Chris Piech
CS109

Lecture #20
Nov 7th, 2018

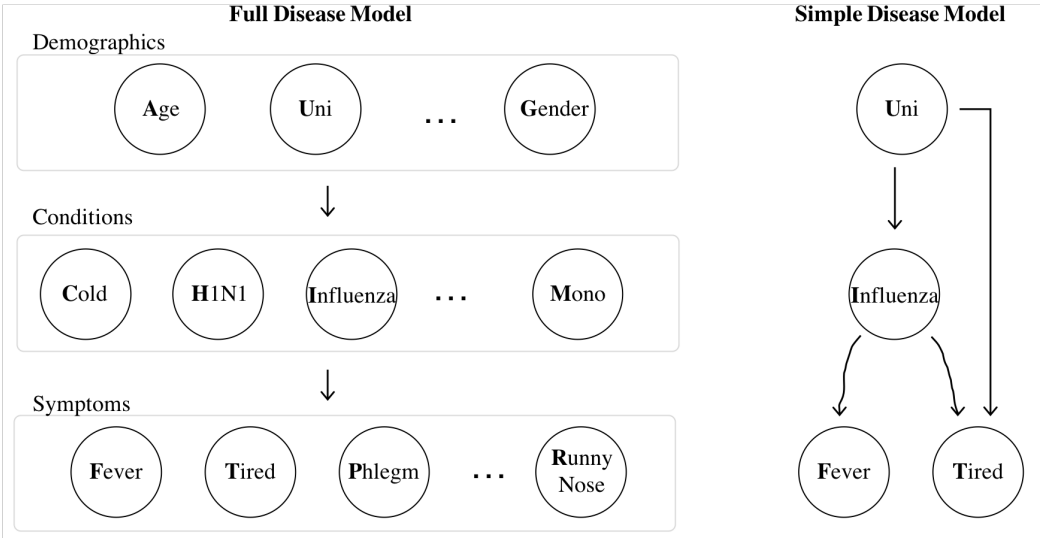
General Inference

At this point in CS109 we have developed tools for analytically solving for probabilities. We can calculate the likelihood of random variables taking on values, even if they are interacting with other random variables (which we have called multi-variate models, or we say the random variables are jointly distributed). We have also started to study samples and sampling.

As a capstone for this part of the class I would like to consider the task of "general inference" in the world of disease prediction. A website, WebMd Symptom Checker, exemplifies our task. They have built a probabilistic model with random variables which roughly fall under three categories: symptoms, risk factors and diseases. For any combination of observed symptoms and risk factors, they can calculate the probability of any disease. For example, they can calculate the probability that I have influenza given that I am a 21-year-old female who has a fever and who is tired: $P(I = 1|A = 21, G = 1, T = 1, F = 1)$. Or they could calculate the probability that I have a cold given that I am a 30-year-old with a runny nose: $P(C = 1|A = 30, R = 1)$. At first blush this might not seem difficult. But as we dig deeper we will realize just how hard it is. There are two challenges: (1) sufficiently specifying the probabilistic model and (2) calculating any desired probability.

1 Bayesian Networks

Before we jump into how to solve probability (aka inference) questions, let's take a moment to go over how an expert doctor could specify the relationship between so many random variables. Ideally we could have our expert sit down and specify the entire "joint distribution" (see the first lecture on multi-variable models). She could do so either by writing a single equation that relates all the variables (which is as impossible as it sounds), or she could come up with a joint distribution table where she specifies the probability of any possible combination of assignments to variables. It turns out that is not feasible either. Why? Imagine there are $N = 100$ binary random variables in our WebMD model. Our expert doctor would have to specify a probability for each of the $2^N > 10^{30}$ combinations of assignments to those variables, which is approaching the number of atoms in the universe. Thankfully, there is a better way. We can simplify our task if we know the "generative" process that creates a joint assignment. Based on the generative process we can make a data structure known as a **Bayesian Network**. Here are two networks of random variables for diseases:



For diseases the flow of influence is directed. The states of "demographic" random variables influence whether someone has particular "conditions", which influence whether someone shows particular "symp-

toms". On the right is a simple model with only four random variables. Though this is a less interesting model it is easier to understand when first learning Bayesian Networks. Being in university (binary) influences whether or not someone has influenza (binary). Having influenza influences whether or not someone has a fever (binary) and the state of university and influenza influences whether or not someone feels tired (also binary).

In a Bayesian Network an arrow from random variable X to random variable Y articulates our assumption that X directly influences the likelihood of Y. We say that X is a *parent* of Y. To fully define the Bayesian network we **must** provide a way to compute the probability of each random variable (X_i) conditioned on knowing the value of all their parents: $P(X_i = k|\text{Parents of} X_i \text{ take on their values})$. Here is a concrete example for the simple disease model. Recall that each of the random variables in the simple model is binary:

$$P(\text{Uni} = 1) = 0.8$$

$$P(\text{Influenza} = 1|\text{Uni} = 1) = 0.2 \qquad P(\text{Fever} = 1|\text{Influenza} = 1) = 0.9$$

$$P(\text{Influenza} = 1|\text{Uni} = 0) = 0.1 \qquad P(\text{Fever} = 1|\text{Influenza} = 0) = 0.05$$

$$P(\text{Tired} = 1|\text{Uni} = 0, \text{Influenza} = 0) = 0.1 \qquad P(\text{Tired} = 1|\text{Uni} = 0, \text{Influenza} = 1) = 0.9$$

$$P(\text{Tired} = 1|\text{Uni} = 1, \text{Influenza} = 0) = 0.8 \qquad P(\text{Tired} = 1|\text{Uni} = 1, \text{Influenza} = 1) = 1.0$$

The reason this is so useful is that the "joint" probability can be expressed as the product of the probabilities of each random variable conditioned on its parents! Without loss of generality, let X_i refer to the ith random variable (such that if X_i is a parent of X_j then $i < j$):

$$P(\text{Joint}) = P(X_1 = k_1, \ldots, X_n = k_n) = \prod_i P(X_i = k_i|\text{Parents of } X_i \text{ take on their values})$$

Let's put this in programming terms. All that we need to do in order to code up a Bayesian network is to define a function: `getProbXi(i, k, parents)` which returns the probability that X_i (the random var with index `i`) takes on the value `k` given a value for each of the parents of X_i encoded by `parents`: $P(X_i = k_i|\text{Parents}(X_i) \text{ take on their values})$

Deeper understanding: Using the chain rule we can decompose the joint probability. To make the following math easier to digest I am going to use k_i as shorthand for the event that $X_i = k_i$:

$$\begin{aligned} P(k_1, \ldots, k_n) &= P(k_n, |k_{n-1} \ldots, k_1)P(k_{n-1}, |k_{n-2} \ldots, k_1) \cdots P(k_2|k_1)P(k_1) && \text{chain rule} \\ &= \prod_i P(k_i|k_{i-1}, \ldots, k_1) && \text{change in notation} \\ &= \prod_i P(k_i|\text{Parents of } X_i \text{ take on their values}) && \text{implied by Bayes Net} \end{aligned}$$

The central assumption made is that $P(k_i|k_{i-1}, \ldots, k_1) = P(k_i|\text{Parents of } X_i \text{ take on their values})$. In other words: each random variable is conditionally independent of its non-descendents, given its parents.

In the next part of CS109 we are going to talk about how we could learn such probabilities from data. For now let's start with the (reasonable) assumption that an expert can write `getProbXi`. We haven't talked about continuous of multinomial random variables in Bayes Nets. None of the theory changes: the expert will just have to define `getProbXi` to handle more values of `k` than 0 or 1.

Great! We have a feasible way to define a large network of random variables. First challenge complete. However a Bayesian network is not very interesting to us unless we can use it to solve different conditional probability questions.

2 General Inference via Joint Sampling

Now we have a reasonable way to specify the joint probability of a network of many random variables. Before we celebrate, realize that we still don't know how to use such a network to answer probability questions.

There are many techniques for doing so. I am going to introduce you to one of the great ideas in probability for computer science: we can use sampling to solve inference questions on Bayesian networks. Sampling is frequently used in practice because it is relatively easy to understand and easy to implement.

As a warmup consider what it would take to sample an assignment to each of the random variables in our Bayes net. Such a sample is often called a "joint sample" or a "particle" (as in a particle of sand). To sample a particle, simply sample a value for each random variable one at a time based on the value of the random variable's parents. This means that if X_i is a parent of X_j, you will have to sample a value for X_i before you sample a value for X_j.

Let's work through an example of sampling a "particle" for the Simple Disease Model in the previous section:

1. Sample from $P(\text{Uni} = 1)$: $\text{Bern}(0.8)$. Sampled value for Uni is 1.
2. Sample from $P(\text{Influenza} = 1|\text{Uni} = 1)$: $\text{Bern}(0.2)$. Sampled value for Influenza is 0.
3. Sample from $P(\text{Fever} = 1|\text{Influenza} = 0)$: $\text{Bern}(0.05)$. Sampled value for Fever is 0.
4. Sample from $P(\text{Tired} = 1|\text{Uni} = 1, \text{Influenza} = 0)$: $\text{Bern}(0.8)$. Sampled value for Tired is 0.

Thus the sampled particle is: [Uni = 1, Influenza = 0, Fever = 0, Tired = 0]. If we were to run the process again we would get a new particle (with likelihood determined by the joint probability).

Now our strategy is simple: we are going to generate N samples where N is in the hundreds of thousands (if not millions). Then we can compute probability queries by counting. Let $N(\mathbf{X} = \mathbf{k})$ be notation for the number of particles where random variables $\mathbf{X}$ take on values $\mathbf{k}$. Recall that the bold notation $\mathbf{X}$ means that $\mathbf{X}$ is a vector with one or more elements. By the "frequentist" definition of probability:

$$P(\mathbf{X} = \mathbf{k}) = \frac{N(\mathbf{X} = \mathbf{k})}{N}$$

Counting for the win! But what about conditional probabilities? Well using the definition of conditional probabilities, we can see it's still some pretty straightforward counting:

$$P(\mathbf{X} = \mathbf{a}|\mathbf{Y} = \mathbf{b}) = \frac{P(\mathbf{X} = \mathbf{a}, \mathbf{Y} = \mathbf{b})}{P(\mathbf{Y} = \mathbf{b})} = \frac{\frac{N(\mathbf{X}=\mathbf{a}, \mathbf{Y}=\mathbf{b})}{N}}{\frac{N(\mathbf{Y}=\mathbf{b})}{N}} = \frac{N(\mathbf{X} = \mathbf{a}, \mathbf{Y} = \mathbf{b})}{N(\mathbf{Y} = \mathbf{b})}$$

Let's take a moment to recognize that this is straight-up fantastic. General inference based on analytic probability (math without samples) is hard even given a Bayesian network (if you don't believe me, try to calculate the probability of flu conditioning on one demographic and one symptom in the Full Disease Model). However if we generate enough samples we can calculate *any* conditional probability question by reducing our samples to the ones that are consistent with the condition (**Y = b**) and then counting how many of those are also consistent with the query (**X = a**). Here is the algorithm in code:

```
N = 10000
// ``query" is the assignment to variables we want probabilities for
// ``condition" is the assignments to variables we will condition on
def getAnyProbability(query, condition):
  particles = generateManyJointSamples(N)
  condParticles = rejectNonConsistentSamples(particles, condition)
  K = countConsistentSamples(condParticles, query)
  return K / len(condParticles)
```

This algorithm is sometimes called "Rejection Sampling" because it works by generating many particles from the joint distribution and rejecting the ones that are not consistent with the set of assignments we are conditioning on. Of course this algorithm is an approximation, though with enough samples it often works out to be a very good approximation. However, in cases where the event we're conditioning on is rare enough that it doesn't occur after millions of samples are generated, our algorithm will not work. The last line of our code will result in a divide by 0 error. See the next section for solutions!

3 General Inference when Conditioning on Rare Events

Joint Sampling is a powerful technique that takes advantage of computational power. But it doesn't always work. In fact it doesn't work any time that the probability of the event we are conditioning is rare enough that we are unlikely to ever produce samples that exactly match the event. The simplest example is with continuous random variables. Consider the Simple Disease Model. Let's change Fever from being a binary variable to being a continuous variable. To do so the only thing we need to do is re-specify the likelihood of fever given assignments to its parents (influenza). Let's say that the likelihoods come from the normal PDF:

$$\text{if Influenza = 0, then Fever} \sim N(\mu = 98.3, \sigma = 0.7) \qquad \therefore f(\text{Fever} = x) = \frac{1}{\sqrt{2\pi \cdot 0.7}} e^{-\frac{(x-98.3)^2}{2 \cdot 0.7}}$$

$$\text{if Influenza = 1, then Fever} \sim N(\mu = 100.0, \sigma = 1.8) \qquad \therefore f(\text{Fever} = x) = \frac{1}{\sqrt{2\pi \cdot 1.8}} e^{-\frac{(x-100.0)^2}{2 \cdot 1.8}}$$

Drawing samples (aka particles) is still straightforward. We apply the same process until we get to the step where we sample a value for the Fever random variable (in the example from the previous section that was step 3). If we had sampled a 0 for influenza we draw a value for fever from the normal for healthy adults (which has $\mu = 98.3$). If we had sampled a 1 for influenza we draw a value for fever from the normal for adults with the flu (which has $\mu = 100.0$). The problem comes in the "rejection" stage of joint sampling.

When we sample values for fever we get numbers with infinite precision (eg 100.819238 etc). If we condition on someone having a fever equal to 101 we would reject every single particle. Why? No particle will have exactly a fever of 101.

There are several ways to deal with this problem. One especially easy solution is to be less strict when rejecting particles. We could round all fevers to whole numbers.

There is an algorithm called "Likelihood Weighting" which sometimes helps, but which we don't cover in CS109. Instead, in class we talked about a new algorithm called Markov Chain Monte Carlo (MCMC) that allowed us to sample from the "posterior" probability: the distribution of random variables after (post) us fixing variables in the conditioned event. The version of MCMC we talked about is called Gibbs Sampling. While I don't require that students in CS109 know how to implement Gibbs Sampling, I wanted everyone to know that it exists and that it isn't beyond your capabilities. If you need to use it, you can learn it given the knowledge you have now.

MCMC does require more math than Joint Sampling. For every random variable you will need to specify how to calculate the likelihood of assignments given the variable's: parents, children and parents of its children (a set of variables cozily called a "blanket"). Want to learn more? Take CS221 or CS228!

4 Thoughts

While there are slightly-more-powerful "general inference algorithms" that you will get to learn in the future, it is worth recognizing that at this point we have reached an important milestone in CS109. You can take very complicated probability models (encoded as Bayesian networks) and can answer general inference queries on them. To get there we worked through the concrete example of predicting disease. While the WebMd website is great for home users, similar probability models are being used in thousands of hospitals around the world. As you are reading this general inference is being used to improve health care (and sometimes even save lives) for real human beings. That's some probability for computer scientists that is worth learning. Now there is just one last question for us to look into in CS109. What if we don't have an expert? Could we learn those probabilities from data?