is the closer the empirical frequencies should match. We can do this via the **chi-square distance** (think of this as squared loss)

$$d_{\chi^2}(\mathcal{D}) = |\mathcal{D}| \sum_{x,y} \frac{(\hat{p}(x, y) - \hat{p}(x)\hat{p}(y))^2}{\hat{p}(x)\hat{p}(y)};$$

if this quantity is small, then we can conclude that X and Y are independent. And alternatively we can consider the **mutual information** (think of this as a different loss function)

$$MI(X, Y) = D_{\text{KL}}(p(x, y)||p(x)p(y)) = \sum_{x,y} p(x, y) \log\left(\frac{p(x, y)}{p(x)p(y)}\right) \approx \sum_{x,y} \hat{p}(x, y) \log\left(\frac{\hat{p}(x, y)}{\hat{p}(x)\hat{p}(y)}\right);$$

we know that if X and Y are independent then the mutual information should be zero, and thus if our approximation from $\hat{p}$ is small then we can conclude that we do have independence. So in both of these cases, we can use this as our "test for the minimal parent set" $\vec{A}_i$, and then we get a minimal I-map by doing this over some particular ordering.

Fact 180

The cons of this strategy are that we are likely to make mistakes when we have to do repeated conditional independence tests at every stage – it's especially big of a problem when our dataset is small. And if we are doing tests for independence with more than two variables, we will need more and more data to estimate the entries accurately. But if we impose some constraints on the maximum number of allowed parents, then it does a pretty good job.

Instead, thinking about this structure learning problem as an **optimization problem** often works better. We can start with a dataset $\mathcal{D}$ and a set of possible graphs (for example Bayesian networks) over our variables, and we can set up a performance metric that tells us how well each graph does for representing our data. Then we can go through our set of possible graphs and pick the one with the best performance. If we have BNs, we know that we can find the optimal parameters very easily via maximum likelihood estimation, and this means we can calculate the log-likelihood of data for each potential graph quickly. (Then a higher log-likelihood would mean that we fit the data better.)

The problem here is that if we just want to find a graph G that maximizes $\text{argmax}_G \text{LL}(\mathcal{D}|G)$, but the optimal solution is always a complete DAG G which tells us nothing about the structure (we showed in our homework that adding edges always increases the log-likelihood). So one strategy is to **regularize to penalize graphs that are too complex**, adding another term besides the log-likelihood, and the other is to just have a hard constraint where we only consider some subset of simpler graphs G (this is analogous to only considering linear models in machine learning).

Example 181

One strategy is to only consider **tree-structured Bayesian networks** in which each node only has at most one parent. There are still (exponentially) many tree networks to enumerate over, but there is a clever algorithm called the **Chow-Liu algorithm** which lets us find the best tree in complexity that is only quadratic in the number of variables.

This algorithm makes use of the mutual information we defined earlier in the lecture – for each pair of variables X, U, the mutual information $MI(X, U)$ tells us how much one variable helps us predict the other one. Like we mentioned before, we will approximate the true mutual information by the one coming from the empirical distribution: **for all** X, U**, compute**

$$MI_{\hat{p}}(X, U) = \sum_{x,u} \hat{p}(x, u) \log \frac{\hat{p}(x, u)}{\hat{p}(x)\hat{p}(u)},$$

where remember that $\hat{p}(x_i, x_j)$ is the fraction of all data points which have $X_i = x_i, X_j = X_j$. This will then give us a **mutual information graph**, where between each pair of variables we label with some real number (here we use that mutual information is symmetric). We can then select a skeleton for the Bayesian network which chooses edges with large mutual information: that is, we find the **maximum spanning tree** (a tree with maximal sum of weights along the edges) via the Kruskal or Prim algorithm, where we just greedily add edges while making sure that we have a tree at each step.

Once we have our undirected edges, we need to pick a directed acyclic graph, and we can do this by picking any node as the root and assigning all arrows to point away from that root. (The idea is that if we want a tree-structured Bayesian network, we must avoid V-structures.) This gives us an overall Bayesian network; computing the MI pairs takes $O(n^2)$ time and the maximum spanning tree computation also takes $O(n^2)$ time, so our total complexity is $O(n^2)$.

Example 182

The key fact we're using here is that **log-likelihood is essentially dictated by sum of mutual information**, and we'll go through the derivation now!

Recall that if we have some fixed graph structure, our **data** log-likelihood

$$\log p(\mathcal{D}|\theta, G) = \sum_i \sum_{x_i} \sum_{\vec{x}_{\text{Pa}(i)}} \text{Count}(x_i, \vec{x}_{\text{Pa}(i)}) \log p(x_i|\vec{x}_{\text{Pa}(i)}; \theta),$$

so that when we plug in the MLE parameters θ^{ML}, we get the maximum likelihood score

$$\log p(\mathcal{D}|\theta^{\text{ML}}, G) = |\mathcal{D}| \sum_i \sum_{x_i} \sum_{\vec{x}_{\text{Pa}(i)}} \hat{p}(x_i, \vec{x}_{\text{Pa}(i)}) \log \hat{p}(x_i|\vec{x}_{\text{Pa}(i)}).$$

If we do a bit of algebra, we can rewrite this as

$$|\mathcal{D}| \sum_i \sum_{x_i} \sum_{\vec{x}_{\text{Pa}(i)}} \hat{p}(x_i, \vec{x}_{\text{Pa}(i)}) \log \left(\frac{\hat{p}(x_i, \vec{x}_{\text{Pa}(i)})}{\hat{p}(\vec{x}_{\text{Pa}(i)})} \frac{\hat{p}(x_i)}{\hat{p}(x_i)} \right),$$

but then rearranging this (and applying the definitions of entropy and mutual information) simplifies the log-likelihood to

$$\boxed{|\mathcal{D}| \sum_i MI_{\hat{p}}(X_i, \vec{X}_{\text{Pa}(i)}) - |\mathcal{D}| \sum_i H_{\hat{p}}(X_i)},$$

where $H_{\hat{p}}(X_i)$ is the empirical entropy for the random variable X_i – in particular, it doesn't depend on the structure of the graph and is just some measurement of individual randomness! So maximum likelihood likes graphs where parents of a node are **informative of the node** (meaning we have high predictive power). If we're assuming that G has to be a tree where each node has at most one parent, then each term $MI_{\hat{p}}(X_i, \vec{X}_{\text{Pa}(i)})$ simplifies to just adding the mutual information over the edges (i, j), and thus we've justified that our max spanning tree problem is actually doing the correct thing.

Fact 183

Notice however that we are still not doing causal inference – the directionality of the tree does not affect the log-likelihood or mutual information, and that's a **fundamental limitation** we have here.

We may ask whether we can use this same machinery to optimize over all graphs G, but again complete DAGs give us the optimal solution and often they are **overfitting the data** (doing well against the training data doesn't mean we have a good model of the world). So again, we must add either a soft (regularization) or hard constraint to prefer simpler models (in the spirit of Occam's razor). To turn this in to an algorithm, we need a way to **specify the complexity of a model** and also a way to **balance that complexity with how well we fit the data**.

Definition 184

In Bayesian networks, we measure model complexity by using the number of independent parameters in the model (essentially the number of entries in the conditional probability tables, minus one per row because probabilities must sum to one). We call the number of independent parameters the **dimension** of G and denote it $||G||$.

For example, if we have binary random variables A, B, C, D, we need $1 + 2 + 2 + 2 = 7$ parameters if we have a tree-structured Bayesian network, and we need $1 + 2 + 4 + 8 = 15$ parameters if we have a fully connected DAG. Our new **score function** then takes the form

$$\text{Score}(G : \mathcal{D}) = LL(\mathcal{D}|G) - \psi(|\mathcal{D}|)||G||,$$

where we have a term for the data fit, but we also get a penalty for complexity proportional to the dimension of G. The extra degree of freedom $\psi(|\mathcal{D})|$ here is governed by the size of our dataset; recall that the log-likelihood expression boxed above scales linearly in $|\mathcal{D}|$.

- One choice is to set $\psi(|\mathcal{D}|) = 1$ to get the **Akaike information criterion (AIC)**, in which model complexity is a secondary issue and only used when different models have very similar log-likelihoods.
- Another choice of $\psi(|\mathcal{D}|) = \frac{1}{2}\log(|\mathcal{D}|)$ yields the **Bayesian information criterion (BIC)**, where again the influence of model complexity is less relevant than the log-likelihood term. The idea is that we do penalize for complexity, but we do so correctly from a Bayesian treatment if we use Dirichlet priors – in the limit of infinite data there is no overfitting, so it makes sense for log-likelihood to eventually dominate. (This also has some relation to the "shortest way to describe a dataset.")

Theorem 185

BIC is consistent. Specifically, if our data is generated from a distribution p^* and G^* is a perfect map for p^*, then as $|\mathcal{D}| \to \infty$, our scoring function satisfies $\text{Score}(G : \mathcal{D}) \le \text{Score}(G^* : \mathcal{D})$ for all G, and if G' is not I-equivalent to G^*, then we have a strict inequality.

Remember that the models $A \to B$ and $B \to A$ are I-equivalent, and indeed this whole framework does not give us a way to distinguish between them. But other than that, we'll have the "correct structure of p^*" up to I-equivalence. And note that if we only have likelihood score, G^* will maximize the score, but so will the fully-connected DAG (which is not I-equivalent to G^* in general), so not having a penalty on model complexity doesn't give us this consistency property that we want.

Example 186
However, notice that we are still searching for a network structure that achieves the highest score, and there's an exponentially large number of graphs to search through (we can't do gradient-style methods because we have to do every DAG separately).

So this can be intractable in general (we could only do it for trees with a clever algorithm), and the way we have to do things in general is to do **local search**, picking the best solution in a greedy way. For example, we can start with a good graph structure that we think may maximize the score, and we have some local procedure for making small changes and seeing if we get a better score for each of them (for example adding edges, removing edges, or flipping their direction). This process is more efficient because score decomposes additively over conditional probability tables, meaning that we can evaluate these local changes efficiently.

Alternatively, we can again constrain our search space by doing the generation step-by-step (like the way we found our I-map). Specifically, we choose some ordering of variables, and then for each X_i we find some good set of previous variables $\vec{U}_i$ that maximize local score. This doesn't force us to worry about creating cycles and lets us decompose into optimization problems, and as long as we bound the number of maximum parents d we can brute-force this computation. But this can still be hard for large d, and again the final outcome depends on what particular variable ordering we have chosen.

Fact 187
While we don't have much time to talk about it in lecture, we can look up further discussion about evaluating causality in the textbook!

22 March 5, 2024

We'll talk today about **exponential families** and introduce a bit of informtion geometry as well. The idea is just to develop a framework for the ideas we've talked about so far in learning and inference, showing some connections to statistics and optimization. (There won't be any new algorithms today.)

As a recap, when doing parameter learning in a Markov random field, we have some joint distribution $p(\vec{x}) = \exp\left(\sum_{c\in C} \theta_c(\vec{x}_c) - \ln Z\right)$ (here the θ_cs are the logs of the entries in the factors ϕ_c), where the normalization constant Z is given by a sum of exponentials $\sum_{\vec{x}} \exp\left(\sum_{c\in C} \theta_c(\vec{x}_c)\right)$. In particular, in the learning setting, we get a family of probability distributions parameterized by θ of the form

$$p(\vec{x};\theta) = \exp\left(\sum_{c\in C} \theta_c(\vec{x}_c) - \ln Z(\vec{\theta})\right),$$

where the tricky thing is that this joint probability distribution depends simply on θ in the first term but not the second. Such a distribution can be expressed in the **log-linear form** $p(\vec{x};\theta) = \exp\left(\vec{\theta}^T \vec{f}(\vec{x}) - \ln Z(\theta)\right)$, where in the discrete case the function $\vec{f}$ is a concatenation of indicator functions for various realizations of our variables in a clique $\vec{x}_c$.

Writing it in this form is nice because it means the log-likelihood has a linear term in addition to some other nice convex term.

Definition 188

Distributions in the **exponential family** take the form

$$p(\vec{x}; \vec{w}) = h(\vec{x}) \exp\left(\vec{w}^T \vec{f}(\vec{x}) - A(\vec{w})\right),$$

where $h(\vec{x})$ is some **base measure** (often a constant), $\vec{w} \in \mathbb{R}^d$ are **natural parameters**, the function $\vec{f}(\vec{x})$ is called the **sufficient statistic**, and $A(\vec{w})$ is the **log-partition function** which normalizes the distribution.

In particular, we have the expression (note that $\vec{x}$ can be continuous now, and in many settings it often is)

$$A(\vec{w}) = \log\left(\int h(\vec{x}) \exp[\vec{w}^T \vec{f}(\vec{x})] d\vec{x}\right).$$

We can think of $\vec{f}$ as constructing some set of properties (features) which are the only ones that affect how likely $\vec{x}$ is in the model. And $\vec{w}$ takes the role of $\vec{\theta}$ that we had before.

Example 189

A Bernoulli distribution with parameter $\pi = P(X = 1)$ takes the form $p(x|\pi) = \pi^x(1-\pi)^{1-x}$, where x takes either value 0 or 1. In particular, this means

$$p(x|\pi) = \left(\frac{\pi}{1-\pi}\right)^x (1-\pi) = \exp\left(\log\left(\frac{\pi}{1-\pi}\right) x + \log(1-\pi)\right).$$

Pulling out the constant, this means we can pattern-match to the form of an exponential family: we have $\boxed{h(x) = 1}$, $\boxed{f(x) = x}$, $\boxed{w = \log\left(\frac{\pi}{1-\pi}\right)}$, and $\boxed{A(w) = -\log(1-\pi) = \log(1+e^w)}$ (using the expression for w we just found). Importantly, notice that we don't want to pull that $\log(1-\pi)$ term out of the exponential – it depends on our parameters, so we shouldn't treat it as the "base measure."

Example 190

The density for a Gaussian takes the form

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

We can do some algebra here to rewrite this in the same form as above as well.

Specifically, we have

$$p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{\mu}{\sigma^2} x - \frac{1}{2\sigma^2} x^2 - \frac{1}{2\sigma^2}\mu^2 - \log\sigma\right).$$

Thus this time we want our sufficient statistic to be $\boxed{\vec{f}(x) = \begin{bmatrix} x \\ x^2 \end{bmatrix}}$, meaning that $\boxed{\vec{w} = \begin{bmatrix} \mu/\sigma^2 \\ -1/2\sigma^2 \end{bmatrix}}$. Then the log-normalizing constant is $A(\vec{w}) = \frac{\mu^2}{2\sigma^2} + \log\sigma$; again we want to rewrite this in terms of the parameters w_1, w_2 and we find that $\boxed{A(\vec{w}) = -\frac{w_1^2}{4w_2} - \frac{1}{2}\log(-2w_2)}$ and $\boxed{h(x) = \frac{1}{\sqrt{2\pi}}}$.

Remark 191. *Many other probabilistic models that are widely used also happen to be exponential families, such as log-linear models, multinomial distributions, as well as Poisson, exponential, Gamma, Weibull, chi-square, Dirichlet, and geometric distributions.*

Thus, it's interesting to study the properties of such distributions that can be written in this functional form – many of the properties we found for MRF will turn out to hold in general too! As mentioned above, the sufficient statistic is a function that summarizes our data in the sense that for any dataset of iid samples $\{\vec{x}_1, \cdots, \vec{x}_M\}$, the likelihood can be written as

$$\begin{aligned} p(\vec{x}_1, \cdots, \vec{x}_M) &= h(\vec{x}_1) \cdots h(\vec{x}_M) \exp\left(\vec{w}^T \vec{f}(\vec{x}_1) - A(\vec{w})\right) \cdots \exp\left(\vec{w}^T \vec{f}(\vec{x}_M) - A(\vec{w})\right) \\ &= h(\vec{x}_1) \cdots h(\vec{x}_M) \exp\left(\vec{w}^T \sum_{m=1}^{M} \vec{f}(\vec{x}_m) - MA(\vec{w})\right). \end{aligned}$$

So if we want to estimate $\vec{w}$ via MLE (or see how good a parameter is for a particular dataset), we just need to sum the sufficient statistic function over all data points, regardless of how big the set is – we always condense all the information into a single $\mathbb{R}^d$ vector. And a similar calculation shows that the sufficient statistic is also all that is required for a Bayesian approach where we compute the posterior $p(\vec{w}|\mathcal{D})$.

Definition 192

Suppose $X_1, \cdots, X_M$ are random samples taken from some distribution $p(X|\vec{w})$ depending on some unknown parameter $\vec{w}$. A **statistic** is a real-valued function $T(X_1, \cdots, X_M)$ of our sample (for example the sum, maximum, or median); in particular this is some random variable. We say that a statistic is **sufficient** if the conditional distribution of $\vec{w}$ given $T = t$ does not depend on the dataset; that is,

$$(X_1, \cdots, X_M) \perp \vec{w} | T(X_1, \cdots, X_M).$$

The point is that knowing T is enough – knowing the exact values of the data points won't give us any more information about our parameters, so inference only depends on T. It's difficult to find sufficient statistics in general, but for exponential families the point is that $\vec{f}$ is sufficient because of the functional form. To do this we must show that $\vec{x} \perp \vec{w} | f(\vec{x})$, and we show this by computing (for any fixed value $\overline{\vec{f}}$ for $\vec{f}$)

$$p(\vec{f} = \overline{\vec{f}} | \vec{w}) = \sum_{\vec{x} : f(\vec{x}) = \overline{f}} h(\vec{x}) \exp\left(\vec{w}^T \vec{f}(\vec{x}) - A(\vec{w})\right),$$

and now because the density only depends on $\vec{x}$ through $\vec{f}$ we can simplify this to

$$\sum_{\vec{x} : f(\vec{x}) = \overline{f}} h(\vec{x}) \exp\left(\vec{w}^T \overline{\vec{f}} - A(\vec{w})\right) = \exp\left(\vec{w}^T \overline{\vec{f}} - A(\vec{w})\right) \sum_{\vec{x} : f(\vec{x}) = \overline{f}} h(\vec{x}).$$

But then evaluating $p(\vec{x} | \vec{f} = \overline{f}, \vec{w})$ via the definition of conditional probability, the expression simplifies because the $\exp\left(\vec{w}^T \overline{\vec{f}} - A(\vec{w})\right)$s cancel out: we just have

$$p(\vec{x} | \vec{f} = \overline{f}, \vec{w}) = \frac{h(\vec{x})}{\sum_{\vec{x} : f(\vec{x}) = \overline{\vec{f}}} h(\vec{x})},$$

which doesn't depend on $\vec{w}$, so we get the desired conditional independence. And the same proof shows that the sum of the sufficient statistic function $T(\vec{x}_1, \cdots, \vec{x}_M) = \sum_{m=1}^{M} \vec{f}(\vec{x}_m)$ is sufficient for $\vec{w}$ if we have M iid samples as well. (So if we want to estimate a Gaussian from a dataset, we just need the sum of x_i and sum of x_i^2 across all data points.

We'll now turn to the log-normalization term $A(\vec{w})$; recall that this log-partition function has some nice properties in the MRF case. To recover them here, we'll look at the moments of this function. Like before, we have

$$\frac{\partial A(\vec{w})}{\partial w_i} = \frac{\partial \log\left(\int h(\vec{x}) \exp[\vec{w}^T \vec{f}(\vec{x})] d\vec{x}\right)}{\partial w_i} = \frac{\int h(\vec{x}) f_i(\vec{x}) \exp[\vec{w}^T \vec{f}(\vec{x})] d\vec{x}}{\int h(\vec{x}) \exp[\vec{w}^T \vec{f}(\vec{x})] d\vec{x}}$$

by moving the partial derivative inside the integral, and we can interpret this as the expectation

$$\int h(\vec{x}) \exp\left[\vec{w}^T \vec{f}(\vec{x}) - A(\vec{w})\right] f_i(\vec{x}) d\vec{x} = \mathbb{E}_{p(\vec{x}|\vec{w})}[f_i],$$

so we again get the **moments of the sufficient statistics via derivatives of the log normalizer** – we find an integral by just calculating a derivative. We can similarly show that the second derivatives give the second-order moments as well:

Proposition 193

We have $(\nabla A)_i = \mathbb{E}[f_i]$ and $(\nabla^2 A(\vec{w}))_{ij} = \left(\text{cov}(\vec{f}(\vec{x}))\right)_{ij}$; in particular, since covariance matrices are positive semidefinite, $A(\vec{w})$ is a convex function of $\vec{w}$.

Doing further calculations shows that we recover higher moments with more derivatives too.

Example 194

Returning to the Gaussian case, we know that $A(\vec{w}) = -\frac{w_1^2}{4w_2} - \frac{1}{2}\log(-2w_2)$, where $w_1 = \frac{\mu}{\sigma^2}$ and $w_2 = \frac{-1}{2\sigma^2}$. If we compute $\frac{\partial A}{\partial w_1}$, we get $-\frac{2w_1}{4w_2} = \mu$, so we indeed get back the first feature $f_1 = x$ (which is the non-central moment). Similarly, taking the other derivative gives the expectation of x^2; from these we can recover μ and σ.

Recall that the parameterization $\vec{f}$ we used for an MRF felt redundant in some way, because our representation encoded realizations of cliques in repeated ways:

Definition 195

An exponential family is **minimal** if the components of the sufficient statistic $\vec{f}$ are linearly independent, in the sense that there is no nonzero vector $\vec{a}$ with $\vec{a} \cdot \vec{f}(\vec{x})$ is **constant** for all $\vec{x}$. If an exponential family is not minimal, we call it **overcomplete**.

If we had such a vector $\vec{a}$, then we would have $p(\vec{x}|\vec{w} + \vec{a}) \propto p(\vec{x}|\vec{w})$ (since we get an $\vec{a}^T \vec{f}(\vec{x})$ term), so we'd be recovering the same distribution from two different vectors of parameters. We don't want that situation from a learning perspective (trying to fit our model without a minimal exponential family would yield multiple equally good solutions), and indeed our previous feature vectors $\vec{f}$ were overcomplete.

Example 196

In addition to the description given in Example 189, we could also write a Bernoulli random variable with distribution $w^x(1-w)^{1-x}$ as $\exp\left(\begin{bmatrix}\log w & \log(1-w)\end{bmatrix} \cdot \begin{bmatrix} x \\ 1-x \end{bmatrix}\right)$, but this would be an overcomplete parameterization because $x + (1-x) = 1$ for all x.

Proposition 197

The log-partition function $A(\vec{w})$ is convex, and **if the family is minimal then it is strictly convex** (meaning there is a single global optimum for the log-likelihood).

And in such a **minimal** exponential family, the expected sufficient statistics, also called the **mean parameters**

$$\vec{\mu} = \mathbb{E}_{p(\vec{x}|\vec{w})}\left[\vec{f}(X)\right]$$

are an equivalent way to parameterize our distribution, meaning that we have a one-to-one mapping between $\vec{w}$ and $\vec{\mu}$. (Intuitively in one dimension, a strictly convex function has A' strictly increasing, meaning the map to the mean parameters is invertible.) And that's why many of our distributions are indeed parameterized in terms of mean parameters (like the mean and variance in a Gaussian – we have $\mathbb{E}[x] = \mu$ and $\mathbb{E}[x^2] = \mu^2 + \sigma^2$). So if we want to compute the "forward mapping" $\vec{\mu} = \nabla A(\vec{w})$, we basically need to do the **inference problem**

$$\frac{\partial A(\vec{w})}{\partial \vec{w}_c(\overline{x}_c)} = \mathbb{E}_{p(\vec{x}|\vec{w})}[1\{\vec{x}_c = \overline{x}_c\}]p(\vec{x}_c = \overline{x}_c|\vec{w}).$$

On the other hand, the "inverse map" ∇A^{-1} corresponds to a **learning problem**! The idea here is that **exponential families play well with KL divergence**, since an expression like $\log \frac{p}{q}$ simplifies nicely. In particular, in the maximum likelihood learning problem where we have an empirical distribution $\tilde{p}(\vec{x})$ and some family of parametric distributions $Q = \{q(\vec{x}|\vec{w}) : \vec{w} \in \Omega)$ that we're trying to fit $\tilde{p}$ to, we want to minimize the KL divergence, which is equivalent to maximizing log-likelihood of the data

$$\text{argmin}_{q \in Q} D(\tilde{p}||q) = \text{argmax}_{\vec{w}} \sum_{\vec{w}} \tilde{p}(\vec{x}) \log q(\vec{x}|\vec{w}).$$

We call this an **M-projection** of our empirical distribution $\tilde{p}$ onto a model family Q. If p is an arbitrary distribution, the M-projection of p onto Q looks like

$$q^* = \text{argmin}_{q \in Q} \sum_{\vec{x}} p(\vec{x}) \log \frac{p(\vec{x})}{q(\vec{x})}, .$$

Theorem 198

Suppose there is some $\vec{w}^*$ with $\mathbb{E}_{q(\vec{x}|\vec{w}^*)}[\vec{f}(\vec{x})] = \mathbb{E}_p[\vec{f}(\vec{x})] = \vec{\mu}$ (that is, we've "matched moments" between p and q for the sufficient statistics). Then $q(\vec{x}|\vec{w}^*)$ is actually the M-projection of p onto Q.

The point is that M-projection preserves marginals, and the best way to approximate the distribution is to choose the one with the same expected sufficient statistics (often marginal probabilities or moments). We've actually seen this before in the language of maximum likelihood learning – evaluating the probability of our data given $\vec{w}$ simplifies to something that only depends on $\sum \vec{f}$, and taking the log yields

$$\log q(\vec{x}_1, \cdots, \vec{x}_N|\vec{w}) = \vec{w}^T \sum_{m=1}^{N} \vec{f}(\vec{x}_m) - NA(\vec{w}) + \text{constant},$$

which is a concave optimization problem. So we can solve it by setting the gradient equal to zero, which indeed gives us the correct moment matching condition. That is, the process of inverting the gradient mapping (going from a given value of $\vec{\mu}$ and trying to find the corresponding $\vec{w}$) is indeed a (maximum likelihood) learning problem – in exponential families "learning and inference are inverses of each other."

Remark 199. *We can think of exponential families in terms of an* **entropy interpretation***: suppose we want a distribution which satisfies $\mathbb{E}_{p(x)}[f_i(x)] = t_i$ for all features f_i, and we also want $p(x)$ to have the* **maximum entropy** *$-\int p(x)\log p(x)dx$. (For example, we might specify a particular mean and variance, but otherwise we want p to have the "most randomness otherwise.") It turns out* **exponential families** *maximize the entropy – the optimal solution is always to pick something with that set of sufficient statistics. This can be proved by Lagrange multipliers using the Lagrangian $L = -\int p(x)\log p(x)dx + \sum_i \theta_i(\mathbb{E}_{p(x)}[f_i(x)] - t_i)$; we find that*

$$0 = \frac{\partial L}{\partial p} = -1 - \log p(x) + \sum_i \theta_i f_i(x),$$

and this is exactly specifying that $p \propto \exp(\vec{\theta}^T \vec{f}(x))$ (where we're given t and we want to find $\vec{w}^*$ such that $(\nabla A)\vec{w}^* = \vec{t}$, so we need to invert ∇A to solve the problem).

23 March 7, 2024

In today's lecture, we'll discuss variational inference – we'll go back to the idea of computing marginal or conditional probabilities in an approximate way, thinking about algorithms that let us solve this computational problem efficiently but not exactly. This will all relate to the idea of exponential families and coming up with a closest tractable approximation to some intractable distribution. (And in next lecture, we'll see that we can justify loopy BP, as well as many other algorithms, in this setting as well.)

Example 200

The setting we'll be considering is the following: we have joint probability distribution $p(x_1, \cdots, x_n)$ represented as a graphical model, and our goal is to compute $p(x_1|e)$ (that is, do marginal inference).

We know that this problem is NP-hard in general, so we don't expect to have a general strategy. But instead, we'll try to approximate the posterior distribution with something simple but as close to $p(x_1|e)$ as possible. Intuitively, this is a reasonable strategy because even though the prior distribution $p(\vec{x})$ can be very complicated, the posterior distribution $p(\vec{x}|\vec{e})$ is often a lot simpler. Two reasons for this are (1) we've made more observations and thus have less uncertainty, and (2) as we've seen in undirected graphical models, we can sometimes remove conditional independence via conditioning. So we'll represent the posterior as an undirected model of the form

$$p(\vec{x}|\vec{e}) = \frac{p(\vec{x}, \vec{e})}{p(\vec{e})} = \frac{\prod_i p(x_i|\text{Pa}(X_i))}{p(\vec{e})} = \frac{1}{Z}\prod_{c \in C} \phi_c(\vec{x}_c),$$

where just like in variable elimination, we clamp the evidence variables to their particular observations $\vec{e}$. We'll then approximate this with some new distribution $q(\vec{x})$ which is easy to do computations on – think of q as a graphical model with low treewidth – and gauge how close we are to the true posterior via the KL divergence $D(p||q) = \sum_{\vec{x}} p(\vec{x}) \log \frac{p(\vec{x})}{q(\vec{x})}$.

Recall that even though this is asymmetric in p and q, $D(p||q) \geq 0$ for all p, q with equality only when $p = q$, so smaller KL divergence means our distributions are more similar in some sense. If we want to do inference with some true distribution $p(\vec{x})$ (we assume we've clamped the $\vec{e}$ variables already so we omit it from the notation), then we may be curious about the difference between $\text{argmin}_q D(p||q)$ (recall this is the **M-projection** of p) and $\text{argmin}_q D(q||p)$ (we call this the **I-projection**. In general, these will be different if we restrict q to only take values over some **restricted** set of distributions; in particular, if our true p is not actually in Q, then we can't exactly represent the posterior with our approximation, and the answers here will be different.

Example 201

Suppose that $p(\vec{x})$ is a 2-dimensional Gaussian with non-diagonal covariance, which means we can represent its general shape (its level curves) by a skewed ellipse in the xy-plane. Meanwhile, let Q be the set of Gaussians with **diagonal** covariance matrices. (This means our level curves will be ellipses aligned with the axes, and in Q the two variables are uncorrelated.)

In words, we can choose our center and spread arbitrarily, and our goal is to find the choice q that minimizes $D(p||q)$ – that would give us the M-projection of p onto Q. It turns out the answer is to choose the mean vector of Q to be the same as that of p and then spread out the variance somewhat so that q "covers" p. **On the other hand**, if we do the reverse KL divergence and try to minimize $D(q||p)$ to get the I-projection, we again choose the correct mean but now choose the variance so that p "covers" q instead, which in general means we get a much more concentrated Gaussian. The point is that when we optimize for $D(p||q)$, we weight the log factor by p, so we need to make the distributions match as much as possible along p. But when we optimize for $D(q||p)$, we weight by q instead, so we end up getting more "mode-seeking behavior."

Example 202

Next suppose $p(\vec{x})$ is a **mixture** of two 2-dimensional Gaussians, and Q is a set of all **single** Gaussians (now allowing arbitrary covariance matrices). If we try to find the M-projection of the mixture p onto Q, we get a Gaussian which puts the mean of the Gaussian between the two peaks of p, with covariance chosen to "support" p. On the other hand, the I-projection will essentially match one of the two Gaussians while ignoring the other one – there are two equivalently good solutions, but both of them are "overconfident" about where the mass goes.

Put another way, M-projection specifies that if $p(x) > 0$, then we should choose $q(x) > 0$, while I-projection specifies the reverse condition. So both choices make sense but there are tradeoffs, and the question now turns to how we can choose a class of distributions Q that are simple. The simplest choice is called the **mean field** approximation, where we assume q fully factorizes as a product of marginals

$$q(X_1, \cdots, X_n) = q_1(X_1)q_2(X_2)\cdots q_n(X_n).$$

(This naming comes from statistical physics, where we use this kind of approach to understand behavior of systems with high treewidth.) So if Q is the family of all such fully-factored distributions on n variables (for example when $n = 2$ and our variables are binary, this is the set of all distributions of the form $q_\phi(x_1, x_2) = \phi_1^{x_1}(1-\phi_1)^{1-x_1}\phi_2^{x_2}(1-\phi_2)^{1-x_2}))$, we can now try to find the M-projection or the I-projection of p onto q. Recall that we can write the KL divergence as a difference of two pieces

$$D(p||q) = \sum_{\vec{x}} p(\vec{x}) \log p(\vec{x}) - \sum_{\vec{x}} p(\vec{x}) \log q(\vec{x}),$$

and now we can rewrite using the simple factored form by adding and subtracting some terms to get

$$\begin{aligned} D(p||q) &= \sum_{\vec{x}} p(\vec{x}) \log \frac{p(\vec{x})}{\prod_i p(x_i)} + \sum_{\vec{x}} p(\vec{x}) \log \prod_i \frac{p(x_i)}{q_i(x_i)} \\ &= \sum_{\vec{x}} p(\vec{x}) \log \frac{p(\vec{x})}{\prod_i p(x_i)} + \sum_{\vec{x}} p(\vec{x}) \sum_i \log \frac{p(x_i)}{q_i(x_i)}. \end{aligned}$$

If we let $q_M(x) = \prod_i p(x_i)$, then this means we actually have

$$D(p||q) = D(p||q_M) + \sum_i D(p(x_i)||q_i(x_i)),$$

and in particular because KL divergence is always positive this is bounded from below by $D(p||q_M)$. Therefore, the optimal solution from the class Q is actually q_M, **where we factor using the true marginals from p.** So this recovers the fact from last lecture that M-projection is really just matching moments – a fully factored distribution is a special case of an exponential family where we just have isolated cliques, so the vector of sufficient statistics is just a bunch of indicators for each variable. And matching the moments of those statistics is then exactly recovering the probabilities of each individual variable taking on particular values.

Unfortunately, this shows that M-projection would require us to find expectations with respect to p in general, and thus this is too expensive computational. So in practice, **variational inference algorithms instead use the I-projection**, since we then only need to do expectations over q.

Example 203

Turning now to I-projections, we let $p(\vec{x}|\vec{e}) = \exp\left(\sum_{c\in C} \theta_c(\vec{x}_c) - \ln Z(\theta)\right)$ be some unnormalized distribution that we want to approximate. We know that $Z(\theta) = p(\vec{e})$ is generally intractable, but this won't be a problem when we try to do the I-projection.

Specifically, we can compute

$$D(q||p) = \sum_{\vec{x}} q(\vec{x}) \log \frac{q(\vec{x})}{p(\vec{x})} = -\sum_{\vec{x}} q(\vec{x}) \log p(\vec{x}) + \sum_{\vec{x}} q(\vec{x}) \log q(\vec{x}).$$

Plugging in our expression for p from above, this simplifies to

$$\begin{aligned} D(q||p) &= -\sum_{\vec{x}} q(\vec{x}) \left(\sum_{c\in C} \theta_c(\vec{x}_c) - \ln Z(\theta)\right) - H(q(\vec{x}) \\ &= -\sum_{c\in C}\sum_{\vec{x}} q(\vec{x})\theta_c(\vec{x}_c) + \sum_{\vec{x}} q(\vec{x}) \ln Z(\theta) - H(q(\vec{x})), \end{aligned}$$

where $H(q)$ is the entropy of the distribution q. But now the log partition function doesn't depend on q, so the second term simplifies to just $\ln Z$, and the first term can also be rewritten more compactly:

$$D(q||p) = -\sum_{c\in C} \mathbb{E}_q[\theta_c(\vec{x}_c)] + \ln Z(\theta) - H(q(\vec{x})).$$

We know that this whole expression is always nonnegative by definition of KL divergence, but that means we can rearrange to find

$$\ln Z(\theta) \geq \sum_{c\in C} \mathbb{E}_q[\theta_c(\vec{x}_c)] + H(q(\vec{x})).$$

The point now is that on the right-hand side, we have unnormalized log-probabilities, so as long as q is simple we can evaluate that tractably. And for any such choice of approximating distribution q, we get a lower bound on the log-partition function, so if we are doing this process on a BN, this gives us a bound on the probability $p(\vec{e})$. We also know that if Q is sufficiently flexible so that p is in Q, then we can have $D(q||p) = 0$ and thus the inequality is actually an equality – the closer q is to p, the better our estimate of the log partition function. So now our game in variational inference is that **if q can range over all distributions**

$$\ln Z(\theta) = \max_q \sum_{c\in C} \mathbb{E}_q[\theta_c(\vec{x}_c)] + H(q(\vec{x})),$$

and now we are doing an optimization problem over some set of functions, hence the name "variational." Interpreting this equation, we want to find a distribution q with high entropy and also high p-probability in expectation over q. (In machine learning and other disciplines, this is a good strategy in general – we have tricks for solving optimization problems and ways to get bounds.) The main trouble is that q may take exponentially many parameters to specify, so we do need to adequately restrict to simpler families Q.

Example 204

One reasonable choice is to choose the value of $\vec{x}^*$ that maximizes the unnormalized probability, yielding the most likely state under p:

$$\vec{x}^* = \text{argmax}_{\vec{x}} \sum_{c\in C} \theta_c(\vec{x}_c).$$

If we then choose q to put all the probability mass on $\vec{x}^*$, then our entropy $H(q(\vec{x}))$ is zero, and the bound we get is $\ln Z(\theta) \ge \sum_{c\in C} \theta_c(\vec{x}^*_c)$.

Remembering that the partition function is a sum of exponentials of terms that look like the right-hand side, and here we're just choosing the largest such term. So if p is sharply peaked around $\vec{x}^*$, then this is a good approximation, but if p is relatively flat (near uniform), it will be quite poor. In fact, the first term in the optimization problem does not vary much, and we instead want to choose q to be flatter to maximize the entropy term instead. So in either case, this optimization problem does intuitively try to make q look similar to p.

So now we need to know how we can actually solve the variational inference problem we've set up. Today we'll discuss the **mean-field** solution – we first define some parameterization of the functions we're optimizing over, and in the mean-field case we consider distributions of the form $q(\vec{x}) = \prod_{i\in V} q_i(x_i)$. Plugging this into the objective function, we can decompose the joint entropy $H(q)$ into sums of local entropies

$$H(q) = -\sum_{\vec{x}} q(\vec{x}) \sum_{i\in V} \ln q_i(\vec{x}_i) = -\sum_{i\in V}\sum_{\vec{x}} q(\vec{x}) \ln q_i(x_i) = -\sum_{i\in V} q_i(x_i)\ln q_i(x_i) \sum_{\vec{x}_{V\setminus i}} q(\vec{x}_{V\setminus i}|x_i) = \sum_{i\in V} H(q_i),$$

(the blue part just sums to 1), and the $q(\vec{x}_c)$ factors also factor into terms from individual variables. So our variational objective is now

$$\boxed{\max_{q\in Q} \sum_{c\in C}\sum_{\vec{x}_c} \theta_c(\vec{x}_c) \prod_{i\in c} q_i(x_i) + \sum_{i\in V} H(q_i)},$$

subject to the constraints that $q_i(x_i) \ge 0$ for all i, x_i and $\sum_{x_i} q_i(x_i) = 1$. And again, any choice of q will give us a lower bound on the true partition function $\ln Z(\theta)$.

Fact 205

In statistical physics and other models, our true probability distribution $p(\vec{x};\theta)$ is often a pairwise MRF (for example imagine a graph on a lattice grid with energy terms coming from edge interactions). Then the mean-field approximation calculates the marginals that increases the objective function above as much as possible using a fully factored q, which means we are trying to solve

$$\max_q \sum_{ij\in E}\sum_{x_i,x_j} \theta_{ij}(x_i,x_j) q_i(x_i) q_j(x_j) - \sum_{i\in V}\sum_{x_i} q_i(x_i)\ln q_i(x_i).$$

This means we are trying to choose configurations where x_i, x_j adjacent have distributions that align well under the θ_{ij} table, while also spreading out the distribution values to increase entropy. Unfortunately, this is a nonconcave optimization problem with many local maxima, meaning it's not obvious how to solve the problem globally. But we can greedily maximize using **block coordinate ascent**. In this approach, we initialize the qs in some arbitrary way, and then we go through each variable and try to find the optimal choice for that variable while everything else is fixed – this can be done efficiently and in closed form. Specifically, we just construct the Lagrangian, take a derivative and set it to zero, and calculate that (if we consider the variable X_i and fix everything else)

$$q(x_i) = \frac{1}{Z_i} \exp\left(\sum_{j \in N(i)} \sum_{x_j} q_j(x_j)\theta_{ij}(x_i, x_j)\right).$$

We may notice that this is essentially the same update as in the Gibbs sampling algorithm, except now we specify the whole distribution rather than a particular instantiation! (We're basically computing the conditional probabilities given the Markov blanket, averaging over the neighbor's value.) Then similar to loopy belief propagation, we do a bunch of updates until we converge, and that will give us a reasonable guess for the answer to the optimization problem. (The picture to have in mind is that we're in a high-dimensional space, but we maximize the function along one direction at a time.)

At each step, our objective will keep improving – we keep iterating to get the distributions to be consistent with each other – and we're guaranteed to converge to some local maximum. This final value then gives us both an approximating distribution Q (in the form of the approximate marginals $Q(x_i)$) and a lower bound on $\ln Z$, and we can do this multiple times from different starting points. But there's no guarantee of the quality of our final estimate in general – to improve mean-field, we can assume a more structured approximation like a spanning tree, which gives us a bigger space of Q distributions to work with.

24 March 12, 2024

Today's lecture is taught by Charlie Marx. We'll continue discussing variational inference today, analyzing some different strategies and focusing specifically on the geometric perspective. The point is to cast loopy belief propagation as a variational inference problem to understand why it works so well in practice.

Remark 206. *This week's material won't be on the final exam, but it's meant to be a nice way to connect concepts across the course.*

Recall that the setup we have is as follows: we have a difficult distribution $p(\vec{x}|\vec{e})$ to work with, and we want to approximate it with some "close" distribution $q(\vec{x})$ where computation is simple and $p(\vec{x}|\vec{e})$ is close to $q(\vec{x})$. Last time, we mentioned that we measure our distance between distributions using the KL-divergence, but there's a difference between minimizing $D(p||q)$ (the M-projection) and $D(q||p)$ (the I-projection), and in particular the optimal q will be different in the two cases. We use the I-projection for computational reasons (since computing expectations with respect to q is easier), and we stated last time that one way to cast our problem in this setting is to assume p is a member of an exponential family (like in an MRF) and then expand out the definitions to rewrite

$$D(q||p) = -\vec{\theta}^T \sum_{\vec{x}} q(\vec{x})\vec{f}(\vec{x}) + \ln Z(\vec{\theta}) - H(q(\vec{x})) \geq 0,$$

meaning that we get a variational lower bound

$$\ln Z(\vec{\theta}) \geq \vec{\theta}^T \sum_{\vec{x}} q(\vec{x})\vec{f}(\vec{x}) + H(q(\vec{x}))$$

for **any** approximating distribution q. So our goal is to find the approximating distribution over some family which maximizes this right-hand side (since if we plug in p for q, we get the exact value of $\ln Z(\vec{\theta})$). Unfortunately, doing this is hard in general (we haven't made any simplifications), so we must choose a the family of distributions q that can be represented compactly and we need a way to find the maximum efficiently.

Last time, we talked about one such strategy, which is the **mean-field approximation**. Specifically, we assumed q is fully factored as a product of marginals (this is a very strong assumption), which is an extremely strong assumption. Today's goal is instead to approximate in a less strong way while still maintaining tractability, and in particular we'll see that loopy BP is an example of this. We'll let $\vec{f}(\vec{x})$ be the sufficient statistic vector, so that

$$\ln Z(\vec{\theta}) = \max_q \vec{\theta}^T \sum_{\vec{x}} q(\vec{x})\vec{f}(\vec{x}) + H(q(\vec{x})).$$

We can let $\mu_q = \mathbb{E}_q[\vec{f}(\vec{x})]$ be our marginals (we call these the **mean parameters**), and now instead of optimizing over all **distributions** q, we can optimize over **marginal vectors** $\vec{\mu}$, writing

$$\ln Z(\vec{\theta}) = \max_{\mu \in M} \max_{q : \mathbb{E}_q[\vec{f}(\vec{x})] = \mu} \vec{\theta}^T \mu + H(q(\vec{x})).$$

So the idea is that within each μ in the **marginal polytope** M, there could be multiple distributions q_1, q_2 that achieve this μ, and we'll pick the one with the highest entropy among them.

Example 207

Suppose we have binary random variables. Then the marginal polytope takes the form

$$M = \left\{ \vec{\mu} : \vec{\mu} = \sum_x q(\vec{x})\vec{f}(\vec{x}) \text{ for some } q(x) \geq 0, \sum_x q(x) = 1 \right\},$$

which is the **convex hull of the points** $\vec{f}(\vec{x})$ for each $\vec{x} \in \{0,1\}^n$.

The idea is that we have a bunch of vectors $\vec{\mu}$ corresponding to **specific instantiations** of 1s and 0s for our variables, and then we can take any convex combination of such vectors. And now rewriting our variational problem again, we have

$$\ln Z(\vec{\theta}) = \max_{\vec{\mu} \in M} \vec{\theta}^T \vec{\mu} + H(\vec{\mu}),$$

where $H(\vec{\mu})$ is defined to be the maximum entropy over all distributions with the correct mean parameters.

We've **still not done any approximations**, so our problem is still hard – indeed, the marginal polytope M is still a convex hull over possibly exponentially many vertices and facets. And even if we could pick a $\vec{\mu}$ in this set, we still have the problem that $H(\vec{\mu})$ can be very difficult to compute (it requires inference), so in fact both terms are hard. To solve these two problems, we make two approximations:

1. Replace M with some **relaxation** of the marginal polytope – for example we can only consider the local consistency constraints M_L.
2. Replace $H(\vec{\mu})$ with some other easier-to-compute function $\tilde{H}(\vec{\mu})$ (we'll be more specific soon).

Example 208

Returning to the pairwise MRF example as in Fact 205, our goal is to maximize

$$\ln Z(\vec{\theta}) = \max_{\vec{\mu}\in M} \sum_{i\in V}\sum_{x_i} \theta_i(x_i)\mu_i(x_i) \sum_{ij\in E}\sum_{x_i,x_j} \theta_{ij}(x_i,x_j)\mu_{ij}(x_i,x_j) + H(\vec{\mu}),$$

and our goal is to simplify the constraint $\vec{\mu} \in M$.

We know some of our constraints for being in the marginal polytope: we must satisfy $\mu_i(x_i) \geq 0$ and $\sum_{x_i} \mu(x_i) = 1$ for each marginal, and furthermore along each edge $\mu_{ij}(x_i, x_j) \geq 0$ and $\sum_{x_i,x_j} \mu ij(x_i, x_j) = 1$. We must also enforce how the local assignments are locally consistent – specifically, marginalizing $\mu_{ij}(x_i, x_j)$ must match with the marginals $\mu_i(x_i)$ and $\mu_j(x_j)$ if we marginalize over x_j, x_i, respectively. We **should** enforce other – for example, there may be a chain of implications across the graph – but there are so many such interactions that this is what gives us the exponentially-faceted shape, and our simplification is to **ignore that and only consider the local consistency polytope** defined by the constraints above.

This is a strictly larger set than M, so if $\vec{\mu} \in M$ then $\vec{\mu} \in M_L$ as well. **In the case where** p is tree-structured, it turns out it's sufficient to assume local consistency everywhere – this is related to why calibration of tree-shaped graphs and variable elimination are simpler on trees. The point is that M_L can be defined compactly with relatively few constraints, and thus we make the approximation

$$\ln Z(\vec{\theta}) \approx \max_{\vec{\mu}\in M_L} \vec{\theta}^T \vec{\mu} + H(\vec{\mu}).$$

For trees, this is all easy to compute – max entropy and optimizing over the set of constraints are both easy. For a bit of intuition for why entropy is actually easy to compute, recall that exponential families are exactly maximizing entropy given some sufficient statistics – there's a relation between the gradient of the log partition function and the sufficient statistics. So if we have some set of valid marginal distributions M and take some particular $\vec{\mu} \in M$, there are multiple distributions q_1, q_2 that map to that marginal vctor, but the one with the highest entropy will be from the exponential family. So it suffices to just search over the exponential family, which is an MLE learning problem – we've actually done this before with the Chow-Liu algorithm. Indeed, our entropy for trees decomposes as

$$H(\vec{\mu}) = \sum_{i\in V} H(\mu_i) - \sum_{ij\in T} I(\mu_{ij}),$$

where H is the entropy and I the mutual information (this is the same formula as for Chow-Liu, but now using $\vec{\mu}$ instead of the empirical marginals). The key step now is to also use the **Bethe entropy approximation** on a **general** graph (not necessarily a tree) and approximate

$$H(\vec{\mu}) \approx H_{\text{Bethe}}(\vec{\mu}) = \sum_{i\in V} H(\mu_i) - \sum_{ij\in E} I(\mu_{ij})$$

(note that this can be an overestimate or underestimate). We thus have an easier variational approximation problem of the form

$$\max_{\vec{\mu}\in M_L} \sum_{c\in C}\sum_{\vec{x}_c} \theta_c(\vec{x}_c)\mu_c(\vec{x}_c) + H_{\text{Bethe}}(\vec{\mu}).$$

This is a non-concave function in general, so it's hard to maximize, but **if loopy BP converges it will approach some extremum or saddle point**. In general, it turns out there is an exact correspondence between fixed points of the loopy BP algorithm and stationary points of the Bethe variational objective.

Fact 209

If we do Lagrange multipliers for our approximation for $\ln Z(\vec{\theta})$ (using the Bethe free energy approximation and our local consistency constraint polytope M_L), we actually exactly recover the loopy BP updates.

We can now compare this to the naive mean-field approximation – remember that there we **imposed** a strong constraint on our factorization, adding more conditions for $\vec{\mu}$. So in that case, we actually optimize and get an **inner bound** on the true marginal polytope. What's nice is that in mean-field we actually get a **lower bound** on the partition function, since we're optimizing over a smaller subset of the allowed distributions M. Then it seems that the relaxation with local consistency constraints should give an upper bound on the partition function, but unfortunately we need to make the Bethe entropy approximation and thus can't say that in general.

Fact 210

However, we **do** get an upper bound if we have some upper bound $\tilde{H}(\vec{\mu})$ on our true entropy $H(\vec{\mu})$. Then we **would** get an upper bound

$$\ln Z(\vec{\theta}) \le \max_{\vec{\mu} \in M_L} \sum_{c \in C} \sum_{\vec{x}_c} \theta_c(\vec{x}_c)\mu_c(\vec{x}_c) + \tilde{H}(\vec{\mu}).$$

One such example is called the **tree-weighted approximation** – this is a concave relaxation where we specify the distribution over spanning trees of the graph.

So the point overall is that we can get both a lower and an upper bound on our true value of $\ln Z(\vec{\theta})$ using our different variational approaches – in both cases we cast inference as an optimization problem, and then we use certain tools that already exist in the optimization world that make the problem more tractable. People are still interested in improving these techniques – it's an active research area to find more accurate approximations in alternative ways or find stable algorithms for solving the optimization problems.

25 March 14, 2024

We'll revisit the EM algorithm today, seeing how to combine EM ideas with variational inference and connect the dots towards training deep generative models with latent variables.

Example 211

As usual, we're in a setting where we have some joint distribution $p(\vec{X}, \vec{Z}; \theta)$, where the random variables $\vec{X}$ are observed but $\vec{Z}$ are not (for example a mixture of Gaussians where Z is a categorical variable representing the component). But this problem can still be approached with maximum likelihood learning if we "marginalize over the unknown components," integrating out over uncertainty by looking at all completions of the data.

Specifically, we try to maximize the loglikelihood $\sum_{\vec{x} \in \mathcal{D}} \log p(\vec{x}; \theta)$, and the challenge is that in our latent variable model we instead have

$$\ell(\theta; \mathcal{D}) = \sum_{\vec{x} \in \mathcal{D}} \log \sum_{\vec{z}} p(\vec{x}, \vec{z}; \theta).$$

The fact that we have a log of a sum then makes things more complicated (we no longer have a closed-form decomposition even if p is just a Bayesian network), and it in particular requires us to evaluate the posterior $p(\vec{z}|\vec{x}; \theta)$, which can be expensive. But today we'll see a **variational formulation** of how the EM algorithm works, which will allow us to generalize the ideas we previously saw.

Specifically, let $q(\vec{z})$ now be some probability distribution over the hidden variables $\vec{Z}$. Then the KL-divergence (here we want to take expectation with respect to q, which makes the calculations more tractable) is

$$\begin{aligned} D\left(q(\vec{z})||p(\vec{z}|\vec{x};\theta)\right) &= \sum_{\vec{z}} q(\vec{z}) \log \frac{q(\vec{z})}{p(\vec{z}|\vec{x};\theta)} \\ &= -\sum_{\vec{z}} q(\vec{z}) \log p(\vec{z}|\vec{x};\theta) - H(q) \\ &= -\sum_{\vec{z}} q(\vec{z}) \log \frac{p(\vec{z},\vec{x};\theta)}{p(\vec{x};\theta)} - H(q) \end{aligned}$$

(this form of the expression should look similar to the derivations we've been doing recently), and now since the denominator of the log doesn't depend on z, we can write out

$$\begin{aligned} D\left(q(\vec{z})||p(\vec{z}|\vec{x};\theta)\right) &= -\sum_{\vec{z}} q(\vec{z}) \log p(\vec{z},\vec{x};\theta) + \sum_{\vec{z}} q(\vec{z}) \log p(\vec{x};\theta) - H(q) \\ &= -\sum_{\vec{z}} q(\vec{z}) \log p(\vec{z},\vec{x};\theta) + \log p(\vec{x};\theta) - H(q), \end{aligned}$$

where the red term is the log partition function. So now because KL-divergence is nonnegative, for **any** q we have the variational bound

$$\boxed{\log p(\vec{x};\theta) \geq \sum_{\vec{z}} q(\vec{z}) \log p(\vec{z},\vec{x};\theta) + H(q)}.$$

Like before, if the posterior is concentrated on a particular value for $\vec{z}$, we would want to make q concentrated on that as well to maximize the first term on the right. But if the posterior is spread out, we'd want to spread out q to maximize the entropy term again. And if q is **exactly** the posterior, then the KL divergence is exactly zero, so this lower bound becomes an equality instead (and thus we can indeed hope to get near the true value).

Turning now to a derivation of EM, another way to rewrite the equation we have above is

$$H(q) + \sum_{\vec{z}} q(\vec{z}) \log p(\vec{z},\vec{x};\theta) = \log p(\vec{x};\theta) - D(q||p(\vec{z}|\vec{x};\theta).$$

Proposition 212

Call the quantity above $F[q,\theta]$ (it depends on both the parameters θ and the distribution q). Then the EM algorithm is equivalent to performing coordinate ascent on q and θ.

Indeed, the algorithm has us perform alternating E steps and M steps, and our E steps involve computing

$$q^{(t)} = \mathrm{argmax}_{\mathrm{q}} F[q,\theta^{(t)}]$$

(since if we look at the right-hand definition of F, we're trying to minimize the KL divergence), while our M steps involve computing

$$\theta^{(t+1)} = \mathrm{argmax}_{\theta} F[q^{(t)},\theta]$$

(since now if we look at the left-hand definition of F, we are choosing the parameters that optimize the log-likelihood of a fully-observed data point). So we alternately optimize the function F while keeping θ or q fixed, and that's exactly doing ascent on alternating axes. And this means the log-likelihood of the data (that is, $p(\vec{x};\theta^{(t+1)} = F[q^{(t+1)}, q^{(t+1)}])$ is always at least as big as it was at the previous step (namely $\log p(\vec{x};\theta^{(t)}) = F[q^{(t)},\theta^{(t)}]$).

Remark 213. *We've made this kind of argument from a different perspective already – each choice of q gives us a lower bound for the true log-likelihood, but we get a tight lower bound at our current value $\theta^{(t)}$ at our E-step. Then in the M-step, we maximize our estimate instead, which can also only increase the true value at $\theta^{(t+1)}$.*

But we know that inference in general is hard, and computing posteriors in the E-step can be intractable. So based on this intuition that we're optimizing over q and θ separately, we can define a set of tractable probability distributions $q(\vec{z}|\vec{x}, \phi)$ that we can actually work with and still use the same strategy. (For example, we may consider the set of fully factored probability distributions and make the mean-field approximation.) Then the same machinery goes through – it's still true that we have the **evidence lower bound** (we often call the right hand side an **ELBO** for short)

$$\log p(\vec{x}; \theta) \geq \sum_{\vec{z}} q(\vec{z}|\vec{x}, \phi) \log p(\vec{z}, \vec{x}; \theta) + H(q(\vec{z}|\vec{x}; \phi)),$$

just that we can no longer necessarily reach equality (because there may not be any q that is very similar to the posterior). Call the right-hand side $\mathcal{L}(\vec{x}; \theta, \phi)$ – we can then use that to approximate the log-likelihood

$$\ell(\theta; \mathcal{D}) = \sum_{\vec{x}^{(i)} \in \mathcal{D}} \log p(\vec{x}^{(i)}; \theta) \geq \sum_{\vec{x}^{(i)}} \mathcal{L}(\vec{x}^{(i)}; \theta, \phi^{(i)}),$$

where crucially we use a **different** set of variational parameters $\phi^{(i)}$ for **every data point**, because different data points will require different ϕs to match the posterior $q(\vec{z}|\vec{x}; \theta)$. And this inequality goes in a good direction – if we want to make the left-hand side large, it's good to make the right-hand side large. We have an incentive both to choose θ so that the true log-likelihood is large **and** to choose ϕ so that the simple q matches the posterior quite well.

Remark 214. *Upper bounding KL divergence is very difficult in general – there are bounds, but they tend to be very loose. So it does mean that we never know exactly how close we are to the true distribution.*

So the quantity we now want to maximize (which gives us a lower bound for $\max_\theta \ell(\theta; \mathcal{D})$) is

$$\max_{\theta, \phi^{(1)}, \cdots, \phi^{(M)}} \sum_{\vec{x}^{(i)} \in \mathcal{D}} \mathcal{L}(\vec{x}^{(i)}, \theta, \phi^{(i)}).$$

In the **variational EM** algorithm, each $q_{\phi^{(i)}} = q(\vec{z}|\vec{x}^{(i)}, \phi^{(i)})$ is some distribution over $\vec{z}$, and we can rewrite the previous equation as

$$H(q_{\phi^{(i)}}) + \sum_{\vec{z}} q_{\phi^{(i)}}(\vec{z}) \log p(\vec{z}, \vec{x}^{(i)}; \theta) = \log p(\vec{x}^{(i)}; \theta) - D(q_{\phi^{(i)}} || p(\vec{z}|\vec{x}^{(i)}; \theta)).$$

Calling this quantity $F^{(i)}[\phi^{(i)}, \theta]$ and summing across all data points, we then can view variational EM as coordinate assent on $F[\phi^{(1)}, \cdots, \phi^{(M)}, \theta]$. (Note that $\phi^{(i)}$ and $\vec{x}^{(i)}$ here denote the ith **data point**, not the ith iteration of the algorithm.) At each iteration t, our "E step" involves maximizing $F^{(i)}[\phi^{(i)}, \theta^{(t)}]$ over all possible $\phi^{(i)}$s, and then in the "M step" we choose the θ which maximizes $\sum_i F^{(i)}[\phi^{(i)}, \theta]$ for our current values of ϕ. But the key point is that **here the marginal likelihood might decrease because our bounds are not tight** – since we're not weighting with respect to the true posterior, we're not guaranteed to reach a local maximum.

Example 215

This is the type of machinery that is actually used in deep generative models – posteriors can be very intractable to compute in situations where we sample a latent variable z (like a standard normal) and then $p(x|z)$ is very complicated (like a normal where the mean and covariance are neural networks). For such models, we may have a dataset $\mathcal{D}$ where z variables are latent (not observed), and we may want to maximize the likelihood of that dataset with respect to θ without inverting the neural network.

The difference now is that we cannot use a different variational parameter $\phi^{(i)}$ for each data point $\vec{x}^{(i)}$, since that's not scalable for large datasets. Instead, we'll specify the variational parameters via a single parametric function f_ϕ, mapping data points $\vec{x}$ to good variational parameters (think of this as a regression function or another neural network). We then get an approximated posterior $q_\phi(\vec{z}|\vec{x})$ where the ϕs are determined in a shared way across all data points, and then our goal is to solve the more tractable **variational EM optimization problem**

$$\max_\theta \ell(\theta; \mathcal{D}) \geq \max_{\theta,\phi} \sum_{\vec{x}^{(i)} \in \mathcal{D}} \mathcal{L}(\vec{x}^{(i)}, \theta, \phi).$$

To do this problem, we'll no longer alternatingly optimize θ and ϕ – we can now optimize this function via gradient ascent. Looking at the expression for the ELBO, we have

$$\mathcal{L}(\vec{x}; \theta, \phi) = \sum_{\vec{z}} q_\phi(\vec{z}|\vec{x}) \log p(\vec{z}, \vec{x}; \theta) + H(q_\phi(\vec{z}|\vec{x})) = \mathbb{E}_{q_\phi(\vec{z}|\vec{x})}\left[\log p(\vec{z}, \vec{x}; \theta) - \log q_\phi(\vec{z}|\vec{x})\right].$$

We can then initialize θ, ϕ, repeatedly sample data points from $\mathcal{D}$, and update θ and ϕ in the gradient direction. (So we're trying to adjust our ϕ so our probability mass is on $\vec{z}$s likely under the joint distribution, while still capturing posterior.)

The question is then how to compute gradients – we now have distributions that are complicated, so we **use Monte Carlo sampling to estimate the gradients instead**. Let $\vec{z}^{(1)}, \cdots, \vec{z}^{(K)}$ be samples from $q_\phi(\vec{z}|\vec{x})$, and then we can approximate

$$\mathbb{E}_{q_\phi(\vec{z}|\vec{x})}\left[\log p(\vec{z}, \vec{x}; \theta) - \log q_\phi(\vec{z}|\vec{x})\right] \approx \frac{1}{K} \sum_{k=1}^{K} \log p(\vec{z}^{(k)}, \vec{x}; \theta) - \log q_\phi(\vec{z}^{(k)}|\vec{x}).$$

By doing this, we don't need to worry about the whole distribution of $\vec{z}$s, and we can now use this to compute gradients. The θ one is easy because we just have the term corresponding to fully observed data – we have

$$\nabla_\theta \mathbb{E}_{q_\phi(\vec{z}|\vec{x})}\left[\log p(\vec{z}, \vec{x}; \theta) - \log q_\phi(\vec{z}|\vec{x})\right] \approx \frac{1}{K} \sum_k \nabla_\theta \log p(\vec{z}^{(k)}, \vec{x}; \theta).$$

The trickier one is the ϕ-gradient, since the expectation we're taking itself depends on ϕ. We'd still like to estimate with a Monte Carlo average, but we'd have to understand how that sampling changes along with ϕ. The idea here is that we can use the REINFORCE idea from reinforcement learning – we choose some set of possible actions with various rewards $r(\vec{z})$, and we then set up a **randomized policy** for choosing our actions parameterized by ϕ. Then if we want to compute a gradient of the expected reward $\mathbb{E}_{q_\phi(\vec{z})}[r(\vec{z})]$, and our policy $q_\phi(\vec{z})$ can be written as $\exp(\vec{f}(\vec{z}) \cdot \phi)$ for some feature vector $\vec{f}$, we can use the importance sampling trick and write

$$\begin{aligned} \frac{\partial}{\partial \phi_i} \mathbb{E}_{q_\phi(\vec{z})}[r(\vec{z})] &= \sum_{\vec{z}} q_\phi(\vec{z}) \frac{\partial \log q_\phi(\vec{z})}{\partial \phi_i} r(\vec{z}) \\ &= \mathbb{E}_{q_\phi(\vec{z})}\left[\frac{\partial \log q_\phi(\vec{z})}{\partial \phi_i} r(\vec{z})\right]. \end{aligned}$$

So we can actually rewrite this gradient itself as an expectation, meaning we can approximate with samples from $q_\phi(\vec{z})$ (yielding an unbiased estimator of the true value). Applying this to the ELBO (scoring with the reward function $\log p(\vec{z}, \vec{x}; \theta) - \log q_\phi(\vec{z}|\vec{x})$) then allows us to get the ϕ-gradient as well, making the gradient ascent algorithm tractable via Monte Carlo.

Definition 216

Such a structure keeping track of both $p(\vec{x}|\vec{z})$ and $q(\vec{z}|\vec{x})$ is called a **variational autoencoder** – it contains a **decoder** (using the θ parameters which help us determine $\vec{x}$ from $\vec{z}$) and an **encoder** (using the variational parameters ϕ defining the distribution q to describe the posterior). Both sets of parameters are usually encoded by neural networks.

Models trained on large image dataset do indeed fit data quite well. So in particular, latent variable models are used in practice, and the ideas of probabilistic modeling, inference, and sampling do fundamentally show up in lots of different problems. Much of this is still a very active area of research! And if we're interested in learning more about deep generative models, we can take a look at CS 236.