Computational Probability and Inference

LaTeXdocument by Colin Leach, from an HTML original by MIT course staff

October 2016

Abstract

This is a set of notes for the online course "6.008.1x Computational Probability and Inference" given by the staff of MIT during Fall 2016. Most of the sections are copied from notes on the edX platform (with grateful thanks to those at MIT who provided them!), but I added some further sections as necessary. You should assume that everything here is MIT copyright.

All LaTeXsource is currently available at https://github.com/colinleach/probinf-notes but it is unlikely that it will remain publicly accessible. I plan to make the repository private once the course ends (late December 2016).

Contents

Chapter 1

Probability and Inference

1.1 Introduction to Probability

1.1.1 Introduction

Probabilities appear in everyday life and feed into how we make decisions. For example:

- The weather forecast might say that "tomorrow there is a 70% chance of rain". This 70% chance of rain is a probability, and if it is sufficiently high, then we may want to bring an umbrella when we go outdoors.
- We could predict that the probability of car traffic is higher during rush hour than otherwise, so if we don't want to be stuck in traffic while driving, we should avoid driving during rush hour.

We aim to build computer programs that can reason with probabilities.

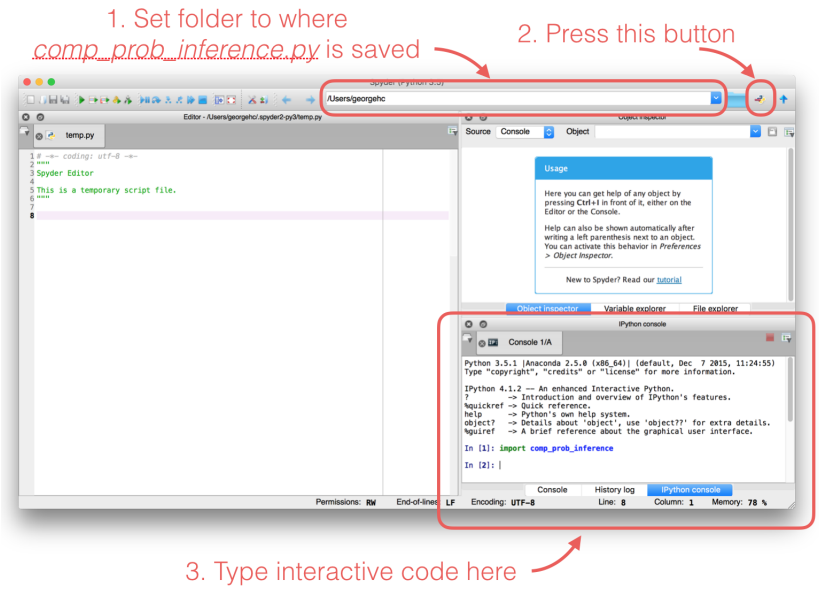
1.1.2 A First Look at Probability

Perhaps the simplest example of probability is flipping a fair coin for which we say that the probability of heads is 1/2 and, similarly, the probability of tails is also 1/2. (Don't worry, we'll see much more exciting problems soon!) What do we mean when we say that the probability of heads is 1/2?

The basic idea is that if we repeat this experiment of flipping a coin a huge number of times, say n, then the number of heads we should see should be close to $n/2$ as we increase n. While you could certainly try this out in real life by flipping a coin, say, 100,000 times, doing this would be disastrously tedious. Let's simulate these flips in Python instead.

Simulating Coin Flips Follow along in an IPython prompt within Spyder.

We have provided a package `comp_prob_inference.py`, which you should save to your computer. Within Spyder, do the following:



Let's start by importing the package comp_prob_inference:

```
> import comp_prob_inference
```

To simulate flipping a fair coin, enter:

```
> comp_prob_inference.flip_fair_coin()
```

You should get either 'heads' or 'tails'. Try re-running the above line a few times. You should see that the coin flip results are random.

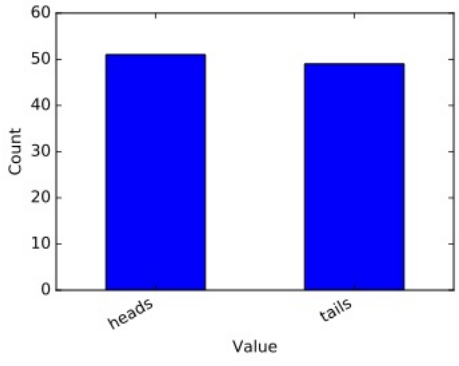
To flip the fair coin 100 times, enter:

```
> flips = comp_prob_inference.flip_fair_coins(100)
```

Let's plot how many times we see the two possible outcomes in the same bar graph, called a histogram:

```
> comp_prob_inference.plot_discrete_histogram(flips)
```

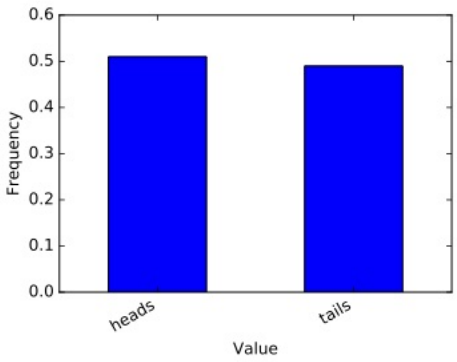
For example, we get the following plot:



Often what we will care about in this course is the fraction (also called the frequency of times an outcome happens. To plot the fraction of times heads or tails occurred, we again use the plot_discrete_histogram function but now add the keyword argument frequency=True:

```
> comp_prob_inference.plot_discrete_histogram(flips, frequency=True)
```

Doing so, we get the following plot:



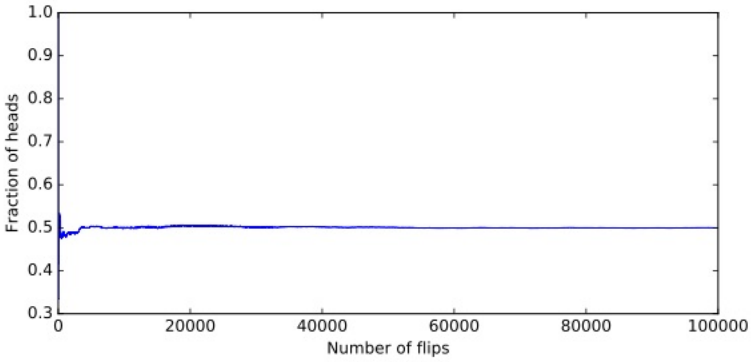
Next, let's plot the fraction of heads as a function of the number of flips (going up to 100,000 flips).

```
n = 100000
heads_so_far = 0
fraction_of_heads = []
for i in range(n):
    if comp_prob_inference.flip_fair_coin() == 'heads':
        heads_so_far += 1
    fraction_of_heads.append(heads_so_far / (i+1))
```

Note that `fraction_of_heads[i]` tells us what the fraction of heads is after the first i tosses. Then to actually plot the fraction of heads vs the number of tosses, enter the following:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 4))
plt.plot(range(1, n+1), fraction_of_heads)
plt.xlabel('Number of flips')
plt.ylabel('Fraction of heads')
```

For example, when we run this we get the following plot:



The fraction of heads initially can be far from 1/2 but as the number of flips increases, the fraction stabilizes and gets closer to 1/2, the probability of heads.

Computer note: Many times in this course, it will be helpful to run simulations to test code and plot histograms for different outcomes to get a sense of how likely the outcomes are. Simulations and visualizations can be powerful not only in making sure your code is working correctly but also to present results to people!

1.1.3 Probability and the Art of Modeling Uncertainty

Since probability effectively corresponds to a fraction, it is a value between 0 and 1. Of course, we can have impossible events that have probability 0, or events that deterministically happen and thus have probability 1. Each time we model uncertainty in the world, there will be some underlying experiment (such as flipping a coin in our running example). An event happens with probability $q \in [0,1]$ if in a massive number of repeats of the experiment, the event happens roughly a fraction q of the time; more repeats of the experiment make it so that the fraction gets closer to q.

Some times, an underlying experiment cannot possibly be repeated. Take for instance weather forecasting. Whereas we could actually physically flip a coin many times to repeat the same experiment, we cannot physically repeat a real-life experiment for what different realizations of tomorrow's weather will be. We could wait until tomorrow to see the weather, but then we would need a time machine to go back in time by one day to repeat and see what the weather is like tomorrow (and this assumes that there's some inherent randomness in tomorrow's weather)! In such a case, our only hope is to somehow model or simulate tomorrow's weather given measurements up to present time.

Different people could model the same real world problem differently! Throughout the course, a recurring challenge in building computer programs that reason probabilistically is figuring out how to model real-world problems. A good model — even if not actually accurate in describing, for instance, the science behind weather — enables us to make good predictions.

Once a weather forecaster has anchored some way of modeling or simulating weather, then if it claims that there's a 30% chance of rain tomorrow, we could interpret this as saying that *using their way of simulating tomorrow's weather*, in roughly 30% of simulated results for tomorrow's weather, there is rain.

1.2 Probability Spaces and Events

1.2.1 Two Ingredients to Modeling Uncertainty

When we think of an uncertain world, we will always think of there being some underlying experiment of interest. To model this uncertain world, it suffices to keep track of two things:

The set of all possible outcomes for the experiment: this set is called the sample space and is usually denoted by the Greek letter Omega Ω. (For the fair coin flip, there are exactly two possible outcomes: heads, tails. Thus, $\Omega = \{\text{heads}, \text{tails}\}$.)

The probability of each outcome: for each possible outcome, assign a probability that is at least 0 and at most 1. (For the fair coin flip, $\mathbb{P}(\text{heads}) = \frac{1}{2}$ and $\mathbb{P}(\text{tails}) = \frac{1}{2}$.)

Notation: Throughout this course, for any statement $\mathscr{S}$, "$\mathbb{P}(\mathscr{S})$" denotes the probability of $\mathscr{S}$ happening.

In Python:

```
> model = {'heads': 1/2, 'tails': 1/2}
```

In particular, we see that we can model uncertainty in code using a Python dictionary. The sample space is precisely the keys in the dictionary:

```
> sample_space = set(model.keys())
{'tails', 'heads'}
```

Of course, the dictionary gives us the assignment of probabilities, meaning that for each outcome in the sample space (i.e., for each key in the dictionary), we have an assigned probability:

```
> model['heads']
0.5
```

```
> model['tails']
0.5
```

A few important remarks:

- The sample space is always specified to be *collectively exhaustive*, meaning that every possible outcome is in it, and *mutually exclusive*, meaning that once the experiment is run (e.g., flipping the fair coin), exactly one possible outcome in the sample space happens. It's impossible for multiple outcomes in the sample space to simultaneously happen! It's also impossible for none of the outcomes to happen!
- Probabilities can be thought of as fractions of times outcomes occur; thus, probabilities are nonnegative and at least 0 and at most 1.
- If we add up the probabilities of all the possible outcomes in the sample space, we get 1. (For the fair coin flip, $\mathbb{P}(\text{heads}) + \mathbb{P}(\text{tails}) = \frac{1}{2} + \frac{1}{2} = 1$.)

Some intuition for this: Consider the coin flipping experiment. What does the fraction of times heads occur and the fraction of times tails occur add up to? Since these are the only two possible outcomes (and again, recall that these outcomes are exclusive in that they can't simultaneously occur, and exhaustive since they are the only possible outcomes), these two fractions will always sum to 1. For a massive number of repeats of the experiment, these two fractions correspond to $\mathbb{P}(\text{heads})$ and $\mathbb{P}(\text{tails})$; the fractions sum to 1 and so these probabilities also sum to 1.

1.2.2 Probability Spaces

At this point, we've actually already seen the most basic data structure used throughout this course for modeling uncertainty, called a *finite probability space* (in this course, we'll often also just call this either a *probability space* or a *probability model*):

A *finite probability space* consists of two ingredients:

- a sample space Ω consisting of a *finite* (i.e., not infinite) number of collectively exhaustive and mutually exclusive possible outcomes
- an assignment of probabilities: for each possible outcome $\omega \in \Omega$, we assign a probability $\mathbb{P}(\text{outcome } \omega)$ at least 0 and at most 1, where we require that the probabilities across all the possible outcomes in the sample space add up to 1:
  $\sum_{\omega \in \Omega} \mathbb{P}(\text{outcome } \omega) = 1$

Notation: As shorthand we occasionally use the tuple "$(\Omega, \mathbb{P})$" to refer to a finite probability space to remind ourselves of the two ingredients needed, sample space Ω and an assignment of probabilities $\mathbb{P}$. As we already saw, in code these two pieces can be represented together in a single Python dictionary. However, when we want to reason about probability spaces in terms of the mathematics, it's helpful to have names for the two pieces.

Why finite? Of the two pieces making up a finite probability space $(\Omega, \mathbb{P})$, the sample space Ω being finite is a fairly natural constraint, corresponding to how we typically work with Python dictionaries where there is only a finite number of keys. As we'll see, finite probability spaces are already extremely useful in practice. Pedagogically, finite probability spaces also provide a great intro to probability theory as they already carry a wealth of intuition, much of which carries over to a more complete story of general probability spaces!

1.2.3 Table Representation

A probability space is a data structure in that we can always visualize as a table of nonnegative entries that sum to 1. Let's see a concrete example of this, first writing the table out on paper and then coding it up.

Example: Suppose we have a model of tomorrow's weather given as follows: sunny with probability 1/2, rainy with probability 1/6, and snowy with probability 1/3. Here's the probability space, shown as a table:

		Probability
	sunny	1/2
Outcome	rainy	1/6
	snowy	1/3

Note: This a table of 3 nonnegative entries that sum to 1. The rows correspond to the sample space $\Omega = \{\text{sunny}, \text{rainy}, \text{snowy}\}$.

We will often use this table representation of a probability space to tell you how we're modeling uncertainty for a particular problem. It provides the simplest of visualizations of a probability space.

Of course, in Python code, the above probability space is given by:

```
prob_space = {'sunny': 1/2, 'rainy': 1/6, 'snowy': 1/3}
```

A different way to code up the same probability space is to separately specify the outcomes (i.e., the sample space) and the probabilities:

```
outcomes = ['sunny', 'rainy', 'snowy']
probabilities = np.array([1/2, 1/6, 1/3])
```

The i-th entry of `outcomes` has probability given by the i-th entry of `probabilities`. Note that `probabilities` is a vector of numbers that we represent as a Numpy array. Numpy has various built-in methods that enable us to easily work with vectors (and more generally arrays) of numbers.

1.2.4 More on Sample Spaces

In the video, we saw that a sample space encoding the outcomes of 2 coin flips encodes all the information for 1 coin flip as well. Thus, we could use the same sample space to model a single coin flip. However, if we really only cared about a single coin flip, then a sample space encoding 2 coin flips is richer than we actually need it to be!

When we model some uncertain situation, how we specify a sample space is not unique. We saw an example of this already in an earlier exercise where for rolling a single six-sided die, we can choose to name the outcomes differently, saying for instance "roll 1" instead of "1". We could even add a bunch of extraneous outcomes that all have probability 0. We could add extraneous information that doesn't matter such as "Alice rolls 1", "Bob rolls 1", etc where we enumerate out all the people who could roll the die in which the outcome is a 1. Sure, depending on the problem we are trying to solve, maybe knowing who rolled the die is important, but if we don't care about who rolled the die, then the information isn't helpful but it's still possible to include this information in the sample space.

Generally speaking it's best to choose a sample space that is as simple as possible for modeling what we care about solving. For example, if we were rolling a six-sided die, and we actually only care about whether the face shows up at least 4 or not, then it's sufficient to just keep track of two outcomes, "at least 4" and "less than 4".

1.2.5 Probabilities with Events

TODO – add notes from video

1.2.6 Events as Sets

TODO – add notes from video

1.2.7 Code for Dealing with Sets in Python

In the video, the set operations can actually be implemented in Python as follows:

```
sample_space = {'HH', 'HT', 'TH', 'TT'}
A = {'HT', 'TT'}
B = {'HH', 'HT', 'TH'}
C = {'HH'}
A_intersect_B = A.intersection(B) # equivalent to "B.intersection(A)" or "A & B"
A_union_C = A.union(C) # equivalent to "C.union(A)" and also "A | C"
B_complement = sample_space.difference(B) # equivalent also to "sample_space - B"
```

1.2.8 Probabilities with Events and Code

From the videos, we see that an event is a subset of the sample space Ω. If you remember our table representation for a probability space, then an event could be thought of as a subset of the rows, and the probability of the event is just the sum of the probability values in those rows!

The *probability of an event* $\mathscr{A} \subseteq \Omega$ is the sum of the probabilities of the possible outcomes in $\mathscr{A}$:

$\mathbb{P}(\mathscr{A}) \triangleq \sum_{\omega \in \mathscr{A}} \mathbb{P}(\text{outcome } \omega)$,

where "$\triangleq$" means "defined as".

We can translate the above equation into Python code. In particular, we can compute the probability of an event encoded as a Python set `event`, where the probability space is encoded as a Python dictionary `prob_space`:

```
def prob_of_event(event, prob_space):
    total = 0
    for outcome in event:
        total += prob_space[outcome]
    return total
```

Here's an example of how to use the above function:

```
prob_space = {'sunny': 1/2, 'rainy': 1/6, 'snowy': 1/3}
rainy_or_snowy_event = {'rainy', 'snowy'}
print(prob_of_event(rainy_or_snowy_event, prob_space))
```

1.3 Random Variables

1.3.1 A First Look at Random Variables

Follow along in an IPython prompt.

We continue with our weather example.

```
> prob_space = {'sunny': 1/2, 'rainy': 1/6, 'snowy': 1/3}
```

We can simulate tomorrow's weather using the above model of the world. Let's simulate two different values, one

(which we'll call `W` for "weather") for whether tomorrow will be sunny, rainy, or snowy, and another (which we'll call `I` for "indicator") that is 1 if it is sunny and 0 otherwise:

```
> random_outcome = comp_prob_inference.sample_from_finite_probability_space(prob_space)
> W = random_outcome
> if random_outcome == 'sunny':
>     I = 1
> else:
>     I = 0
```

Print out the variables `W` or `I` to see that they take on specific values. Then re-run the above block of code a few times.

You should see that `W` and `I` change and are random (following the probabilities given by the probability space).

This code shows something that's of key importance that we'll see throughout the course. Variables `W` and `I` store the values of what are called *random variables*.

1.3.2 Random Variables

To mathematically reason about a random variable, we need to somehow keep track of the full range of possibilities for what the random variable's value could be, and how probable different instantiations of the random variable are. The resulting formalism may at first seem a bit odd but as we progress through the course, it will become more apparent how this formalism helps us study real-world problems and address these problems with powerful solutions.

To build up to the formalism, first note, computationally, what happened in the code in the previous part.

1. First, there is an underlying probability space $(\Omega, \mathbb{P})$, where $\Omega = \{\text{sunny}, \text{rainy}, \text{snowy}\}$, and

$$\begin{aligned} \mathbb{P}(\text{sunny}) &= 1/2, \\ \mathbb{P}(\text{rainy}) &= 1/6, \\ \mathbb{P}(\text{snowy}) &= 1/3. \end{aligned}$$

2. A random outcome $\omega \in \Omega$ is sampled using the probabilities given by the probability space $(\Omega, \mathbb{P})$. This step corresponds to an underlying experiment happening.

3. Two random variables are generated:

   - W is set to be equal to ω. As an equation:

$$W(\omega) = \omega \quad \text{for } \omega \in \{\text{sunny}, \text{rainy}, \text{snowy}\}.$$

     This step perhaps seems entirely unnecessary, as you might wonder "Why not just call the random outcome W instead of ω?" Indeed, this step isn't actually necessary for this particular example, but the formalism for random variables has this step to deal with what happens when we encounter a random variable like I.

   - I is set to 1 if $\omega = \text{sunny}$, and 0 otherwise. As an equation:

$$I(\omega) = \begin{cases} 1 & \text{if } \omega = \text{sunny}, \\ 0 & \text{if } \omega \in \{\text{rainy}, \text{snowy}\}. \end{cases}$$

     Importantly, multiple possible outcomes (rainy or snowy) get mapped to the same value 0 that I can take on.

We see that random variable W maps the sample space $\Omega = \{\text{sunny}, \text{rainy}, \text{snowy}\}$ to the same set $\{\text{sunny}, \text{rainy}, \text{snowy}\}$. Meanwhile, random variable I maps the sample space $\Omega = \{\text{sunny}, \text{rainy}, \text{snowy}\}$ to the set $\{0, 1\}$.

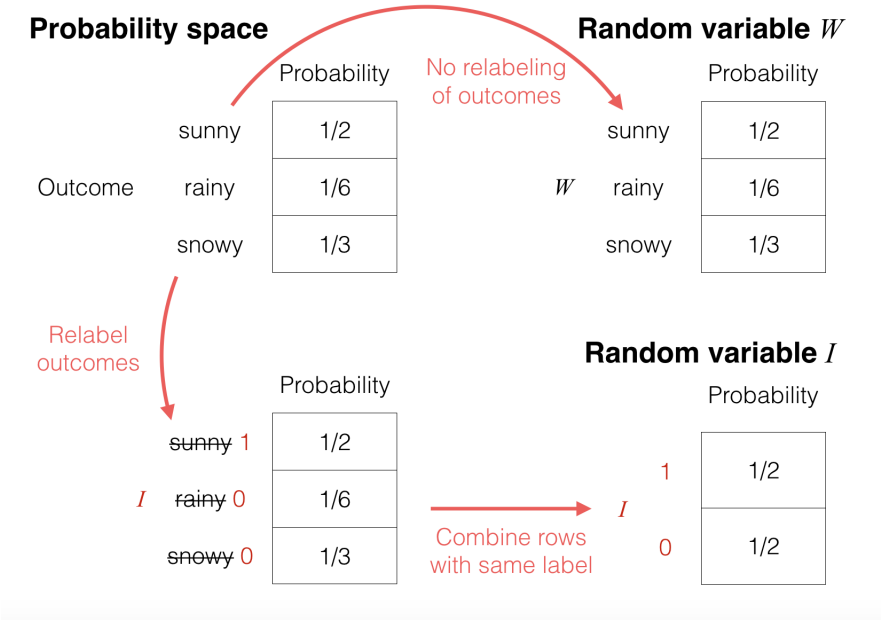
In general:

Definition of a "finite random variable" (in this course, we will just call this a "random variable"): Given a finite probability space $(\Omega, \mathbb{P})$, a finite random variable X is a mapping from the sample space Ω to a set of values X that random variable X can take on. (We will often call X the "alphabet" of random variable X.)

For example, random variable W takes on values in the alphabet sunny,rainy,snowy, and random variable I takes on values in the alphabet $\{0, 1\}$.

Quick summary: There's an underlying experiment corresponding to probability space $(\Omega, \mathbb{P})$. Once the experiment is run, let $\omega \in \Omega$ denote the outcome of the experiment. Then the random variable takes on the specific value of $X(\omega) \in \mathcal{X}$.

Explanation using a picture: Continuing with the weather example, we can pictorially see what's going on by looking at the probability tables for: the original probability space, the random variable W, and the random variable I:



These tables make it clear that a "random variable" really is just reassigning/relabeling what the values are for the possible outcomes in the underlying probability space (given by the top left table):

- In the top right table, random variable W does not do any sort of relabeling so its probability table looks the same as that of the underlying probability space.
- In the bottom left table, the random variable I relabels/reassigns "sunny" to 1, and both "rainy" and "snowy" to 0. Intuitively, since two of the rows now have the same label 0, it makes sense to just combine these two rows, adding their probabilities $\frac{1}{6} + \frac{1}{3} = \frac{1}{2}$. This results in the bottom right table.

Technical note: Even though the formal definition of a finite random variable doesn't actually make use of the probability assignment $\mathbb{P}$, the probability assignment will become essential as soon as we talk about how probability works with random variables.

1.3.3 Two Ways to Specify a Random Variable in Code

Two ways we can fully represent a random variable on a computer are as follows.

Approach 1. Go with the mathematical definition of a random variable. First, specify what the underlying probability space is:

```
> prob_space = {'sunny': 1/2, 'rainy': 1/6, 'snowy': 1/3}
```

Then provide a way to map from the sample space to the alphabet:

```
> W_mapping = {'sunny': 'sunny', 'rainy': 'rainy', 'snowy': 'snowy'}
> I_mapping = {'sunny': 1, 'rainy': 0, 'snowy': 0}
```

Then we can generate a random sample/draw for random variables W and I:

```
> random_outcome = comp_prob_inference.sample_from_finite_probability_space(prob_space)
> W = W_mapping[random_outcome]
> I = I_mapping[random_outcome]
```

Approach 2. Remember how we wrote out probability tables for random variables W and I? Let's directly store these probability tables:

```
> W_table = {'sunny': 1/2, 'rainy': 1/6, 'snowy': 1/3}
> I_table = {0: 1/2, 1: 1/2}
```

Treating the tables as probability spaces, draw samples for W and I:

```
> W = comp_prob_inference.sample_from_finite_probability_space(W_table)
> I = comp_prob_inference.sample_from_finite_probability_space(I_table)
```

1.3.4 Random Variables Notation and Terminology

In this course, we denote random variables with capital/uppercase letters, such as X, W, I, etc. We use the phrases "probability table", "probability mass function" (abbreviated as PMF), and "probability distribution" (often simply called a distribution) to mean the same thing, and in particular we denote the probability table for X to be p_X or $p_X(\cdot)$).

We write $p_X(x)$ to denote the entry of the probability table that has label $x \in \mathscr{X}$ where $\mathscr{X}$ is the set of values that random variable $\mathscr{X}$ takes on. Note that we use lowercase letters like x to denote variables storing nonrandom values. We can also look up values in a probability table using specific outcomes, e.g., from earlier, we have $p_W(\text{rainy}) = 1/6$ and $p_I(1) = 1/2$.

Note that we use the same notation as in math where a function f might also be written as $f(\cdot)$ to explicitly indicate that it is the function of one variable. Both f and $f(\cdot)$ refer to a function whereas $f(x)$ refers to the value of the function f evaluated at the point x.

As an example of how to use all this notation, recall that a probability table consists of nonnegative entries that add up to 1. In fact, each of the entries is at most 1 (otherwise the numbers would add to more than 1). For a random variable X taking on values in $\mathscr{X}$, we can write out these constraints as:

$$0 \leqslant p_X(x) \leqslant 1 \quad \text{for all } x \in \mathscr{X}, \qquad \sum_{x \in \mathscr{X}} p_X(x) = 1.$$

Often in the course, if we are making statements about all possible outcomes of X, we will omit writing out the alphabet $\mathscr{X}$ explicitly. For example, instead of the above, we might write the following equivalent statement:

$$0 \leqslant p_X(x) \leqslant 1 \quad \text{for all } x, \qquad \sum_x p_X(x) = 1.$$

1.4 Jointly Distributed Random Variables

1.4.1 Relating Two Random Variables

At the most basic level, inference refers to using an observation to reason about some unknown quantity. In this course, the observation and the unknown quantity are represented by random variables. The main modeling question is: How do these random variables relate?

Let's build on our earlier weather example, where now another outcome of interest appears, the temperature, which we quantize into to possible values "hot" and "cold". Let's suppose that we have the following probability space:

Outcome	Probability
sunny, hot	3/10
sunny, cold	1/5
rainy, hot	1/30
rainy, cold	2/15
snowy, hot	0
snowy, cold	1/3

You can check that the nonnegative entries do add to 1. If we let random variable W be the weather (sunny, rainy, snowy) and random variable T be the temperature (hot, cold), then notice that we could rearrange the table in the following fashion:

Outcome	Probability
W = sunny, T = hot	3/10
W = sunny, T = cold	1/5
W = rainy, T = hot	1/30
W = rainy, T = cold	2/15
W = snowy, T = hot	0
W = snowy, T = cold	1/3

Rearrange table entries ↓

W \ T	hot	cold
sunny	3/10	1/5
rainy	1/30	2/15
snowy	0	1/3

When we talk about two separate random variables, we could view them either as a single "super" random variable that happens to consist of a pair of values (the first table; notice the label for each outcome corresponds to a pair of values), or we can view the two separate variables along their own different axes (the second table).

The first table tells us what the underlying probability space is, which includes what the sample space is (just read off the outcome names) and what the probability is for each of the possible outcomes for the underlying experiment at hand.

The second table is called a *joint probability table* $p_{W,T}$ for random variables W and T, and we say that random variables W and T are *jointly distributed* with the above distribution. Since this table is a rearrangement of the

earlier table, it also consists of nonnegative entries that add to 1.

The joint probability table gives probabilities in which W and T co-occur with specific values. For example, in the above, the event that "W = sunny" and the event that "T = hot" co-occur with probability 3/10. Notationally, we write

$$p_{W,T}(\text{sunny}, \text{hot}) = \mathbb{P}(W = \text{sunny}, T = \text{hot}) = \tfrac{3}{10}.$$

Conceptual note: Given the joint probability table, we can easily go backwards and write out the first table above, which is the underlying probability space.

Preview of inference: Inference is all about answering questions like "if we observe that the weather is rainy, what is the probability that the temperature is cold?" Let's take a look at how one might answer this question.

First, if we observe that it is rainy, then we know that "sunny" and "snowy" didn't happen so those rows aren't relevant anymore. So the space of possible realizations of the world has shrunk to two options now: $(W = \text{rainy}, T = \text{hot})$ or $(W = \text{rainy}, T = \text{cold})$. But what about the probabilities of these two realizations? It's not just 1/30 and 2/15 since these don't sum to 1 — by observing things, adjustments can be made to the probabilities of different realizations but they should still form a valid probability space.

Why not just scale both 1/30 and 2/15 by the same constant so that they sum to 1? This can be done by dividing 1/30 and 2/15 by their sum:

$$\text{hot:} \quad \frac{\frac{1}{30}}{\frac{1}{30}+\frac{2}{15}} = \frac{1}{5}, \qquad \text{cold:} \quad \frac{\frac{2}{15}}{\frac{1}{30}+\frac{2}{15}} = \frac{4}{5}.$$

Now they sum to 1. It turns out that, given that we've observed the weather to be rainy, these are the correct probabilities for the two options "hot" and "cold". Let's formalize the steps. We work backwards, first explaining what the the denominator "$\frac{1}{30} + \frac{2}{15} = \frac{1}{6}$" above comes from.

1.4.2 Representing a Joint Probability Table in Code

There are various ways to represent a joint probability table in code. Here are a few.

Note that we have updated `comp_prob_inference.py`! Please re-download it!

Approach 0: Don't actually represent the joint probability table. This doesn't store the 2D table at all but is a first attempt at coding up something that has all the information in the joint probability table. Specifically, we can just code up the entries like how we coded up a probability space:

```
> prob_table = {('sunny', 'hot'): 3/10,
>     ('sunny', 'cold'): 1/5,
>     ('rainy', 'hot'): 1/30,
>     ('rainy', 'cold'): 2/15,
>     ('snowy', 'hot'): 0,
>     ('snowy', 'cold'): 1/3}
```

Thus, if we want the probability of W = rainy and T = cold, we write:

```
> prob_table[('rainy', 'cold')]
0.13333333333333333
```

Some times, this representation is sufficient. Given a specific weather and temperature stored as strings in `w` and `t` respectively, `prob_table[(w, t)]` gives you the joint probability table evaluated at $W =$ `w` and $T =$ `t`.

Approach 1: Use dictionaries within a dictionary. This works as follows:

```
> prob_W_T_dict = {}
> for w in {'sunny', 'rainy', 'snowy'}:
>     prob_W_T_dict[w] = {}
>
> prob_W_T_dict['sunny']['hot'] = 3/10
> prob_W_T_dict['sunny']['cold'] = 1/5
> prob_W_T_dict['rainy']['hot'] = 1/30
> prob_W_T_dict['rainy']['cold'] = 2/15
> prob_W_T_dict['snowy']['hot'] = 0
> prob_W_T_dict['snowy']['cold'] = 1/3
>
> comp_prob_inference.print_joint_prob_table_dict(prob_W_T_dict)
          cold      hot
rainy 0.133333 0.033333
snowy 0.333333 0.000000
sunny 0.200000 0.300000
```

Note that because dictionary keys aren't ordered, the row ordering and column ordering need not match the tables we have been showing in the course notes earlier. This is not a problem.

If we want the probability of $W =$ rainy and $T =$ cold, we write:

```
> prob_W_T_dict['rainy']['cold']
0.13333333333333333
```

The probability for $W =$ `w` and $T =$ `t` is stored in `prob_W_T_dict[w][t]`.

Approach 2: Use a 2D array. Another approach is to directly store the joint probability table as a 2D array, separately keeping track of what the rows and columns are. We use a NumPy array (but really you could use Python lists within a Python list, much like how the previous approach used dictionaries within a dictionary; the indexing syntax changes only slightly):

```
> import numpy as np
> prob_W_T_rows = ['sunny', 'rainy', 'snowy']
> prob_W_T_cols = ['hot', 'cold']
> prob_W_T_array = np.array([[3/10, 1/5], [1/30, 2/15], [0, 1/3]])
> comp_prob_inference.print_joint_prob_table_array(prob_W_T_array, prob_W_T_rows, prob_W_T_cols)
          hot      cold
sunny 0.300000 0.200000
rainy 0.033333 0.133333
snowy 0.000000 0.333333
```

Note that the ordering of rows is specified, as is the ordering of the columns, unlike in the dictionaries within a dictionary representation.

Retrieving a specific table entry requires a little bit more code since we need to figure out what the row and column numbers are corresponding to a specific pair of row and column labels. For example, if we want the probability of $W =$ rainy and $T =$ cold, we write:

```
> prob_W_T_array[prob_W_T_rows.index('rainy'), prob_W_T_cols.index('cold')]
0.13333333333333333
```

Note that `prob_W_T_rows.index('rainy')` finds the row number (starting from 0) corresponding to "rainy".

Using `.index` does a search through the whole list of row/column labels, which for large lists can be slow. Let's fix this!

A cleaner and faster way is to create separate dictionaries mapping the row and column labels to row and column indices in the 2D array. In other words, instead of writing `prob_W_T_rows.index('rainy')` to find the row number for 'rainy', we want to just be able to write something like `prob_W_T_row_mapping['rainy']`, which returns the row number. We can define Python variable `prob_W_T_row_mapping` as follows:

```
> prob_W_T_row_mapping = {}
> for index, label in enumerate(prob_W_T_rows):
>     prob_W_T_row_mapping[label] = index
```

Note that `enumerate(prob_W_T_rows)` produces an iterator that consists of pairs `(0, 'sunny'), (1, 'rainy'), (2, 'snowy')`. You can see this by entering:

```
> print(list(enumerate(prob_W_T_rows)))
[(0, 'sunny'), (1, 'rainy'), (2, 'snowy')]
```

Note that each pair consists of the row number followed by the label.

In fact, the three lines we used to define `prob_W_T_row_mapping` can be written in one line with a Python dictionary comprehension:

```
> prob_W_T_row_mapping = {label: index for index, label in enumerate(prob_W_T_rows)}
```

We can do the same thing to define a mapping of column labels to column numbers:

```
> prob_W_T_col_mapping = {label: index for index, label in enumerate(prob_W_T_cols)}
```

In summary, we can represent the joint probability table as follows:

```
> prob_W_T_rows = ['sunny', 'rainy', 'snowy']
> prob_W_T_cols = ['hot', 'cold']
> prob_W_T_row_mapping = {label: index for index, label in enumerate(prob_W_T_rows)}
> prob_W_T_col_mapping = {label: index for index, label in enumerate(prob_W_T_cols)}
> prob_W_T_array = np.array([[3/10, 1/5], [1/30, 2/15], [0, 1/3]])
```

Now the probability for $W =$ `w` and $T =$ `t` is given by:

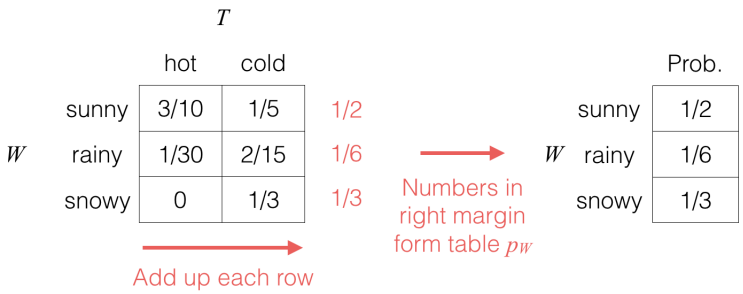
```
> prob_W_T_array[prob_W_T_row_mapping[w], prob_W_T_col_mapping[t]]
```

Some remarks: The 2D array representation, as we'll see soon, is very easy to work with when it comes to basic operations like summing rows, and retrieving a specific row or column. The main disadvantage of this representation is that you need to store the whole array, and if the alphabet sizes of the random variables are very large, then storing the array will take a lot of space!

The dictionaries within a dictionary representation allows for easily retrieving rows but not columns (try it: write a Python function that picks out a specific row and another function that picks out a specific column; you should see that retrieving a row is easier because it corresponds to looking at the value stored for a single key of the outer dictionary). This also means that summing a column's probabilities is more cumbersome than summing a row's probabilities. However, a huge advantage of this way of representing a joint probability table is that in many problems, we have a massive joint probability table that is mostly filled with 0's. Thus, the 2D array representation would require storing a very, very large table with many 0's, whereas the dictionaries within a dictionary representation is able to only store the nonzero table entries. We'll see more about this issue when we look at robot localization in the second section of the course on inference in graphical models.

1.4.3 Marginalization

Given a joint probability table, often we'll want to know what the probability table is for just one of the random variables. We can do this by just summing or "marginalizing" out the other random variables. For example, to get the probability table for random variable W, we do the following:



We take the joint probability table (left-hand side) and compute out the row sums (which we've written in the margin).

The right-hand side table is the probability table p_W for random variable W; we call this resulting probability distribution the marginal distribution of W (put another way, it is the distribution obtained by marginalizing out the random variables that aren't W).

In terms of notation, the above marginalization procedure whereby we used the joint distribution of W and T to produce the marginal distribution of W is written:

$$p_W(w) = \sum_{t \in \mathcal{T}} p_{W,T}(w,t),$$

where $\mathcal{T}$ is the set of values that random variable T can take on. In fact, throughout this course, we will often omit explicitly writing out the alphabet of values that a random variable takes on, e.g., writing instead

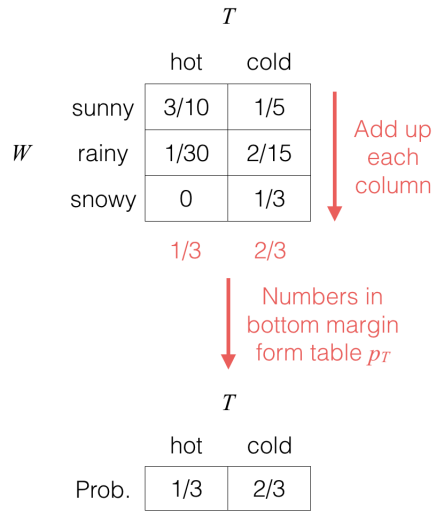
$$p_W(w) = \sum_{t} p_{W,T}(w,t).$$

It's clear from context that we're summing over all possible values for t, which is going to be the values that random variable T can possibly take on.

As a specific example,

$$p_W(\text{rainy}) = \sum_t p_{W,T}(\text{rainy},t) = \underbrace{p_{W,T}(\text{rainy},\text{hot})}_{1/30} + \underbrace{p_{W,T}(\text{rainy},\text{cold})}_{2/15} = \tfrac{1}{6}.$$

We could similarly marginalize out random variable W to get the marginal distribution p_T for random variable T:



(Note that whether we write a probability table for a single variable horizontally or vertically doesn't actually matter.)

As a formula, we would write:

$$p_T(t) = \sum_w p_{W,T}(w,t).$$

For example,

$$p_T(\text{hot}) = \sum_w p_{W,T}(w,\text{hot}) = \underbrace{p_{W,T}(\text{sunny},\text{hot})}_{3/10} + \underbrace{p_{W,T}(\text{rainy},\text{hot})}_{1/30} + \underbrace{p_{W,T}(\text{snowy},\text{hot})}_{0} = \tfrac{1}{3}.$$

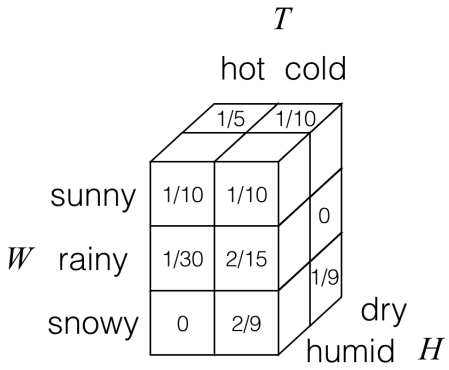
In general:

Marginalization: Consider two random variables X and Y (that take on values in the sets $\mathscr{X}$ and $\mathscr{Y}$ respectively) with joint probability table $p_{X,Y}$. For any $x \in \mathscr{X}$, the *marginal probability* that $X = x$ is given by

$$p_X(x) = \sum_y p_{X,Y}(x,y).$$

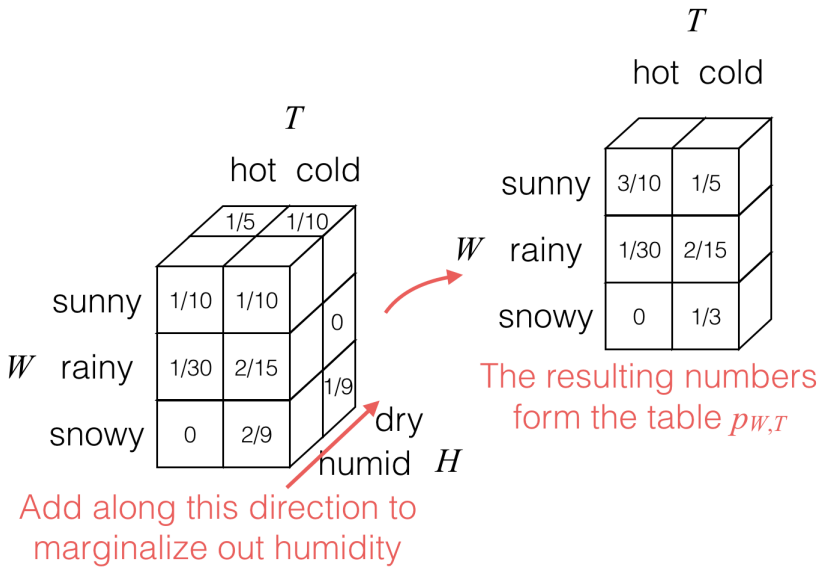
1.4.4 Marginalization for Many Random Variables

What happens when we have more than two random variables? Let's build on our earlier example and suppose that in addition to weather W and temperature T, we also had a random variable H for humidity that takes on values in the alphabet dry, humid. Then having a third random variable, we can draw out a 3D joint probability table for random variables W, T, and H. As an example, we could have the following:



Here, each of the cubes/boxes stores a probability. Not visible are two of the cubes in the back left column, which for this particular example both have probability values of 0.

Then to marginalize out the humidity H, we would add values as follows:



The result is the joint probability table for weather W and temperature T, shown still in 3D cubes with each cube storing a single probability.

As an equation:

$$p_{W,T}(w,t) = \sum_h p_{W,T,H}(w,t,h).$$

In general, for three random variables X, Y, and Z with joint probability table pX,Y,Z, we have

$$\begin{aligned} p_{X,Y}(x,y) &= \sum_z p_{X,Y,Z}(x,y,z), \\ p_{X,Z}(x,z) &= \sum_y p_{X,Y,Z}(x,y,z), \\ p_{Y,Z}(y,z) &= \sum_x p_{X,Y,Z}(x,y,z). \end{aligned}$$

Note that we can marginalize out different random variables in succession. For example, given joint probability table $p_{X,Y,Z}$, if we wanted the probability table p_X, we can get it by marginalizing out the two random variables Y and Z:

$$p_X(x) = \sum_y p_{X,Y}(x,y) = \sum_y \left(\sum_z p_{X,Y,Z}(x,y,z) \right).$$

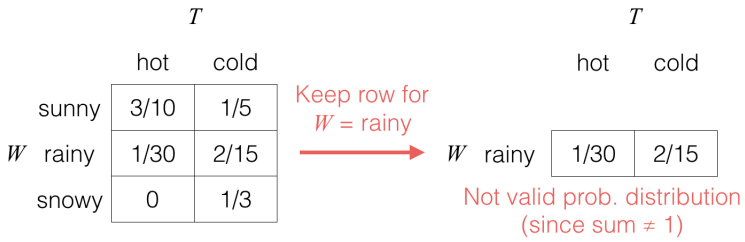
Even with more than three random variables, the idea is the same. For example, with four random variables W, X, Y, and Z with joint probability table $p_{W,X,Y,Z}$, if we want the joint probability table for X and Y, we would do the following:

$$p_{X,Y}(x,y) = \sum_w \left(\sum_z p_{W,X,Y,Z}(w,x,y,z) \right).$$

1.4.5 Conditioning for Random Variables

When we observe that a random variable takes on a specific value (such as W = rainy from earlier for which we say that we condition on random variable W taking on the value "rainy"), this observation can affect what we think are likely or unlikely values for another random variable.

When we condition on W = rainy, we do a two-step procedure; first, we only keep the row for W corresponding to the observed value:



Second, we "normalize" the table so that its entries add up to 1, which corresponds to dividing it by the sum of the entries, which is equal to $p_W(\text{rainy})$ in this case:

Notation: The resulting probability table $p_{T|W}(\cdot \mid \text{rainy})$ is associated with the random variable denoted $(T \mid W = \text{rainy})$; we use "|" to denote that we're conditioning on things to the right of "|" happening (these are things that we have observed or that we are given as having happened). We read "$T \mid W = \text{rainy}$" as either "T given W is rainy" or "T conditioned on W being rainy". To refer to specific entries of the table, we write, for instance,

$$p_{T|W}(\text{cold} \mid \text{rainy}) = \mathbb{P}(T = \text{cold} \mid W = \text{rainy}) = \tfrac{4}{5}.$$

In general:

Conditioning: Consider two random variables X and Y (that take on values in the sets $\mathscr{X}$ and $\mathscr{Y}$ respectively) with joint probability table $p_{X,Y}$ (from which by marginalization we can readily compute the marginal probability table p_Y). For any $x \in \mathscr{X}$ and $y \in \mathscr{Y}$ such that $p_Y(y) > 0$, the conditional probability of event $X = x$ given event $Y = y$ has happened is

$$p_{X|Y}(x \mid y) \triangleq \frac{p_{X,Y}(x,y)}{p_Y(y)}.$$

For example,

$$p_{T|W}(\text{cold} \mid \text{rainy}) = \frac{p_{W,T}(\text{rainy},\text{cold})}{p_W(\text{rainy})} = \frac{\frac{2}{15}}{\frac{1}{6}} = \frac{4}{5}.$$

Computational interpretation: To compute $p_{X|Y}(x \mid y)$, take the entry $p_{X,Y}(x,y)$ in the joint probability table corresponding to $X = x$ and $Y = y$, and then divide the entry by $p_Y(y)$, which is an entry in the marginal probability table p_Y for random variable Y.

1.4.6 Moving Toward a More General Story for Conditioning

Jointly distributed random variables play a central role in this course. Remember that we will model observations as random variables and the quantities we want to infer also as random variables. When these random variables are jointly distributed so that we have a probabilistic way to describe how they relate (through their joint probability table), then we can systematically and quantitatively produce inferences.

We just saw how to condition on a random variable taking on a specific value. What about if we wanted to condition on a random variable taking on any one of of many values rather just one specific value? To answer this question, we look at a more general story of conditioning which is in terms of events.

1.5 Conditioning on Events

1.5.1 Conditioning on Events Intro

The next important concept is called conditioning. In practice, we often encounter this kind of situation in that we have a probability model for some situation, but then we made some observation or something happened, and it changed the probability model. For example, if you observe that it was raining today, then the probability of tomorrow being a rainy day would certainly change. If we observed some information about a certain company today, then the probability that its stock price would increase tomorrow would be different. How do we address this in our formal mathematical language? This is about conditioning.

Now let's go back to this familiar picture of a general sample space, Ω. Suppose we have a probability model all built up, $(\Omega, \mathbb{P})$. There are two events in general A and B. They could have some intersection $A \cap B$. Depending on where the outcome falls, we say event A occurs, event B occurs, or both A and B occur.

Suppose that we have an observation that A occurred. That means the outcome of the random experiment is $\omega \in A$. Now we should not be worrying about anything outside of A, because they don't happen anymore. The set A becomes our new sample space: $\Omega \longrightarrow A$. We still don't know exactly where the outcome would be. And therefore, there should be a new sample space, a new model, and a new probability assignment $\mathbb{P}$ conditioned on A: $\mathbb{P} \longrightarrow \mathbb{P}(\cdot|A)$. This is the conditional probability, conditioned on "event A occurs".

For any possible outcome ω from the original picture, what is $\mathbb{P}(\omega|A)$? There could be two different cases:

- If $\omega \notin A$, $\mathbb{P}(\omega|A) = 0$. That's never going to occur anymore.
- If $\omega \in A$, $\mathbb{P}(\omega|A) = \frac{\mathbb{P}(\omega)}{\mathbb{P}(A)}$.

Scaling with $\frac{1}{\mathbb{P}(A)}$ normalizes the conditional distribution so that $\sum_{\omega \in A} \mathbb{P}(\omega|A) = 1$.

1.5.2 The Product Rule for Events

We should start to avoid writing probability in this way. We should always think of probability of some kind of event. So when I write an ω, I'm thinking of a single element event — a set with only one element ω. A better way of writing this is to look at the probability of another event B given A occurs. The only way for B to occur is that the random experiment gives an outcome in $A \cap B$. The conditional probability after normalizing is $\mathbb{P}(B|A) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(A)}$.

Alternatively, this can be written $\mathbb{P}(A \cap B) = \mathbb{P}(A) \cdot \mathbb{P}(B|A)$. This is called the "product rule"

1.5.3 Bayes' Theorem for Events

Important note about dividing by probabilities: We will often divide by probabilities. In videos, we might not always say this, but this is required: we cannot divide by 0. To ensure this, we will not condition on events that have probability 0.

Given two events $\mathscr{A}$ and $\mathscr{B}$ (both of which have positive probability), Bayes' theorem, also called Bayes' rule or Bayes' law, gives a way to compute $\mathbb{P}(\mathscr{A}|\mathscr{B})$ in terms of $\mathbb{P}(\mathscr{B}|\mathscr{A})$. This result turns out to be extremely useful for inference because often times we want to compute one of these, and the other is known or otherwise straightforward to compute.

Bayes' theorem is given by

$$\mathbb{P}(\mathscr{A}|\mathscr{B}) = \frac{\mathbb{P}(\mathscr{B}|\mathscr{A})\mathbb{P}(\mathscr{A})}{\mathbb{P}(\mathscr{B})}.$$

The proof of why this is the case is a one liner:

$$\mathbb{P}(\mathscr{A}|\mathscr{B}) \stackrel{(a)}{=} \frac{\mathbb{P}(\mathscr{A}\cap\mathscr{B})}{\mathbb{P}(\mathscr{B})} \stackrel{(b)}{=} \frac{\mathbb{P}(\mathscr{B}|\mathscr{A})\mathbb{P}(\mathscr{A})}{\mathbb{P}(\mathscr{B})},$$

where step (a) is by the definition of conditional probability for events, and step (b) is due to the product rule for events (which follows from rearranging the definition of conditional probability for $\mathbb{P}(\mathscr{B}|\mathscr{A})$).

1.5.4 Practice Problem: Bayes' Theorem and Total Probability

Your problem set is due in 15 minutes! It's in one of your drawers, but they are messy, and you're not sure which one it's in.

The probability that the problem set is in drawer k is d_k. If drawer k has the problem set and you search there, you have probability p_k of finding it. There are a total of m drawers.

Suppose you search drawer i and do not find the problem set.

- (a) Find the probability that the paper is in drawer j, where $j \neq i$.
- (b) Find the probability that the paper is in drawer i.

Solution: Let A_k be the event that the problem set is in drawer k, and B_k be the event that you find the problem set in drawer k.

(a) We'll express the desired probability as $\mathbb{P}(A_j|B_i^c)$. Since this quantity is difficult to reason about directly, we'll use Bayes' rule:

$$\mathbb{P}(A_j|B_i^c) = \frac{\mathbb{P}(B_i^c|A_j)\mathbb{P}(A_j)}{\mathbb{P}(B_i^c)}$$

The first probability, $\mathbb{P}(B_i^c|A_j)$, expresses the probability of not finding the problem set in drawer i given that it's in a different drawer j. Since it's impossible to find the paper in a drawer it isn't in, this is just 1.

The second quantity, $\mathbb{P}(A_j)$, is given to us in the problem statement as d_j.

The third probability, $\mathbb{P}(B_i^c) = 1 - \mathbb{P}(B_i)$, is difficult to reason about directly. But, if we knew whether or not the paper was in the drawer, it would become easier. So, we'll use total probability:

$$\begin{aligned}\mathbb{P}(B_i) &= \mathbb{P}(B_i|A_i)\mathbb{P}(A_i) + \mathbb{P}(B_i|A_i^c)\mathbb{P}(A_i^c) \\ &= p_i d_i + 0(1-d_i)\end{aligned}$$

Putting these terms together, we find that

$$\mathbb{P}(A_j|B_i^c) = \frac{d_j}{1-p_i d_i}$$

Alternate method to compute the denominator $\mathbb{P}(B_i^c)$: We could use the law of total probability to decompose $\mathbb{P}(B_i^c)$ depending on which drawer the homework is actually in. We have

$$\begin{aligned}
\mathbb{P}(B_i^c) &= \sum_{k=1}^{m} \underbrace{\mathbb{P}(A_k)}_{d_k} \underbrace{\mathbb{P}(B_i^c|A_k)}_{\substack{1 \text{ if } k\neq i,\\ (1-p_i) \text{ if } k=i}} \\
&= \sum_{\substack{k=1,\\ k\neq i}}^{m} d_k + (1-p_i)d_i \\
&= \sum_{k=1}^{m} d_k - p_i d_i \\
&= 1 - p_i d_i.
\end{aligned}$$

(b) Similarly, we'll use Bayes' rule:

$$\mathbb{P}(A_i|B_i^c) = \frac{\mathbb{P}(B_i^c|A_i)\mathbb{P}(A_i)}{\mathbb{P}(B_i^c)} = \frac{(1-p_i)d_i}{1-p_i d_i}$$

Take-Away Lessons:

- Defining the sample-space is not always going to help solve the problem. (It's difficult to precisely define the sample space for this particular problem)
- When in doubt of being able to precisely define the sample space, try to define events intelligently, i.e., in a way that you use what you're given in the problem.
- The probability law of a probability model is a function on events, or subsets of the sample space, i.e., one can work with the probability law without knowing precisely what the sample-space (as a set) is.

1.6 Inference with Bayes' Theorem for Random Variables

1.6.1 Moving Toward Bayes' Theorem for Random Variables

We introduced inference in the context of random variables, where there was a simple way to visualize what was going on in terms of joint probability tables. Marginalization referred to summing out rows or columns. Conditioning referred to taking a slice of the table and renormalizing so entries within that slice summed to 1. We then saw a more general story in terms of events. In fact, we saw that for many inference problems, using random variables to solve the problem is not necessary – reasoning with events was enough! A powerful tool we saw was Bayes' theorem.

We now return to random variables and build up to Bayes' theorem for random variables. This machinery will be extremely important as it will be how we automate inference for much larger problems in the later sections of the course, where we can have a large number of random variables at play, and a large amount of observations that we need to incorporate into our inference.

1.6.2 The Product Rule for Random Variables (Also Called the Chain Rule)

In many real world problems, we aren't given what the joint distribution of two random variables is although we might be given other information from which we can compute the joint distribution. Often times, we can compute out the joint distribution using what's called the product rule (often also called the chain rule). This is precisely the random variable version of the product rule for events.

As we saw from before, we were able to derive Bayes' theorem for events using the product rule for events: $\mathbb{P}(\mathscr{A} \cap \mathscr{B}) = \mathbb{P}(\mathscr{A})\mathbb{P}(\mathscr{B} \mid \mathscr{A})$. The random variable version of the product rule is derived just like the event version of the

product rule, by rearranging the equation for the definition of conditional probability. For two random variables X and Y (that take on values in sets $\mathscr{X}$ and $\mathscr{Y}$ respectively), the product rule for random variables says that

$$p_{X,Y}(x,y) = p_Y(y)p_{X|Y}(x \mid y) \qquad \text{for all } x \in \mathscr{X}, y \in \mathscr{Y} \text{ such that } p_Y(y) > 0.$$

Interpretation: If we have the probability table for Y, and separately the probability table for X conditioned on Y, then we can come up with the joint probability table (i.e., the joint distribution) of X and Y.

What happens when $p_Y(y) = 0$? Even though $p_{X|Y}(x \mid y)$ isn't defined in this case, one can readily show that $p_{X,Y}(x,y) = 0$ when $p_Y(y) = 0$.

To see this, think about what is happening computationally: Remember how $p_Y(y)$ is computed from joint probability table $p_{X,Y}$? In particular, we have $p_Y(y) = \sum_x p_{X,Y}(x,y)$, so $p_Y(y)$ is the sum of either a row or a column in the joint probability table (whether it's a row or column just depends on how you write out the table and which random variable is along which axis along rows or columns). So if $p_Y(y) = 0$, it must mean that the individual elements being summed are 0 (since the numbers we're summing up are nonnegative).

We can formalize this intuition with a proof:

Claim: Suppose that random variables X and Y have joint probability table $p_{X,Y}$ and take on values in sets $\mathscr{X}$ and $\mathscr{Y}$ respectively. Suppose that for a specific choice of $y \in \mathscr{Y}$, we have $p_Y(y) = 0$. Then

$$p_{X,Y}(x,y) = 0 \qquad \text{for all } x \in \mathscr{X}.$$

Proof: Let $y \in \mathscr{Y}$ satisfy $p_Y(y) = 0$. Recall that we relate marginal distribution p_Y to joint distribution $p_{X,Y}$ via marginalization:

$$0 = p_Y(y) = \textstyle\sum_{x \in \mathscr{X}} p_{X,Y}(x,y).$$

Next, we use a crucial mathematical observation: If a sum of nonnegative numbers (such as probabilities) equals 0, then each of the numbers being summed up must also be 0 (otherwise, the sum would be positive!). Hence, it must be that each number being added up in the right-hand side sum is 0, i.e.,

$$p_{X,Y}(x,y) = 0 \qquad \text{for all } x \in \mathscr{X}.$$

This completes the proof. □

Thus, in general:

$$p_{X,Y}(x,y) \quad = \quad \begin{cases} p_Y(y)p_{X|Y}(x \mid y) & \text{if } p_Y(y) > 0, \\ 0 & \text{if } p_Y(y) = 0. \end{cases}$$

Important convention for this course: For notational convenience, throughout this course, we will often just write $p_{X,Y}(x,y) = p_Y(y)p_{X|Y}(x \mid y)$ with the understanding that if $p_Y(y) = 0$, even though $p_{X|Y}(x \mid y)$ is not actually defined, $p_{X,Y}(x,y)$ just evalutes to 0 anyways.

The product rule is symmetric: We can use the definition of conditional probability with X and Y swapped, and rearranging factors, we get:

$$p_{X,Y}(x,y) = p_X(x)p_{Y|X}(y \mid x) \qquad \text{for all } x \in \mathscr{X}, y \in \mathscr{Y} \text{ such that } p_X(x) > 0,$$

and so similarly we could show that

$$p_{X,Y}(x,y) = \begin{cases} p_X(x)p_{Y|X}(y \mid x) & \text{if } p_X(x) > 0, \\ 0 & \text{if } p_X(x) = 0. \end{cases}$$

Again for notational convenience, we'll typically just write $p_{X,Y}(x,y) = p_X(x)p_{Y|X}(y \mid x)$ with the understanding that the expression is 0 when $p_Y(y) = 0$.

Interpretation: If we're given the probability table for X and, separately, the probability table for Y conditioned on X, then we can come up with the joint probability table for X and Y.

Importantly, for any two jointly distributed random variables X and Y, the product rule is always true, without making any further assumptions! Also, as a recurring theme that we'll see later on as well, we are decomposing the joint distribution into the product of factors (in this case, the product of two factors).

Many random variables: If we have many random variables, say, X_1, X_2, up to X_N where N is not a random variable but is a fixed constant, then we have

$$\begin{aligned} &p_{X_1,X_2,\ldots,X_N}(x_1,x_2,\ldots,x_N) \\ &= p_{X_1}(x_1)p_{X_2|X_1}(x_2 \mid x_1)p_{X_3|X_1,X_2}(x_3 \mid x_1,x_2) \\ &\quad \cdots p_{X_N|X_1,X_2,\ldots,X_{N-1}}(x_N \mid x_1,x_2,\ldots,x_{N-1}). \end{aligned}$$

Again, we write this to mean that this holds for every possible choice of $x_1,x_2,\ldots,x_N$ for which we never condition on a zero probability event. Note that the above factorization always holds without additional assumptions on the distribution of $X_1,X_2,\ldots,X_N$.

Note that the product rule could be applied in arbtirary orderings. In the above factorization, you could think of it as introducing random variable X_1 first, and then X_2, and then X_3, etc. Each time we introduce another random variable, we have to condition on all the random variables that have already been introduced.

Since there are N random variables, there are $N!$ different orderings in which we can write out the product rule. For example, we can think of introducing the last random variable X_N first and then going backwards until we introduce X_1 at the end. This yields the, also correct, factorization

$$\begin{aligned} &p_{X_1,X_2,\ldots,X_N}(x_1,x_2,\ldots,x_N) \\ &= p_{X_N}(x_N)p_{X_{N-1}|X_N}(x_{N-1} \mid x_N)p_{X_{N-2}|X_{N-1},X_N}(x_{N-2} \mid x_{N-1},x_N) \\ &\quad \cdots p_{X_1|X_2,X_3,\ldots,X_N}(x_1 \mid x_2,\ldots,x_N). \end{aligned}$$

1.6.3 Bayes' Rule for Random Variables (Also Called Bayes' Theorem for Random Variables

In inference, what we want to reason about is some unknown random variable X, where we get to observe some other random variable Y, and we have some model for how X and Y relate. Specifically, suppose that we have some "prior" distribution p_X for X; this prior distribution encodes what we believe to be likely or unlikely values that X takes on, before we actually have any observations. We also suppose we have a "likelihood" distribution $p_{Y|X}$.

After observing that Y takes on a specific value y, our "belief" of what X given $Y = y$ is now given by what's called the "posterior" distribution $p_{X|Y}(\cdot \mid y)$. Put another way, we keep track of a probability distribution that tells us how plausible we think different values X can take on are. When we observe data Y that can help us reason about X, we proceed to either upweight or downweight how plausible we think different values X can take on are, making sure that we end up with a probability distribution giving us our updated belief of what X can be.

Thus, once we have observed $Y = y$, our belief of what X is changes from the prior p_X to the posterior $p_{X|Y}(\cdot \mid y)$.

Bayes' theorem (also called Bayes' rule or Bayes' law) for random variables explicitly tells us how to compute the posterior distribution $p_{X|Y}(\cdot \mid y)$, i.e., how to weight each possible value that random variable X can take on, once we've observed $Y = y$. Bayes' theorem is the main workhorse of numerous inference algorithms and will show up many times throughout the course.

Bayes' theorem: Suppose that y is a value that random variable Y can take on, and $p_Y(y) > 0$. Then

$$p_{X|Y}(x \mid y) = \frac{p_X(x)p_{Y|X}(y|x)}{\sum_{x'} p_X(x')p_{Y|X}(y|x')}$$

for all values x that random variable X can take on.

Important: Remember that $p_{X|Y}(\cdot \mid y)$ could be undefined but this isn't an issue since this happens precisely when $p_X(x) = 0$, and we know that $p_{X,Y}(x,y) = 0$ (for every y) whenever $p_X(x) = 0$.

Proof: We have

$$p_{X|Y}(x \mid y) \overset{(a)}{=} \frac{p_{X,Y}(x,y)}{p_Y(y)} \overset{(b)}{=} \frac{p_X(x)p_{Y|X}(y|x)}{p_Y(y)} \overset{(c)}{=} \frac{p_X(x)p_{Y|X}(y|x)}{\sum_{x'} p_{X,Y}(x',y)} \overset{(d)}{=} \frac{p_X(x)p_{Y|X}(y|x)}{\sum_{x'} p_X(x')p_{Y|X}(y|x')},$$

where step (a) uses the definition of conditional probability (this step requires $p_Y(y) > 0$), step (b) uses the product rule (recall that for notational convenience we're not separately writing out the case when $p_X(x) = 0$), step (c) uses the formula for marginalization, and step (d) uses the product rule (again, for notational convenience, we're not separately writing out the case when $p_X(x') = 0$). □

1.6.4 Bayes' Theorem for Random Variables: A Computational View

Computationally, Bayes' theorem can be thought of as a two-step procedure. Once we have observed $Y = y$:

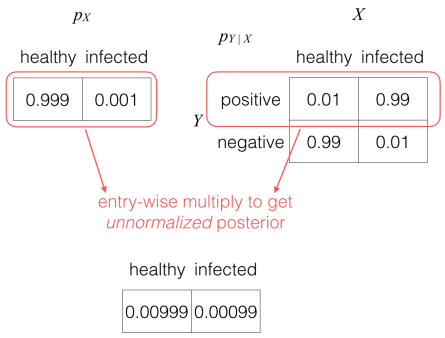
For each value x that random variable X can take on, initially we believed that $X = x$ with a score of $p_X(x)$, which could be thought of as how plausible we thought ahead of time that $X = x$. However now that we have observed $Y = y$, we weight the score $p_X(x)$ by a factor $p_{Y|X}(y \mid x)$, so

$$\text{new belief for how plausible } X = x \text{ is:} \quad \alpha(x \mid y) \triangleq p_X(x)p_{Y|X}(y \mid x),$$

where we have defined a new table $\alpha(\cdot \mid y)$ which is not a probability table, since when we put in the weights, the new beliefs are no longer guaranteed to sum to 1 (i.e., $\sum_x \alpha(x \mid y)$ might not equal 1)! $\alpha(\cdot \mid y)$ is an unnormalized posterior distribution!

Also, if $p_X(x)$ is already 0, then as we already mentioned a few times, $p_{Y|X}(y \mid x)$ is undefined, but this case isn't a problem: no weighting is needed since an impossible outcome stays impossible.

To make things concrete, here is an example from the medical diagnosis problem where we observe $Y = \textit{positive}$:



We fix the fact that the unnormalized posterior table $\alpha(\cdot \mid y)$ isn't guaranteed to sum to 1 by renormalizing:

$$p_{X|Y}(x \mid y) = \frac{\alpha(x|y)}{\sum_{x'} \alpha(x'|y)} = \frac{p_X(x)p_{Y|X}(y|x)}{\sum_{x'} p_X(x')p_{Y|X}(y|x')}.$$

An important note: Some times we won't actually care about doing this second renormalization step because we will only be interested in what value that X takes on is more plausible relative to others; while we could always do the renormalization, if we just want to see which value of x yields the highest entry in the unnormalized table $\alpha(\cdot \mid y)$, we could find this value of x without renormalizing!

1.6.5 Maximum A Posteriori (MAP) Estimation

For a hidden random variable X that we are inferring, and given observation $Y = y$, we have been talking about computing the posterior distribution $p_{X|Y}(\cdot|y)$ using Bayes' rule. The posterior is a distribution for what we are inferring. Often times, we want to report which particular value of X actually achieves the highest posterior probability, i.e., the most probable value x that X can take on given that we have observed $Y = y$.

The value that X can take on that maximizes the posterior distribution is called the *maximum a posteriori* (MAP) estimate of X given $Y = y$. We denote the MAP estimate by $\widehat{x}_{\text{MAP}}(y)$, where we make it clear that it depends on what the observed y is. Mathematically, we write

$$\widehat{x}_{\text{MAP}}(y) = \arg\max_x p_{X|Y}(x|y).$$

Note that if we didn't include the "arg" before the "max", then we would just be finding the highest posterior probability rather than which value – or "argument" – x actually achieves the highest posterior probability.

In general, there could be ties, i.e., multiple values that X can take on are able to achieve the best possible posterior probability.

1.7 Independence Structure

1.7.1 Independent Events

When you flip a coin or roll dice, the outcome of a coin flip or a die roll isn't going to tell you anything about the outcome of a new coin toss or die roll unless you have some very peculiar coins or dice.

we're going to formalize this by saying that two events A and B are independent, which we'll denote by this thing that looks like an upside down T: $\perp\!\!\!\perp$

$$A \perp\!\!\!\perp B \qquad \text{if} \qquad \mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B)$$

If $\mathbb{P}(A) > 0$ we can use the product rule for events to rewrite the left side:

$$\mathbb{P}(B \mid A)\mathbb{P}(A) = \mathbb{P}(A)\mathbb{P}(B)$$

Cancelling $\mathbb{P}(A)$ on both sides:

$$A \perp\!\!\!\perp B \qquad \text{if} \qquad \mathbb{P}(B \mid A) = \mathbb{P}(B)$$

Similarly, if $\mathbb{P}(B) > 0$

$$A \perp\!\!\!\perp B \qquad \text{if} \qquad \mathbb{P}(A \mid B) = \mathbb{P}(A)$$

So knowing B doesn't tell us anything new about A, and *vice versa*.

1.7.2 Independent Random Variables

X and Y are independent ($X \perp\!\!\!\perp Y$) if $p_{X,Y}(x,y) = p_X(x)p_Y(y)$.

Knowing one gives no information about the other, so $p_{X|Y}(x \mid y) = p_X(x)$.

Exercise: Independent Random Variables In this exercise, we look at how to check if two random variables are independent in Python. Please make sure that you can follow the math for what's going on and be able to do this by hand as well.

Consider random variables W, I, X, and Y, where we have shown the joint probability tables $p_{W,I}$ and $p_{X,Y}$.

W \ I	1	0
sunny	1/2	0
rainy	0	1/6
snowy	0	1/3

X \ Y	1	0
sunny	1/4	1/4
rainy	1/12	1/12
snowy	1/6	1/6

In Python:

```
prob_W_I = np.array([[1/2, 0], [0, 1/6], [0, 1/3]])
```

Note that here, we are not explicitly storing the labels, but we'll keep track of them in our heads. The labels for the rows (in order of row index): sunny, rainy, snowy. The labels for the columns (in order of column index): 1, 0.

We can get the marginal distributions p_W and p_I:

```
prob_W = prob_W_I.sum(axis=1)
prob_I = prob_W_I.sum(axis=0)
```

Then if W and I were actually independent, then just from their marginal distributions p_W and p_I, we would be able to compute the joint distribution with the formula:

$$\text{If } W \text{ and } I \text{ are independent:} \qquad p_{W,I}(w,i) = p_W(w)p_I(i) \qquad \text{for all } w,i.$$

Note that variables `prob_W` and `prob_I` at this point store the probability tables p_W and p_I as 1D NumPy arrays, for which NumPy does *not* store whether each of these should be represented as a row or as a column.

We could however ask NumPy to treat them as column vectors, and in particular, taking the outer product of `prob_W` and `prob_I` yields what the joint distribution would be if W and I were independent:

$$\begin{bmatrix} p_W(\text{sunny}) \\ p_W(\text{rainy}) \\ p_W(\text{snowy}) \end{bmatrix} \begin{bmatrix} p_I(1) & p_I(0) \end{bmatrix} = \begin{bmatrix} p_W(\text{sunny})p_I(1) & p_W(\text{sunny})p_I(0) \\ p_W(\text{rainy})p_I(1) & p_W(\text{rainy})p_I(0) \\ p_W(\text{snowy})p_I(1) & p_W(\text{snowy})p_I(0) \end{bmatrix}.$$

The left-hand side is an outer product, and the right-hand side is precisely the joint probability table that would result if W and I were independent.

To compute and print the right-hand side, we do:

```
print(np.outer(prob_W, prob_I))
```

1.7.3 Mutual vs Pairwise Independence

To extend the independence story to more than two variables, the strongest way is by using something called "mutual independence". We'll say that three random variables X, Y, and Z are mutually independent if we can write the joint distribution as simply the product of the three individual distributions:

$$p_{X,Y,Z}(x,y,z) = p_X(x)p_Y(y)p_Z(z)$$

Knowing X and Y won't tell you anything about Z. Knowing X won't tell you anything about Y. They're completely independent.

There are also weaker forms of independence that we're interested in, for example, "pairwise independence". This means that for any two variables, you can write:

$$p_{X,Y}(x,y) = p_X(x)p_Y(y)$$

and similarly for Y,Z and X,Z. This is saying if I know any one, it doesn't tell me think anything about any of the others. But this is not the same as mutual independence, it's not as strong.

As an example of why, suppose that we have two random variables X and Y. They both represent independent fair coin flips. We'll write the outcomes as 0 and 1, and each one has a 50-50 chance of being heads or tails, 0 or 1.

We'll define $Z = X \oplus Y$, where "XOR" (written $\oplus$) is defined to be a function that takes in two things and returns 1 when exactly one of them is 1:

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

What's the probability $p_{X,Y}$ for each of these configurations? They're independent fair coin flips, so each is equally likely. The probability of any particular one is 0.5 for X times 0.5 for Y, so 0.25:

$p_{X,Y}$	x	y	z
0.25	0	0	0
0.25	0	1	1
0.25	1	0	1
0.25	1	1	0

What's the distribution for Z? There are two ways to get $z = 0$ and they each have probability 0.25. If we add them up then we have a 0.5 chance of z being 0 and similarly a 0.5 chance of z being 1.

$$p_Z(z) = \begin{cases} 0.5 & \text{if } z = 0 \\ 0.5 & \text{if } z = 1 \end{cases}$$

What's Z given X? Well, it's actually the same. If $x = 0$, then we can restrict ourselves to just looking at the top two rows of the table, so Z can either be 0 or 1. And, again, they're equally likely so it's 50-50. If $x = 1$, we can just look at the bottom two rows, and again they're equally likely, 50-50.

$$p_{Z|X}(z \mid x) = \begin{cases} 0.5 & \text{if } z = 0 \\ 0.5 & \text{if } z = 1 \end{cases}$$

So $p_{Z|X}(z \mid x) = p_Z(z)$, it doesn't depend on X at all. So this means that $Z \perp\!\!\!\perp X$.

By symmetry, we can make the same argument for $Z \perp\!\!\!\perp Y$, and we said at the start that $X \perp\!\!\!\perp Y$. So here we have three random variables that are all pairwise independent — if you look at any pair, knowing one doesn't tell you about the other.

But if we look all together, they're not mutually independent. For example, if I know any two of them, then I know exactly what the third one is going to be. The distribution over the third one changes from being fair 50-50 to being deterministic. For example, if X is 0 and Y is 0, then Z is also going to be 0. And if I didn't know that, then the probability would be 50-50. Once I do know that, the probability becomes 1 that it's 0 and 0 that it's anything else. So here, they're not mutually independent. In this example it may seem a little contrived, but in general, it's often tempting to assume that knowing things are pairwise independent tells you that they're mutually independent. But you have to be aware that they're not.

1.7.4 Conditional Independence

We say that two random variables X and Y are conditionally independent given the third random variable Z if we can write the conditional distribution for both of them as the product of the individual conditionals:

$$p_{X,Y|Z}(x,y \mid z) = p_{X|Z}(x \mid z)p_{Y|Z}(y \mid z)$$

Intuitively, this means that once we know Z, then knowing something about Y doesn't tell you anything about X, and *vice versa*.

Notice that marginal independence and conditional independence are not the same thing. So if you have marginal independence, then that does not necessarily imply conditional independence. And the reverse is also not true. So two random variables X and Y could be independent but not conditionally independent given something else. Or two random variables could be conditionally independent but not marginally independent given something else.

As an example that illustrates this, suppose we have three random variables R, S, and T. We'll ask two questions: (a) is $S \perp\!\!\!\perp T$? (b) is $S \perp\!\!\!\perp T \mid R$? There are two different ways of writing the joint distribution:

$$\begin{aligned}(a)p_{R,S,T}(r,s,t) &= p_R(r)p_{S|R}(s \mid r)p_{T|R}(t \mid r)\\ (b)p_{R,S,T}(r,s,t) &= p_S(s)p_T(t)p_{R|S,T}(r \mid s,t)\end{aligned}$$

Part (a)**:** To determine whether two things are independent we just write out the distribution and see if it factors — that is, see if we can write it as something that only depends on S times something that only depends on T. If we can, then the former is $p_S(s)$, the latter is $p_T(t)$.

To compute the distribution of $p_{S,T}(s,t)$ from $p_{R,S,T}(r,s,t)$, we just have to sum over every possible value of r:

$$\begin{aligned}p_{S,T}(s,t) &= \sum_r p_{R,S,T}(r,s,t)\\ &= \sum_r p_R(r)p_{S|R}(s \mid r)p_{T|R}(t \mid r)\\ &\neq p_S(s)p_T(t)\end{aligned}$$

In general, we will not be able to factor the result because it will be a sum of different things that depend on S and T in different ways. So therefore, S and T are not independent.

Are they conditionally independent given R? To compute that, we have to see whether $p_{S,T}(s,t) \mid R = p_{S|R}(s \mid r)p_{T|R}(t \mid r)$. To compute a conditional distribution, remember that we just have to divide by the distribution of whatever we're conditioning on:

$$\begin{aligned}p_{S,T|R}(s,t \mid r) &= \frac{p_{R,S,T}(r,s,t)}{p_R(r)}\\ &= \frac{p_R(r)p_{S|R}(s \mid r)p_{T|R}(t \mid r)}{p_R(r)}\\ &= p_{S|R}(s \mid r)p_{T|R}(t \mid r)\end{aligned}$$

Canceling $p_R(r)$ leaves us with exactly the two terms we want. So the joint conditional distribution becomes the product of the two individual conditionals. So S and T are, in fact, conditionally independent given R.

Part (b)**:** We have that

$$p_{R,S,T}(r,s,t) = p_S(s)p_T(t)p_{R|S,T}(r \mid s,t)$$

As in part (a), if we want to compute whether S and T are independent, we have to look at the joint distribution and see whether we can write it as a product of the two individual distributions.

Again, to compute the joint distribution we have sum over every possible r:

$$\begin{aligned} p_{S,T}(s,t) &= \sum_r p_S(s)p_T(t)p_{R|S,T}(r \mid s,t) \\ &= p_S(s)p_T(t)\sum_r p_{R|S,T}(r \mid s,t) \\ &= p_S(s)p_T(t) \end{aligned}$$

As $p_S(s)$ and $p_T(t)$ don't depend on r they can be factored outside the summation. But no matter what S and T are, if we sum over every possible value of r, then $\sum_r p_{R|S,T}(r \mid s,t)$ is just going to give us 1 because it is just a probability distribution over R, and all probability distributions have to sum to 1. Hence $p_{S,T}(s,t) = p_S(s)p_T(t)$, and that's exactly the condition for independence. So in this case, $S \perp\!\!\!\perp T$.

What about conditional independence? As in part (a), we have to look at the conditional distribution $p_{S,T|R}(s,t \mid r)$ and see whether it factors.

$$p_{S,T|R}(s,t \mid r) = \frac{p_S(s)p_T(t)p_{R|S,T}(r|s,t)}{p_R(r)}$$

If we try to factor this, we'll see that we run into trouble when we get to the term $p_{R|S,T}(r \mid s,t)p_R(r)$ because in general, this is not going to factor. We don't know how S and T interact here, and there are no simplifications we can do to make this term go away. So here, $S \not\perp\!\!\!\perp T \mid R$.

We've seen from the last two examples that sometimes we can have marginal independence without conditional independence, and that sometimes we can have conditional independence without marginal independence. So it's important to keep in mind that these two are not always the same thing, and knowing one doesn't necessarily tell you the other.

1.7.5 Explaining Away

Let's look at an example with conditional probability. Suppose we have three events, R, A, and T, where

- R is the event that a Red Sox game
- A is the event there's an accident downtown
- T is the event there is unusually bad traffic

The chance of having a game and the chance of having an accident are each independently 50:50 :

$$\begin{aligned} p_R(r) &= \begin{cases} 0.5 & r = 1 \\ 0.5 & r = 0. \end{cases} \\ p_A(a) &= \begin{cases} 0.5 & a = 1 \\ 0.5 & a = 0. \end{cases} \end{aligned}$$

Whether or not there is traffic depends on whether there's a game and whether there's an accident. This table shows the probability that T is 1 for each configuration of r and a:

r	**a**	$p_{T\|R,A}(1 \mid r,a)$
0	0	0.3
0	1	0.9
1	0	0.9
1	1	0.3

So with no game and no accident, the probability of having bad traffic is 0.3. With either or both, then the probability goes up to 0.9. The table shows us the distribution for $T = 1$. The distribution for $T = 0$ would just have 1 minus this in every entry.

We want to calculate three probabilities:
(a) $\mathbb{P}(R = 1)$
(b) $\mathbb{P}(R = 1 \mid T = 1)$
(b) $\mathbb{P}(R = 1 \mid T = 1, A = 1)$

Part (a) is easy, as $\mathbb{P}(R = 1)$ is given to us as 0.5. But in (b) and (c), we're asked to find information about R given T, whereas the problem setup gives us information about T in terms of R. So we'll use Bayes' rule to turn the conditioning around. In both cases, we're only interested in the case where $T = 1$, so we'll not worry about the full distribution for now.

Bayes' rule tells us that

$$p_{R,A|T}(r,a \mid 1) = \frac{p_{T|R,A}(1|r,a)p_R(r)p_A(a)}{p_T(1)}$$

If we look carefully, we'll see that, in this case, because R and A are independent 50-50, $p_R(r)$ and $p_A(a)$ are going to be the same for every combination of r and a. Similarly, the denominator is also going to be the same for every r and a. So we can take a shortcut and say that these terms together are equal to some constant C:

$$p_{R,A|T}(r,a \mid 1) = C \cdot p_{T|R,A}(1 \mid r,a)$$

The table for $p_{T|R,A}(1 \mid r,a)$ is shown above, and we know that $p_{R,A|T}(r,a \mid 1)$ is some constant times this table. The probabilities are 0.3, 0.9, 0.9, 0.9. Except we know that they have to sum to 1. So if we take everything and divide by their sum, we get the normalized distribution of r and a given $t = 1$.

		a	
		0	1
r	0	0.1	0.3
	1	0.3	0.3

In part (b), we're asking for $\mathbb{P}(R = 1 \mid T = 1)$. So in this case, we don't care about a, we only care about r being 1 and we're marginalizing over A. So from the bottom row of the normalized distribution, $\mathbb{P}(R = 1 \mid T = 1) = 0.3 + 0.3 = 0.6$.

In part (b), we're asking for $\mathbb{P}(R = 1 \mid T = 1, A = 1)$. If we condition on $a = 1$, then we're restricting ourselves to the right column:

		a
		1
r	0	0.3
	1	0.3

The probability that $r = 1$ is 0.3 divided by the marginal probability of the column, that is 0.3 divided by 0.6. It occurs half the time because it's equally likely to be 0 or 1. So here our probability goes back down to 0.5. Intuitively, why is this? Why does our probability change? When we had more information the first time, in (b), $T = 1$, our probability went up from 0.5 to 0.6. But when we added even more information, then it went back down.

If we go back and look at the scenario, what this is asking is only given that there is traffic, what's the probability that there was a game? Well, it's a little higher in (b) because we know games cause traffic. So if there was traffic, then it's reasonable to assume that there was a game. So the probability goes up. But we know that both games and accidents cause traffic. So if we know that there was traffic and we know there was an accident, in (c), then it's more likely that the traffic was caused by the accident. So this is called explaining away, where once we observe one explanation, that is the accident, our belief in a different explanation, the Red Sox game, goes back down.

1.7.6 Practice Problem: Conditional Independence

Suppose $X_0, \ldots, X_{100}$ are random variables whose joint distribution has the following factorization:

$$p_{X_0,\ldots,X_{100}}(x_0,\ldots,x_{100}) = p_{X_0}(x_0) \cdot \prod_{i=1}^{100} p_{X_i|X_{i-1}}(x_i|x_{i-1})$$

This factorization is what's called a Markov chain. We'll be seeing Markov chains a lot more later on in the course.

Show that $X_{50} \perp X_{52}|X_{51}$.

Solution: Notice that we can marginalize out x_{100} as such:

$$p_{X_0,\ldots,X_{99}}(x_0,\ldots,x_{99}) = p_{X_0}(x_0) \cdot \prod_{i=1}^{99} p_{X_i|X_{i-1}}(x_i|x_{i-1}) \cdot \underbrace{\sum_{x_{100}} p_{X_{100}|X_{99}}(x_{100}|x_{99})}_{=1} \tag{2.5}$$

Now we can repeat the same marginalization procedure to get:

$$p_{X_0,\ldots,X_{50}}(x_0,\ldots,x_{50}) = p_{X_0}(x_0) \cdot \prod_{i=1}^{50} p_{X_i|X_{i-1}}(x_i|x_{i-1})$$

In essence, we have shown that the given joint distribution factorization applies not just to the last random variable (X_{100}), but also up to any point in the chain.

For brevity, we will now use $p(x_i^j)$ as a shorthand for $p_{X_i,\ldots,X_j}(x_i,\ldots,x_j)$. We want to exploit what we have shown to rewrite $p(x_{50}^{52})$

$$\begin{aligned}
p(x_{50}^{52}) &= \sum_{x_0\ldots x_{49}} \sum_{x_{53}\ldots x_{100}} p(x_0^{100}) \\
&= \sum_{x_0\ldots x_{49}} \sum_{x_{53}\ldots x_{100}} \left[p(x_0) \prod_{i=0}^{50} p(x_i|x_{i-1}) \right] \cdot p(x_{51}|x_{50}) \cdot p(x_{52}|x_{51}) \cdot \prod_{i=53}^{100} p(x_i|x_{i-1}) \\
&= \sum_{x_0\ldots x_{49}} \sum_{x_{53}\ldots x_{100}} p(x_0^{50}) \cdot p(x_{51}|x_{50}) \cdot p(x_{52}|x_{51}) \cdot \prod_{i=53}^{100} p(x_i|x_{i-1}) \\
&= p(x_{51}|x_{50}) \cdot p(x_{52}|x_{51}) \cdot \sum_{x_0\ldots x_{49}} p(x_0^{50}) \underbrace{\sum_{x_{53}\ldots x_{100}} \prod_{i=53}^{100} p(x_i|x_{i-1})}_{=1} \\
&= p(x_{51}|x_{50}) \cdot p(x_{52}|x_{51}) \cdot \sum_{x_0\ldots x_{49}} p(x_0^{50}) \\
&= p(x_{50}) \cdot p(x_{51}|x_{50}) \cdot p(x_{52}|x_{51})
\end{aligned}$$

where we used (2.5) for the 3rd equality and the same marginalization trick for the 5th equality. We have just shown the Markov chain property, so the conditional independence property must be satisfied.

1.8 Decisions and Expectations

1.8.1 Introduction to Decision Making and Expectations

We now know the basics of working with probabilities. But how do we incorporate probabilities into making decisions?

Let's make this concrete. Suppose we are given the option of playing one of the following lotteries. Which one should we play?

- Lottery 1: Pay $1 and have a one in one million chance of winning $1000.
- Lottery 2: Pay $1 and have a one in one million chance of winning $1000000.
- Lottery 3: Pay $1 and have a one in ten chance of winning $10.

Of course, there isn't a right or wrong answer here, but what we would like to do is come up with some quantitative way to make a decision. Especially if we want to make decisions automatically based on huge amount of observations, a principled quantitative approach is crucial!

One way to go about making a decision is to compute as score for each of the possible choices and choose the decision with the highest score. In this case of choosing between the three lotteries, for each of the lotteries, we could compute some kind of "average" amount of winnings, accounting for the cost to play.

For example, if we play the first lottery, we definitely lose $1, and then there is a $\frac{1}{1000000}$ chance that we win $1,000. So one way we could write out an "average" amount of winnings is something like

$$-1 + 1000 \cdot \tfrac{1}{1000000} = -1 + 0.001 = -0.999.$$

For now, multiplying the $1,000 by $\frac{1}{1000000}$ can be thought of as a heuristic that signifies that we aren't guaranteed to win $1,000, and how much we multiply by we're just picking to be the probability of winning right now.

Using the above way we have just come up with for computing "average" winnings, the second lottery has average winnings

$$-1 + \tfrac{1}{1000000}(1000000) = -1 + 1 = 0.$$

That's not so bad right? The chance of winning is still really low but the average winnings according to this calculation is 0, so it seems like we stand nothing to lose right?

Well, it depends on how much uncertainty we're willing to tolerate. A one in a million chance seems really low so almost always we would just lose a dollar if we play lottery #2.

In the third lottery, the average winnings is

$$-1 + \tfrac{1}{10}(10) = 0,$$

the same as for lottery #2. Sure, we aren't going to win $1000000 in this game but the chance of winning is way higher: 1/10 instead of 1/1000000. Somehow that should matter right?

On the basis of the average winnings calculation we have done though, lotteries #2 and #3 would be equally good as the give the same highest average winnings of $0.

We now discuss how to quantitatively and rigorously reason about scenarios like the ones we've just sketched. The main tool we now introduce is what's called the expected value of a random variable. In making decisions that account for randomness, it often makes sense to account for an "average" scenario that we should expect. Expectation is about taking an average, accounting for how likely different outcomes are.

In 6.008.1x, after we cover our story here on expected values, we won't be seeing them again until the third part of the course on learning probabilistic models. The idea there is that what probabilistic model makes sense for your data can be thought of as decision making! We are deciding which model to use instead of, in our example here, deciding which lottery to play!

1.8.2 The Expected Value of a Random Variable

Consider, for example, the mean of three values: 3, 5, and 10. It can be computed as follows:

$$\tfrac{3+5+10}{3} = 3 \cdot \tfrac{1}{3} + 5 \cdot \tfrac{1}{3} + 10 \cdot \tfrac{1}{3} = 6.$$

Notice, on the right-hand side, that we are adding 3, 5, and 10 each weighted by $\frac{1}{3}$. Concretely, consider a random variable X given by the probability table below:

		Probability
X	3	1/3
	5	1/3
	10	1/3

Then the "expected value" of X is given by

$$3 \cdot p_X(3) + 5 \cdot p_X(5) + 10 \cdot p_X(10) = 3 \cdot \tfrac{1}{3} + 5 \cdot \tfrac{1}{3} + 10 \cdot \tfrac{1}{3} = \tfrac{18}{3} = 6.\ldots$$

But what if, for instance, we think that 3 is actually much more plausible than 5 or 10? Then what we could do is have the weight on 3 be higher than $\frac{1}{3}$ while decreasing the weights for 5 and 10. Consider if instead we had:

		Probability
X	3	2/3
	5	1/6
	10	1/6

Then the expected value of X is given by

$$3 \cdot p_X(3) + 5 \cdot p_X(5) + 10 \cdot p_X(10) = 3 \cdot \tfrac{1}{3} + 5 \cdot \tfrac{1}{3} + 10 \cdot \tfrac{1}{3} = \tfrac{18}{3} = 6.\ldots$$

Using probability, we now formalize the concept of expected value of a random variable. As you can see, all we are doing is taking the sum of the labels in the probability table, where we weight each label by the probability of the label. *Importantly, the labels are numbers so that it's clear what adding them means!*

Now, for the formal definition:

Definition of expected value: Consider a real-valued random variable X that takes on values in a set $\mathscr{X}$. Then the expected value of X, denoted as $\mathbb{E}[X]$, is

$$\mathbb{E}[X] \triangleq \textstyle\sum_{x \in \mathscr{X}} x \cdot p_X(x).$$

Having the random variable be real-valued makes it so that we can add up the labels with weights!

Also, note that whereas X can be represented as a probability table, its expectation $\mathbb{E}[X]$ is just a single number. The expected value is the sum of the values in the set $\mathscr{X}$, weighted by the probabilities of each of the values. The mean is simply the expected value when all of the values in the set $\mathscr{X}$ when there is a uniform probability of each of the values.

Notice that how we came up with the expectation of a random variable X just relied on the probability table for X.

In fact, if we took a different probability table, if the labels are numbers, then we can still compute the expectation! Two important examples are below.

Conditional Expectation

As a first example, suppose we have two random variables X and Y where we know (or we have already computed) $p_{X|Y}(\cdot \mid y)$ for some fixed value y, and X is real-valued. Then we can readily compute the expectation for this probability table by multiplying each value x in the alphabet of random variable X by $p_{X|Y}(x \mid y)$ and summing these up to get a weighted average. This yields what is called the conditional expectation of X given $Y = y$, denoted as

$$\mathbb{E}[X \mid Y = y] = \sum_{x \in \mathscr{X}} x \cdot p_{X|Y}(x \mid y).$$

Expectation of the Function of a Random Variable

As another example, suppose we have a (possibly not real-valued) random variable X with probability table p_X, and we have a function f such that $f(x)$ is real-valued for all x in the alphabet $\mathscr{X}$ of X. Then $f(X)$ has a probability table where the labels are all numbers, and so we can compute $\mathbb{E}[f(X)]$.

Let's work out the math here. First, let's determine the probability table for $f(X)$. To make the notation here easier to parse, let random variable $Z = f(X)$. Note that Z has alphabet $\mathscr{Z} = \{f(x) : x \in \mathscr{X}\}$. Then the probability table for $f(X)$ can be written as p_Z. In terms of the probability table, to compute $p_Z(z)$, we first look at every label in table p_X that gets mapped to z, i.e., the set $\{x \in \mathscr{X} : f(x) = z\}$. Then we sum up the probabilities of these labels to get the probability that $Z = z$, i.e., $p_Z(z) = \sum_{x \in \mathscr{X} \text{ such that } f(x)=z} p_X(x)$.

We introduce a new piece of notation here called an indicator function $\mathbf{1}\{\cdot\}$ that takes as input a statement $\mathscr{S}$ and outputs:

$$\mathbf{1}\{\mathscr{S}\} = \begin{cases} 1 & \text{if } \mathscr{S} \text{ happens,} \\ 0 & \text{otherwise.} \end{cases}$$

Then the probability that $Z = z$ can be written

$$\begin{aligned} p_Z(z) &= \sum_{x \in \mathscr{X} \text{ such that } f(x)=z} p_X(x) \\ &= \sum_{x \in \mathscr{X}} \mathbf{1}\{f(x) = z\} p_X(x). \end{aligned}$$

Next, we compute the expectation of $Z = f(X)$:

$$\begin{aligned} \mathbb{E}[Z] &= \sum_{z \in \mathscr{Z}} z p_Z(z) \\ &= \sum_{z \in \mathscr{Z}} z \left[\sum_{x \in \mathscr{X}} \mathbf{1}\{f(x) = z\} p_X(x) \right] \\ &= \sum_{x \in \mathscr{X}} \underbrace{\sum_{z \in \mathscr{Z}} z \mathbf{1}\{f(x) = z\} p_X(x)}_{\text{there is only 1 nonzero term here: when } z = f(x)} \\ &= \sum_{x \in \mathscr{X}} f(x) p_X(x). \end{aligned}$$

Hence, since $Z = f(X)$, we can write

$$\mathbb{E}[f(X)] = \sum_{x \in \mathscr{X}} f(x) p_X(x).$$

1.8.3 Variance and Standard Deviation

The variance of a real-valued random variable X is defined as

$$\text{var}(X) \triangleq \mathbb{E}[(X - \mathbb{E}[X])^2].$$

Note that as we saw previously, $\mathbb{E}[X]$ is just a single number. To keep the variance of X, what you could do is first compute the expectation of X.

For example, if X takes on each of the values 3, 5, and 10 with equal probability 1/3, then first we compute $\mathbb{E}[X]$ to get 6, and then we compute $\mathbb{E}[(X-6)^2]$, where we remember to use the result that for a function f, if $f(X)$ is a real-valued random variable, then $\mathbb{E}[f(X)] = \sum_x f(x)p_X(x)$. Here, f is given by $f(x) = (x-6)^2$. So

$$\text{var}(X) = (3-6)^2 \cdot \tfrac{1}{3} + (5-6)^2 \cdot \tfrac{1}{3} + (10-6)^2 \cdot \tfrac{1}{3} = \tfrac{26}{3}.$$

1.8.4 Practice Problem: The Law of Total Expectation

Remember the law of total probability? For a set of events $\mathscr{B}_1, \ldots, \mathscr{B}_n$ that partition the sample space Ω (so the $\mathscr{B}_i$'s don't overlap and together they fully cover the full space of possible outcomes),

$$\mathbb{P}(\mathscr{A}) = \sum_{i=1}^n \mathbb{P}(\mathscr{A} \cap \mathscr{B}_i) = \sum_{i=1}^n \mathbb{P}(\mathscr{A} \mid \mathscr{B}_i)\mathbb{P}(\mathscr{B}_i),$$

where the second equality uses the product rule.

A similar statement is true for the expected value of a random variable, called the law of total expectation: for a random variable X (with alphabet $\mathscr{X}$) and a partition $\mathscr{B}_1, \ldots, \mathscr{B}_n$ of the sample space,

$$\mathbb{E}[X] = \sum_{i=1}^n \mathbb{E}[X \mid \mathscr{B}_i]\mathbb{P}(\mathscr{B}_i),$$

where

$$\mathbb{E}[X \mid \mathscr{B}_i] = \sum_{x \in \mathscr{X}} x p_{X|\mathscr{B}_i}(x) = \sum_{x \in \mathscr{X}} x \frac{\mathbb{P}(X=x, \mathscr{B}_i)}{\mathbb{P}(\mathscr{B}_i)}.$$

We will be using this result in the section "Towards Infinity in Modeling Uncertainty".

Show that the law of total expectation is true.

Solution: There are different ways to prove the law of total expectation. We take a fairly direct approach here, first writing everything in terms of outcomes in the sample space.

The main technical hurdle is that the events $\mathscr{B}_1, \ldots, \mathscr{B}_n$ are specified directly in the sample space, whereas working with values that X takes on requires mapping from the sample space to the alphabet of X.

We will derive the law of total expectation starting from the right-hand side of the equation above,
i.e., $\sum_{i=1}^n \mathbb{E}[X \mid \mathscr{B}_i]\mathbb{P}(\mathscr{B}_i)$.

We first write $\mathbb{E}[X \mid \mathscr{B}_i]$ in terms of a summation over outcomes in Ω:

$$\begin{aligned}
\mathbb{E}[X \mid \mathscr{B}_i] &= \sum_{x \in \mathscr{X}} x \frac{\mathbb{P}(X=x, \mathscr{B}_i)}{\mathbb{P}(\mathscr{B}_i)} \\
&= \sum_{x \in \mathscr{X}} x \frac{\mathbb{P}(\{\omega \in \Omega : X(\omega) = x\} \cap \mathscr{B}_i)}{\mathbb{P}(\mathscr{B}_i)} \\
&= \sum_{x \in \mathscr{X}} x \frac{\mathbb{P}(\{\omega \in \Omega : X(\omega) = x \text{ and } \omega \in \mathscr{B}_i\})}{\mathbb{P}(\mathscr{B}_i)} \\
&= \sum_{x \in \mathscr{X}} x \frac{\mathbb{P}(\{\omega \in \mathscr{B}_i : X(\omega) = x\})}{\mathbb{P}(\mathscr{B}_i)} \\
&= \sum_{x \in \mathscr{X}} x \cdot \frac{\sum_{\omega \in \mathscr{B}_i \text{ such that } X(\omega)=x} \mathbb{P}(\{\omega\})}{\mathbb{P}(\mathscr{B}_i)} \\
&= \frac{1}{\mathbb{P}(\mathscr{B}_i)} \sum_{x \in \mathscr{X}} x \sum_{\omega \in \mathscr{B}_i \text{ such that } X(\omega)=x} \mathbb{P}(\{\omega\}) \\
&= \frac{1}{\mathbb{P}(\mathscr{B}_i)} \sum_{\omega \in \mathscr{B}_i} X(\omega)\mathbb{P}(\{\omega\}).
\end{aligned}$$

Thus,

$$\begin{aligned}
\sum_{i=1}^{n} \mathbb{E}[X \mid \mathscr{B}_i]\mathbb{P}(\mathscr{B}_i) &= \sum_{i=1}^{n} \left(\frac{1}{\mathbb{P}(\mathscr{B}_i)} \sum_{\omega \in \mathscr{B}_i} X(\omega)\mathbb{P}(\{\omega\}) \right) \mathbb{P}(\mathscr{B}_i) \\
&= \sum_{i=1}^{n} \sum_{\omega \in \mathscr{B}_i} X(\omega)\mathbb{P}(\{\omega\}) \\
&= \sum_{\omega \in \Omega} X(\omega)\mathbb{P}(\{w\}) \\
&= \sum_{x \in \mathscr{X}} x\mathbb{P}(\{\omega \in \Omega \text{ such that } X(\omega) = x\}) \\
&= \sum_{x \in \mathscr{X}} x p_X(x) \\
&= \mathbb{E}[X].
\end{aligned}$$

1.9 Measuring Randomness

1.9.1 Introduction to Information-Theoretic Measures of Randomness

We just saw some basics for decision making under uncertainty and expected values of random variables. One way we saw for measuring uncertainty was variance. Now we look at a different way of measuring uncertainty or randomness using some ideas from information theory.

In this section, we answer the following questions in terms of bits (as in bits on a computer; everything stored on a computer is actually 0's and 1's each of which is 1 bit):

- How do we measure how random an event is?
- How do we measure how random a random variable or a distribution is?
- How do we measure how different two distributions are?
- How much information do two random variables share?

For now, this material may seem like a bizarre exercise relating to expectation of random variables, but as we will see in the third part of the course on learning probabilistic models, information theory provides perhaps the cleanest derivations for some of the learning algorithms we will derive!

More broadly but beyond the scope of 6.008.1x, information theory is often used to show what the best possible performance we should even hope an inference algorithm can achieve such as fundamental limits to how accurate we can make a prediction. And if you can show that your inference algorithm's performance meets the fundamental limit, then that certifies that your inference algorithm is optimal! Inference and information theory are heavily intertwined!

1.9.2 Shannon Information Content

First, let's consider storing an integer that isn't random. Let's say we have an integer that is from $0, 1, \ldots, 63$. Then the number of bits needed to store this integer is $\log_2(64) = 6$ bits: you tell me 6 bits and I can tell you exactly what the integer is.

A different way to think about this result is that we don't *a priori* know which of the 64 outcomes is going to be stored, and so each outcome is equally likely with probability $\frac{1}{64}$. Then the number of bits needed to store an event $\mathscr{A}$ is given by what's called the "Shannon information content" (also called self-information):

$$\log_2 \frac{1}{\mathbb{P}(\mathscr{A})}.$$

In particular, for an integer $x \in \{0, 1, \ldots, 63\}$, the Shannon information content of observing x is

$$\log_2 \frac{1}{\mathbb{P}(\text{integer is } x)} = \log_2 \frac{1}{1/64} = \log_2 64 = 6 \text{ bits}.$$

If instead, the integer was deterministically 0 and never equal to any of the other values $1,\ldots,63$, then the Shannon information content of observing integer 0 is

$$\log_2 \tfrac{1}{\mathbb{P}(\text{integer is } 0)} = \log_2 \tfrac{1}{1} = 0 \text{ bits.}$$

This is not surprising in that a outcome that we deterministically always observe tells us no new information. Meanwhile, for each integer $x \in \{0,1,\ldots,63\}$,

$$\log_2 \tfrac{1}{\mathbb{P}(\text{integer is } x)} = \log_2 \tfrac{1}{0} = \infty \text{ bits.}$$

How could observing one of the integers {1,2,...,63} tell us infinite bits of information? Well, this isn't an issue since the event that we observe any of these integers has probability 0 and is thus impossible. An interpration of Shannon information content is how surprised we would be to observe an event. In this sense, observing an impossible event would be infinitely surprising.

It is possible to have the Shannon information content of an event be some fractional number of bits (e.g., 0.7 bits). The interpretation is that from many repeats of the underlying experiment, the average number of bits needed to store the event is given by the Shannon information content, which can be fractional.

1.9.3 Shannon Entropy

To go from the number of bits contained in an event to the number of bits contained in a random variable, we simply take the expectation of the Shannon information content across the possible outcomes. The resulting quantity is called the entropy of a random variable:

$$H(X) = \sum_x p_X(x) \qquad \underbrace{\log_2 \frac{1}{p_X(x)}}_{\text{Shannon information content of event } X=x} \qquad .$$

The interpretation is that on average, the number of bits needed to encode each i.i.d. sample of a random variable X is $H(X)$. In fact, if we sample n times i.i.d. from p_X, then two fundamental results in information theory that are beyond the scope of this course state that: (a) there's an algorithm that is able to store these n samples in $nH(X)$ bits, and (b) we can't possibly store the sequence in fewer than $nH(X)$ bits!

Example: If X is a fair coin toss "heads" or "tails" each with probability 1/2, then

$$\begin{aligned} H(X) &= p_X(\text{heads}) \log_2 \frac{1}{p_X(\text{heads})} + p_X(\text{tails}) \log_2 \frac{1}{p_X(\text{tails})} \\ &= \frac{1}{2} \cdot \underbrace{\log_2 \tfrac{1}{\frac{1}{2}}}_{1} + \frac{1}{2} \cdot \underbrace{\log_2 \tfrac{1}{\frac{1}{2}}}_{1} \\ &= 1 \text{ bit.} \end{aligned}$$

Example: If X is a biased coin toss where heads occurs with probability 1 then

$$\begin{aligned} H(X) &= p_X(\text{heads}) \log_2 \frac{1}{p_X(\text{heads})} + p_X(\text{tails}) \log_2 \frac{1}{p_X(\text{tails})} \\ &= 1 \cdot \underbrace{\log_2 \tfrac{1}{1}}_{0} + 0 \cdot \cdot \underbrace{\log_2 \tfrac{1}{0}}_{1} \\ &= 0 \text{ bits,} \end{aligned}$$

where $0 \log_2 \frac{1}{0} = 0 \log_2 1 - 0 \log_2 0 = 0$ using the convention that $0 \log_2 0 \triangleq 0$. (Note: You can use l'Hopital's rule from calculus to show that $\lim_{x \to 0} x \log x = 0$ and $\lim_{x \to 0} x \log \frac{1}{x} = 0$.)

Notation: Note that entropy $H(X) = \sum_x p_X(x) \log_2 \frac{1}{p_X(x)}$ is in the form of an expectation! So in fact, we can write an expectation:

$$H(X) = \mathbb{E}\left[\log_2 \tfrac{1}{p_X(X)}\right].$$

1.9.4 Information Divergence

Information divergence (also called "Kullback-Leibler divergence" or "KL divergence" for short, or also "relative entropy") is a measure of how different two distributions p and q (over the same alphabet) are. To come up with information divergence, first, note that entropy of a random variable with distribution p could be thought of as the expected number of bits needed to encode a sample from p using the information content according to distribution p:

$$\underbrace{\sum_x p(x)}_{\text{expectation using } p} \qquad \underbrace{\log_2 \frac{1}{p(x)}}_{\text{information content according to } p} \qquad \triangleq \mathbb{E}_{X\sim p}\left[\log_2 \tfrac{1}{p(X)}\right].$$

Here, we have introduced a new notation: $\mathbb{E}_{X\sim p}$ means that we are taking the expectation with respect to random variable X drawn from the distribution p. If it's clear which random variable we are taking the expectation with respect to, we will often just abbreviate the notation and write $\mathbb{E}_p$ instead of $\mathbb{E}_{X\sim p}$.

If instead we look at the information content according to a different distribution q, we get

$$\underbrace{\sum_x p(x)}_{\text{expectation using } p} \qquad \underbrace{\log_2 \frac{1}{q(x)}}_{\text{information content according to } q} \qquad \triangleq \mathbb{E}_{X\sim p}\left[\log_2 \tfrac{1}{q(X)}\right].$$

It turns out that if we are actually sampling from p but encoding samples as if they were from a different distribution q, then we always need to use more bits! This isn't terribly surprising in light of the fundamental result we alluded to that entropy of a random variable with distribution p is the minimum number of bits needed to encode samples from p.

Information divergence is the price you pay in bits for trying to encode a sample from p using information content according to q instead of according to p:

$$D(p \parallel q) = \mathbb{E}_{X\sim p}\left[\log_2 \tfrac{1}{q(X)}\right] - \mathbb{E}_{X\sim p}\left[\log_2 \tfrac{1}{p(X)}\right].$$

Information divergence is always at least 0, and when it is equal to 0, then this means that p and q are the same distribution (i.e., $p(x) = q(x)$ for all x). This property is called Gibbs' inequality.

Gibbs' inequality makes information divergence seem a bit like a distance. However, information divergence is not like a distance in that it is not symmetric: in general, $D(p \parallel q) \neq D(q \parallel p)$.

Often times, the equation for information divergence is written more concisely as

$$D(p \parallel q) = \sum_x p(x) \log \tfrac{p(x)}{q(x)},$$

which you can get as follows:

$$\begin{aligned} D(p \parallel q) &= \mathbb{E}_{X\sim p}\Big[\log_2 \frac{1}{q(X)}\Big] - \mathbb{E}_{X\sim p}\Big[\log_2 \frac{1}{p(X)}\Big] \\ &= \sum_x p(x) \log_2 \frac{1}{q(x)} - \sum_x p(x) \log_2 \frac{1}{p(x)} \\ &= \sum_x p(x) \Big[\log_2 \frac{1}{q(x)} - \log_2 \frac{1}{p(x)}\Big] \\ &= \sum_x p(x) \log_2 \frac{p(x)}{q(x)}. \end{aligned}$$

Meanwhile, suppose q is a distribution for a biased coin that always comes up heads (perhaps it's double-headed):

$$q(x) = \begin{cases} 1 & \text{if } x = \text{heads}, \\ 0 & \text{if } x = \text{tails}. \end{cases}$$

Then

$$\begin{aligned} D(p \parallel q) &= p(\text{heads}) \log_2 \frac{p(\text{heads})}{q(\text{heads})} + p(\text{tails}) \log_2 \frac{p(\text{tails})}{q(\text{tails})} \\ &= \frac{1}{2} \log_2 \frac{\frac{1}{2}}{1} + \underbrace{\frac{1}{2} \log_2 \frac{\frac{1}{2}}{0}}_{\infty} \\ &= \infty \text{ bits}. \end{aligned}$$

This is not surprising: If we are sampling from p (for which we could get tails) but trying to encode the sample using q (which cannot possibly encode tails), then if we get tails, we are stuck: we can't store it! This incurs a penalty of infinity bits.

Meanwhile,

$$\begin{aligned} D(q \parallel p) &= q(\text{heads}) \log_2 \frac{q(\text{heads})}{p(\text{heads})} + q(\text{tails}) \log_2 \frac{q(\text{tails})}{p(\text{tails})} \\ &= 1 \log_2 \frac{1}{\frac{1}{2}} + \underbrace{0 \log_2 \frac{0}{\frac{1}{2}}}_{0} \\ &= 1 \text{ bit}. \end{aligned}$$

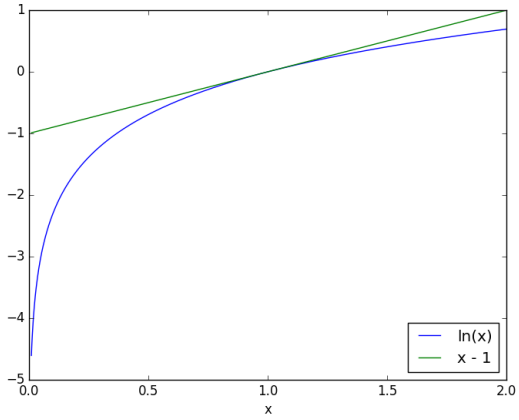
When we sample from q, we always get heads. In fact, as we saw previously, the entropy of the distribution for an always-heads coin flip is 0 bits since there's no randomness. But here we are sampling from q and storing the sample using distribution p. For a fair coin flip, encoding using distribution p would store each sample using on average 1 bit. Thus, even though a sample from q is deterministically heads, we store it using 1 bit. This is the penalty we pay for storing a sample from q using distribution p.

Notice that in this example, $D(p \parallel q) \neq D(q \parallel p)$. They aren't even close — one is infinity and the other is finite!

1.9.5 Proof of Gibbs' Inequality

We provide a proof for Gibbs' inequality here for those who are interested. For those of you up for the challenge, try to prove it yourself!

There are various ways to prove Gibbs' inequality. We'll be using a way that relies on the fact that $\ln x \leqslant x-1$ for all $x > 0$, with equality if and only if $x = 1$, which we provide a proof for at the end of this section, but for which you can also readily see from the following plot:



Code for producing this plot is also at the end of this section.

Gibbs' inequality: For any two distributions p and q defined over the same alphabet, we have $D(p \parallel q) \geqslant 0$, where equality holds if and only if p and q are the same distribution, i.e., $p(x) = q(x)$ for all x.

Proof: Recall that changing the base of a log just changes the log by a constant factor:

$$\log_2 x = \tfrac{\ln x}{\ln 2}.$$

Let $\mathscr{X}$ be the alphabet of distribution p restricted to where the probability is positive, i.e., $\mathscr{X} = \{a \text{ such that } p(a) > 0\}$. (There is no need to look at values a for which $p(a) = 0$.)

If $q(a) = 0$ for any $a \in \mathscr{X}$, then $D(p \parallel q) = \infty$, so trivially $D(p \parallel q) > 0$.

What's left to consider is when $q(a) > 0$ for every $a \in \mathscr{X}$. Then

$$\begin{aligned} D(p \parallel q) &= \textstyle\sum_{a\in\mathscr{X}} p(a) \log_2 \frac{p(a)}{q(a)} \\ &= \tfrac{1}{\ln 2} \textstyle\sum_{a\in\mathscr{X}} p(a) \ln \frac{p(a)}{q(a)} \\ &= -\tfrac{1}{\ln 2} \textstyle\sum_{a\in\mathscr{X}} p(a) \ln \frac{q(a)}{p(a)}. \end{aligned}$$

Next, using the fact that $\ln x \leqslant x-1$ for all $x > 0$, and accounting for the minus sign outside the summation,

$$\begin{aligned} D(p \parallel q) &= -\tfrac{1}{\ln 2} \textstyle\sum_{a\in\mathscr{X}} p(a) \ln \frac{q(a)}{p(a)} \\ &= -\tfrac{1}{\ln 2} \textstyle\sum_{a\in\mathscr{X}} p(a) \left(\frac{q(a)}{p(a)} - 1 \right) \\ &= -\tfrac{1}{\ln 2} \textstyle\sum_{a\in\mathscr{X}} \big(q(a) - p(a) \big) \\ &= -\tfrac{1}{\ln 2} \Big(\underbrace{\sum_{a\in\mathscr{X}} q(a)}_{1} - \underbrace{\sum_{a\in\mathscr{X}} p(a)}_{1} \Big) \\ &= 0 \end{aligned}$$

Recall that inequality $\ln x \leqslant x-1$ becomes an equality if and only if $x = 1$. Thus, the inequality above becomes an equality if and only if, for all $a \in \mathscr{X}$, we have $\ln \frac{q(a)}{p(a)} = \frac{q(a)}{p(a)} - 1$, which holds if and only if $\frac{q(a)}{p(a)} = 1$. Thus $D(p \parallel q) = 0$ if and only if $p(a) = q(a)$ for all $a \in \mathscr{X}$. This finishes the proof. ▫

Claim: $\ln x \leqslant x - 1$ for all $x > 0$ where equality holds if and only if $x = 1$.

Proof: We show that the function f given by $f(x) = x - 1 - \ln x$ is always at least 0 and achieves its minimum value at $x = 1$. First, note that f is differentiable for all $x > 0$ (which implies that f is continuous on $(0, \infty)$ and doesn't, for example, do some crazy jump midway through). In fact, the derivative of f is given by

$$\tfrac{d}{dx} f(x) = \tfrac{d}{dx}(x - 1 - \ln x) = 1 - \tfrac{1}{x}.$$

On the interval $x > 0$, the derivative is 0 (and so there's a local extremum) precisely when $x = 1$. The question is whether this is a local minimum or a local maximum. We look at the second derivative of f to do a second derivative test:

$$\tfrac{d^2}{dx^2} f(x) = \tfrac{1}{x^2},$$

which is strictly positive for all $x > 0$. In other words, $x = 1$ is a local minimum. The only possible other extrema could happen at the boundaries, but it's easy to check that

$$\lim_{x\to 0} f(x) = \lim_{x\to 0}(x - 1 - \ln x) = -1 - \underbrace{\lim_{x\to 0} \ln x}_{-\infty} = \infty,$$

and

$$\lim_{x\to \infty} f(x) = \lim_{x\to \infty}(x - 1 - \ln x) = \infty,$$

since x grows faster than $\ln x$.

Hence, f attains its global minimum at $x = 1$, for which we have

$$f(1) = 1 - 1 - \ln 1 = 0.$$

Since this is the global minimum, we know that $f(x) \geqslant 0$ for all $x > 0$. Furthermore, since there is only one unique global minimum $x = 1$, we further conclude that $f(x) = 0$ if and only if $x = 1$. ▫

Code to produce the plot at the top of this section:

```
plt.figure()
plt.plot(x, np.log(x))
plt.plot(x, x - 1)
plt.xlabel('x')
plt.legend(['ln(x)', 'x - 1'], loc=4)
plt.show()
```

1.9.6 Mutual Information

Mutual information: For two discrete random variables X and Y, the mutual information between X and Y, denoted as $I(X;Y)$, measures how much information they share. Specifically,

$$I(X;Y) \triangleq D(p_{X,Y} \parallel p_X p_Y),$$

where $p_X p_Y$ is the distribution we get if X and Y were actually independent (i.e., if X and Y were actually independent, then we know that the joint probability table would satisfy $\mathbb{P}(X = x, Y = y) = p_X(x) p_Y(y)$.

The mutual information could be thought of as how far X and Y are from being independent, since if indeed they were independent, then $I(X;Y) = 0$.

On the opposite extreme, consider when $X = Y$. Then we would expect X and Y to share the most possible amount of information. In this scenario, we can write $p_{X,Y}(x,y) = p_X(x)\mathbf{1}\{x = y\}$, and so

$$
\begin{aligned}
I(X;Y) &= D(p_{X,Y} \parallel p_X p_Y) \\
&= \sum_x \sum_y p_{X,Y}(x,y) \log_2 \frac{1}{p_X(x) p_Y(y)} \\
&\quad - \sum_x \sum_y p_{X,Y}(x,y) \log_2 \frac{1}{p_{X,Y}(x,y)} \\
&= \sum_x \sum_y p_X(x) \mathbf{1}\{x = y\} \log_2 \frac{1}{p_X(x) p_Y(y)} \\
&\quad - \sum_x \sum_y p_X(x) \mathbf{1}\{x = y\} \log_2 \frac{1}{p_X(x) \mathbf{1}\{x = y\}} \\
&= \sum_x p_X(x) \log_2 \left(\frac{1}{p_X(x)} \right)^2 - \sum_x p_X(x) \log_2 \frac{1}{p_X(x)} \\
&= 2 \sum_x p_X(x) \log_2 \frac{1}{p_X(x)} - \sum_x p_X(x) \log_2 \frac{1}{p_X(x)} \\
&= \sum_x p_X(x) \log_2 \frac{1}{p_X(x)} \\
&= H(X).
\end{aligned}
$$

This is not surprising: if X and Y are the same, then the number of bits they share is exactly the average number of bits needed to store X (or Y), namely $H(X)$ bits.

1.9.7 Exercise: Mutual Information

Consider the following joint probability table for random variables X and Y. We'll compute the mutual information $I(X;Y)$ of random variables X and Y step-by-step.

		Y = 0	1	2
X	0	0.10	0.09	0.11
	1	0.08	0.07	0.07
	2	0.18	0.13	0.17

Mutual information is about comparing the joint distribution of X and Y with what the joint distribution would be if X and Y were actually independent.

In Python (where we won't explicitly store the labels of the rows and columns):

```
import numpy as np
joint_prob_XY = np.array([[0.10, 0.09, 0.11], [0.08, 0.07, 0.07], [0.18, 0.13, 0.17]])
```

The marginal distributions p_X and p_Y are given by:

```
prob_X = joint_prob_XY.sum(axis=1)
prob_Y = joint_prob_XY.sum(axis=0)
```

Next, we produce what the joint probability table would be if X and Y were actually independent:

```
joint_prob_XY_indep = np.outer(prob_X, prob_Y)
```

At this point, we have the joint distribution of X and Y (denoted $p_{X,Y}$) stored in code as `joint_prob_XY`, and also what the joint distribution would be if X and Y were independent (denoted $p_X p_Y$) stored in code as `joint_prob_XY_indep`. The mutual information of X and Y is precisely given by the KL divergence between $p_{X,Y}$ and $p_X p_Y$:

$$I(X;Y) = D(p_{X,Y} \parallel p_X p_Y) = \sum_x \sum_y p_{X,Y}(x,y) \log_2 \frac{p_{X,Y}(x,y)}{p_X(x) p_Y(y)}.$$

1.9.8 Information-Theoretic Measures of Randomness: Where We'll See Them Next

In the third part of 6.008.1x when we talk about learning probabilistic models, at a basic level, what we have are observations (i.e., data we collect from the world), and our goal is to decide which probabilistic model in some sense "best" fits the observations. How we can decide on which probabilistic model to use is to give each candidate probabilistic model a score and then we pick the one with the highest score.

The score we will use is what's called "maximum likelihood", which is quite popular and has been extensively studied. By choosing to use maximum likelihood to decide on which candidate probabilistic model to use, very naturally entropy and information divergence will emerge! Importantly, information divergence will say how far a candidate model is from the observed data. Meanwhile, mutual information will come up to help us figure out which random variables we should directly model pairwise interactions with.

1.10 Towards Infinity in Modeling Uncertainty

1.10.1 Infinite Outcomes

What if we want an infinite number of outcomes? For example, consider an underlying experiment where we keep flipping a coin until we see the first heads. We might have to flip an arbitrarily large number of tosses! Here, the sample space would consist of getting heads for the first time after 1 toss, after 2 tosses, and so forth, ad infinitum.

On a computer, we can't actually store an arbitrary probability table with an infinite number of entries.

A few workarounds:

- Approximate the probability distribution with a finite probability space.
- In the above example, we could for example truncate and lump together all the possible outcomes in which heads appears after the 1000-th toss into a single possible outcome; we lose the ability to reason about the first heads appearing on, say, the 2000-th toss, but we could argue that heads appearing after the 1000th toss is extremely rare anyways.
- For very specific probability distributions, there can be a way for us to represent the distribution "in closed form" meaning that if we just keep track of a few numbers, these few numbers are enough to tell us what the probability is for any possible outcome in an infinitely large sample space.

Recall that the binomial distribution is an example of this: with just two numbers, we can query any entry in a probability table with far more than just two entries!

1.10.2 The Geometric Distribution

We now give an example of a distribution with an infinite alphabet size called the geometric distribution that has only 1 parameter. Thus, storing 1 number tells you what all the probability table entries are, even though there are an infinite number of entries!

Let's say I have some experiment that succeeds with some probability. How many times do I have to run independent trials of this experiment until I see the first success?

I have a biased coin, where if I toss it, the probability of heads $\mathbb{P}(H) = p$. I'm going to keep tossing this coin until I see the first heads. What are the possible outcomes? Well, it could be that I get heads in one toss, or I get tails and then I get heads, or I get 2 tails and then heads, and so forth. So our sample space is:

$$\Omega = \{H, TH, TTH, \dots\}$$

We're going to assign probabilities to each of these outcomes. The probability of heads in one toss is just $\mathbb{P}(H) = p$. The probability of tails and then heads? Since the tosses are independent,

$$\begin{aligned} \mathbb{P}(TH) &= \mathbb{P}(tails1st)\mathbb{P}(heads2nd) \\ &= (1-p)\cdot p \end{aligned}$$

Similarly, the probability of 2 tails and then a heads is going to be $\mathbb{P}(TH) = (1-p)^2 \cdot p$and so on.

But return to the original question, "how many tosses until the first heads?" We can phrase that in terms of a random variable X, the number of tosses until we see the first heads. The possible outcomes are, of course, $\mathscr{X} = \{1, 2, 3, \dots\}$, the set of positive integers. This is the alphabet of random variable X, the possible values that random variable X can take on.

As a reminder, a random variable is a function that maps from the sample space to this alphabet, $X : \Omega \to (X)$. X takes as input an element in the sample space, for example, heads. Then $X(H) = 1$ because this is heads in one toss. And similarly, Then $X(TH) = 2$, because we have two tosses until the first heads. And so forth.

So what is $\mathbb{P}(X = x)$? $X = x$ is shorthand:

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

For example, if $x = 1$, then we're looking at all possible outcomes in the sample space for which X of that outcome, is 1. So there's only one possible answer there, which is "H".

By doing this mapping, you can see that

$$\begin{aligned} \mathbb{P}(X = x) &= \mathbb{P}(x-1 tails followed by heads) \\ &= (1-p)^{x-1} \cdot p \qquad x = 1, 2, \dots \end{aligned}$$

This is called the probability mass function of random variable X. In particular, it is the probability mass function of a geometric random variable with parameter p. So we denote that by saying x is distributed as geometric with parameter p, $X \sim \text{Geo}(p)$. It is referred to as a geometric distribution because of the geometric decay in this probability as you increase x.

1.10.3 Practice Problem: The Geometric Distribution

Let $X \sim \text{Geo}(p)$ so that

$$p_X(x) = (1-p)^{x-1}p \qquad \text{for } x = 1, 2, \dots$$

- Show that each of the table entries $p_X(x)$ is nonnegative for $x = 1, 2, \dots$.

Solution: Note that $(1-p) \geqslant 0$, $x - 1 \geqslant 0$, and $p \geqslant 0$, and so $(1-p)^{x-1}p \geqslant 0$ for all $p \in (0, 1)$ and $x = 1, 2, 3, \dots$

- Show that the sum of all the table entries is 1, i.e., $\sum_{x=1}^{\infty} p_X(x) = 1$.

You may find the following result from calculus helpful: For $r \in (-1, 1)$,

$$\sum_{i=0}^{\infty} r^i = \tfrac{1}{1-r}.$$

Solution: For $p \in (0,1)$,

$$\sum_{x=1}^{\infty} p_X(x) = \sum_{x=1}^{\infty} (1-p)^{x-1} p \overset{(a)}{=} p \sum_{i=0}^{\infty} (1-p)^i \overset{(b)}{=} p \cdot \frac{1}{1-(1-p)} = p \cdot \frac{1}{p} = 1,$$

where step (a) substitutes $i = x - 1$, and step (b) uses the result above from calculus.

1.10.4 Discrete Probability Spaces and Random Variables

In the case of the geometric distribution, the sample space Ω is what's called "countably infinite". This means that it has an infinite (rather than finite) number of entries, and that there's actually a way for us to arrange the elements so that there's a 1st element, 2nd, 3rd, and so forth off into infinity. Note that the set of real numbers is not countable.

Before this section, every time we used the phrases "probability space" and "random variable", we actually meant "finite probability space" and "finite random variable".

More general than the finite probability space and finite random variable are the discrete probability space and discrete random variable:

Definition of a "discrete probability space": A discrete probability space $(\Omega, \mathbb{P})$ is the same thing as a finite probability space except that the sample space Ω is allowed to be either finite or countably infinite. In particular, a discrete probability space consists of two ingredients:

a finite or countably infinite sample space Ω that is the collectively exhaustive, mutually exclusive set of all possible outcomes

an assignment of probability $\mathbb{P}$, where for any outcome $\omega \in \Omega$, we have $\mathbb{P}(\text{outcome } \omega)$ be a number at least 0 and at most 1, and

$$\sum_{\omega \in \Omega} \mathbb{P}(\text{outcome } \omega) = 1.$$

Definition of a "discrete random variable": A discrete random variable X is the same thing as a finite random variable except that it's associated with a discrete probability space rather than a finite probability space. In particular, given a discrete probability space $(\Omega, \mathbb{P})$, a discrete random variable X maps Ω to a set of values $\mathscr{X}$ that the random variable can take on. Again, we can think of such a random variable X as generated from a two step procedure: some possible outcome ω is sampled from the discrete probability space $(\Omega, \mathbb{P})$, and then X takes on the value given by $X(\omega)$.

Formally defining random variables that are even more general than discrete random variables requires more sophisticated mathematical machinery that is beyond the scope of 6.008.1x.

Chapter 2

Inference in Graphical Models

2.1 Introduction

2.1.1 Introduction to Inference in Graphical Models

As we saw earlier, inference about hidden random variables from observations requires an exponential amount of computation. In this section, we will see that exploiting structure and probability distributions can lead to much more efficient inference.

Interestingly, the class of probability distributions that have that structure is quite broad and applicable to a wide range of real world problems. We will formally describe and characterize these probability distributions. Surprisingly, the complexity of the inference algorithms for this class of probability distributions becomes linear in the number of variables involved.

In this section, the core concept that we will introduce and use is graphical models. Graphical models combine graph theory with probability to capture structure in probability distributions. We will use graphical models to describe the structure, to reason about conditional independence relationships in these probability distributions, and to do derive efficient inference algorithms.

The algorithms you will see in the section are closely related to two famous algorithms. The first one is the Kalman filter. It was used to guide the Apollo mission, and is used today in many important engineering applications.

The second algorithm is the Viterbi algorithm. It was originally designed to decode messages in noisy communication networks. And today it is used in almost every wireless communication protocol.

2.2 Efficiency in Computer Programs

2.2.1 Big O Notation

We now give a primer on how to measure how fast and how much space a computer program takes. If we're solving large-scale inference problems, we want the inference to be fast and to be able to handle large probabilistic models (in terms of the number of random variables), which demands using computer memory wisely.

Of course, when it comes to storing probability tables, one could make the argument that there's no need to store all the probabilities. Instead, for a random variable taking on k values, without any known structure for the distribution (e.g., whether it is, for instance, a binomial distribution), then there are k probabilities, but we actually only need to keep track of $k-1$ of them since the last table entry is just going to be one minus the sum of all the other entries!

In talking about how much space a computer program needs to use, we don't want to worry about whether it's k numbers vs $k-1$ numbers. In both of these cases, what matters is that the computer program needs to store roughly

k numbers. In fact, we won't even care about whether it's k vs $2k$. What matters is that the amount of storage needed is roughly linear in k.

This same idea comes into play when we talk about how fast a computer program runs. We will count the number of "basic" operations such as variable assignment, addition, multiplication, and table look-ups. The idea is that each basic operation takes about the same fixed unit of time.

Let's take a look at a specific example:

Suppose we marginalizing out Y with a joint probability table $p_{X,Y}$, represented in a dictionaries-within-a-dictionary representation `p_XY`, to obtain the probability table for X stored as a dictionary `p_X`.

```
p_X = {}
for x in X_alphabet:
    total = 0
    for y in Y_alphabet:
        total = total + p_XY[x][y]
    p_X[x] = total
```

How many basic operations does a computer have to do when running this code? Well, let's suppose initializing a dictionary and assigning it to variable `p_X` takes 1 basic operation (it likely takes more due to initializing the dictionary but that's okay – we will be a bit sloppy with some constant factors here).

Next, the line `total = 0` happens $|\mathscr{X}|$ times, each time incurring a cost of 1 basic operation, so in total it costs $|\mathscr{X}|$ basic operations.

Next, the line `total = total + p_XY[x][y]` involves a table lookup, an addition, and a variable assignment so 3 basic operations (note that the table lookup might take more basic operations than just 1 to deal with the nested dictionaries), but since these 3 basic operations are nested in two for loops, we multiply by the number of times we're in the inner-most for loop: $3|\mathscr{X}\mathscr{Y}|$ basic operations.

Finally the line `p_X[x] = total` does a variable assignment and also indexes into a dictionary so let's say it costs 2 basic operations, which then gets multiplied by the number of times we reach that part in the outer for loop: $2|\mathscr{X}|$ basic operations.

Summing everything up, we get $1+3|\mathscr{X}|+3|\mathscr{X}||\mathscr{Y}|$ basic operations. Let's say $k = |\mathscr{X}||\mathscr{Y}|$. Then we have $1+3k+3k^2$ basic operations, which scales like k^2, the dominating term. We won't worry about the constants being a little off.

To formalize such rough measures of growth, we now introduce big O notation.

Big O notation: We say that a function f that depends on a variable n is $\mathscr{O}(g(n))$ (read as "big O of g of n") if there exists some minimum n_0 such that for all $n \geqslant n_0$,

$$f(n) \leqslant c \cdot g(n)$$

for some constant $c > 0$. Notationally, we write either $f(n) = \mathscr{O}(g(n))$ or $f(n) \in \mathscr{O}(g(n))$. Note that the $\mathscr{O}$ stands for "order" so you could think of f being $\mathscr{O}(g(n))$ as f growing on the order of g as a function of n.

Example: Storing the probability table for a random variable with alphabet size k takes $\mathscr{O}(k)$ amount of space. Even if we wanted to be efficient and store $k-1$ numbers, $k-1$ is still $\mathscr{O}(k)$, and in fact $k-1 \leqslant k$ for all k, satisfying the definition of big O notation (with n_0 being set to whatever we want, and c set to 1).

Example: With joint probability table $p_{X,Y}$ where X and Y each take on k values, then marginalization to compute p_X costs $\mathscr{O}(k^2)$ basic operations. For xample, using the way we counted the number of basic operations earlier, $1+3k+3k^2 \leqslant 6k^2$ for all $k \geqslant 2$, satisfying the definition of big O notation with $n_0 = 2$ and $c = 6$.

The basic idea is that when n is large enough (specifically larger than some n_0), then we'll always have that $f(n)$ grows at most as fast as $g(n)$ scaled by a constant that does not depend on n.

2.2.2 Big O Notation with Multiple Variables

Big O notation can be used with functions that depend on multiple variables.

Example: In our material coverage for Bayes' theorem for random variables, we saw in an exercise where we have n random variables that we want a posterior distribution over, where each of these random variables has alphabet size k. Then computing the denominator of Bayes' theorem involves summing over k^n table entries, a computation that takes running time $\mathcal{O}(k^n)$.

2.2.3 Important Remarks Regarding Big O Notation

In how big O notation is defined, it looks at an upper bound on how fast a function grows. For example, a function that grows linearly in n is$\mathcal{O}(n^2)$ and also $\mathcal{O}(2^n)$, both of which grow much faster than linear. Typically, when using big O notation such as $f(n) = \mathcal{O}(g(n))$, we will pick g that grows at a rate that matches or closely matches the actual growth rate of f.

An example of a mathematical operation for which people often use a function g that doesn't actually match f in order of growth is multiplying two n-by-n matrices, for which the fastest algorithm known takes time $\mathcal{O}(n^{2.373})$ but since the exponent 2.373 is peculiar and the algorithm that achieves it is quite complicated and not what's typically used in practice, often times for convenience people just say that multiplying two n-by-n matrices takes time $\mathcal{O}(n^3)$. Note that when they say such a statement, they are not explicitly saying which matrix multiplication algorithm is used either.

Finally, some terminology:

- If $f(n) = \mathcal{O}(1)$, then we say that f is constant with respect to input n.
- If $f(n) = \mathcal{O}(\log n)$, then we say that f grows logarithmically respect to input n. For example, if f is the running time of a computer program, then we would say that f has logarithmic running time in n.
- If $f(n) = \mathcal{O}(n)$, then we say that f grows linearly in n. For example, if f is the running time of a computer program, then we would say that f has linear running time in n.
- If $f(n) = \mathcal{O}(n^2)$, then we say that f grows quadratically in n. For example, if f is the running time of a computer program, then we would say that f has quadratic running time in n.
- If $f(n) = \mathcal{O}(b^n)$ for some constant $b > 1$, then we say that f grows exponentially in n. For example, if f is the running time of a computer program, then we would say that f has exponential running time in n.

The above are of course just a few examples. There are many other "categories" of functions such as $\mathcal{O}(n^3)$ corresponding to cubic growth in n.

In much of our discussion to follow in 6.008.1x, we will analyze either the "time complexity" (i.e., how long it takes code to run in terms of number of basic operations, in big O) or "space complexity" (i.e., how much space a code uses for storing variables, also in big O).

2.3 Graphical Models

2.3.1 Graphical Models

To introduce graphical models, we start with three simple examples.

Note that previously we have been using random variables X and Y, but here we will use X_1, X_2, up to X_n, as it will provide a clean way to write out the general case.

Graphical Model of Two Independent Random Variables

If X_1 and X_2 are independent random variables, then we know that $p_{X_1,X_2}(x_1,x_2) = p_{X_1}(x_1)p_{X_2}(x_2)$. Graphically, we represent this distribution as two circles. Because they are independent, we do not "connect" these two circles:

On a computer, we store a separate table for each of the two circles, one for p_{X_1} and one for p_{X_2}. Later we will see that tables that we store that depend only on a single random variable need not be a marginal distribution (such as in this example). Thus, we will give the tables new names. We let $\phi_1 = p_{X_1}$ and $\phi_2 = p_{X_2}$. (Later, ϕ_i is the table we store that is associated with a single random variable X_i. In this example, ϕ_1 and ϕ_2 are just set to be the same as the marginal distributions p_{X_1} and p_{X_2}.)

To summarize, the graphical model here includes the picture above (which is called a *graph*) along with the tables ϕ_1 and ϕ_2.

Two Possibly Dependent Random Variables

Now suppose that we do not know whether X_1 and X_2 are independent. Then without further assumptions, we can work directly with the joint probability table p_{X_1,X_2}, or we can use the product rule (which importantly always holds and does not require X_1 and X_2 to be independent).

Here are three different ways we can store the distribution:

- (a) Store p_{X_1,X_2}. If we do this, then there is a single table p_{X_1,X_2} that we are storing that depends on two random variables X_1 and X_2.

  We introduce new notation here: a table we store that depends on exactly two random variables X_i and X_j will be called $\psi_{i,j}$, so in this case, we store $\psi_{i,j} = p_{X_1,X_2}$. Later, we will see examples where $\psi_{i,j}$ is not a joint probability table for two random variables.

  There are no tables here that depend on exactly 1 random variable.

- (b) Store p_{X_1} and $p_{X_2|X_1}$.

  Here, there is one table that depends on exactly 1 random variable: p_{X_1}. We call this $\phi_{X_1} = p_{X_1}$.

  There is one table that depends on exactly 2 random variables: $\psi_{1,2} = p_{X_2|X_1}$. (This $\psi_{1,2}$ is different from the one in (a).)

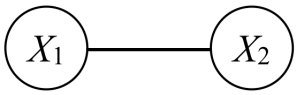
- (c) p_{X_2} and $p_{X_1|X_2}$.

  Here, there is one table that depends on exactly 1 random variable: p_{X_2}. We call this $\phi_{X_2} = p_{X_2}$.

  There is one table that depends on exactly 2 random variables: $\psi_{1,2} = p_{X_1|X_2}$. (This $\psi_{1,2}$ is different from the ones in (a) and (b).)

The tables being stored in each of the above three cases are different, but the joint probability distribution they describe is the same.

A common feature of all three different ways: there is a table that depends on both X_1 and X_2. For this reason, when we draw out a graphical representation in this case, we still have two circles, one for X_1 and one for X_2, but now we connect the two with a line:



Again, the line is there between X_1 and X_2 precisely because to store the associated joint probability table, regardless of which of the different ways we store the tables, we have to use a table that depends on both X_1 and X_2.

Markov Chain

Next, suppose we have three random variables X_1, X_2 and X_3, where we make an assumption here that the joint probability table has factorization

$$p_{X_1,X_2,X_3}(x_1,x_2,x_3) = p_{X_1}(x_1)p_{X_2|X_1}(x_2|x_1)p_{X_3|X_2}(x_3|x_2). \tag{3.1}$$

Note that this factorization is in general not true for three random variables X_1, X_2 and X_3.

To see, this, we can apply the product rule (which is true for any three random variables X_1, X_2 and X_3):

$$p_{X_1,X_2,X_3}(x_1,x_2,x_3) = p_{X_1}(x_1)p_{X_2|X_1}(x_2|x_1)p_{X_3|X_1,X_2}(x_3|x_1,x_2). \tag{3.2}$$

To see what the assumption that equation (3.1) holds means, we equate it with the equation from the product rule (3.2) to get

$$\begin{aligned}
&p_{X_1}(x_1)p_{X_2|X_1}(x_2|x_1)p_{X_3|X_2}(x_3|x_2) \\
&= p_{X_1,X_2,X_3}(x_1,x_2,x_3) \\
&= p_{X_1}(x_1)p_{X_2|X_1}(x_2|x_1)p_{X_3|X_1,X_2}(x_3|x_1,x_2),
\end{aligned}$$

from which we deduce that $p_{X_3|X_1,X_2}(x_3|x_1,x_2) = p_{X_3|X_2}(x_3|x_2)$. This means that given X_2, knowing X_1 does not tell us anything new about X_3:

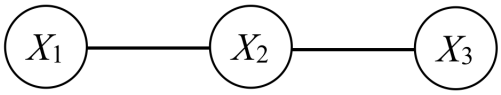
$$X_1 \perp X_3 \mid X_2.$$

So assuming equation (3.1) holds means that we are making an assumption about conditional independence structure!

Next, let's see how to store the distribution when it has factorization given by equation (3.1). We see that it suffices to keep track of three tables ϕ_1, $\psi_{1,2}$, and $\psi_{2,3}$:

$$p_{X_1,X_2,X_3}(x_1,x_2,x_3) = \underbrace{p_{X_1}(x_1)}_{\phi_1(x_1)} \underbrace{p_{X_2|X_1}(x_2|x_1)}_{\psi_{1,2}(x_1,x_2)} \underbrace{p_{X_3|X_1,X_2}(x_3|x_1,x_2)}_{\psi_{2,3}(x_2,x_3)}.$$

The graph associated with this representation has three circles, one for each of the random variables X_1, X_2 and X_3. We have a table $\psi_{1,2}$ that depends on X_1 and X_2 so we draw a line between the circles for X_1 and X_2. Next we have a table $\psi_{2,3}$ that depends on X_2 and X_3 so we draw a line between the circles for X_2 and X_3. This yields the following:



This line-shaped graph is called a Markov chain. We will encounter Markov chains more later on. Notationally, when X_1, X_2 and X_3 form a Markov chain, we write $X_1 \leftrightarrow X_2 \leftrightarrow X_3$.

The General Case

We are almost ready to mathematically define what a graphical model is. As we saw from the above examples, each time we had a graph (a picture with circles and possibly lines) along with tables that we store.

For a graph, the circles are formally called nodes or vertices, and the lines are formally called edges. The graph is

undirected because the edges do not have directionality associated with them. Furthermore:

Each node corresponds to a random variable

Each edge indicates there being some possible dependence between the two nodes that it connects. Specifically: an edge being present between two nodes X_i and X_j means that the equation for the probability distribution has a factor that depends on both X_i and X_j; this factor is stored as table $\psi_{i,j}$.

From these definitions, an important concept emerges: More edges implies that the model can encode more probability distributions but we have to store more tables. (Think about why the second example we presented for two random variables where we don't know whether they're independent or not (and had a factor $\psi_{1,2}$) can actually encode a distribution in which X_1 and X_2 are independent!)

Meanwhile, in the graphs that we will consider, we will assume that there are no loops. We do this for simplicity: probabilistic graphical models with loops are beyond the scope of this course.

Note that a graph is specified by saying what the nodes (circles) are, and what the lines (edges) are. To give specific examples:

- When X_1 and X_2 were independent, the graph we had consisted of the set of nodes $V = \{1,2\}$ and the set of edges $E = \varnothing$.
- When X_1 and X_2 were not known as to whether they are independent or not, the graph we had consisted of the set of nodes $V = \{1,2\}$ and the set of edges $E = \{(1,2)\}$. Note that in general we use (i,j) to mean an edge between nodes i and j, where (i,j) and (j,i) mean the same thing, which is why the set of the edges here includes only $(1,2)$and not both $(1,2)$ and $(2,1)$.
- When we have $X_1 \leftrightarrow X_2 \leftrightarrow X_3$, the set of nodes is $V = \{1,2,3\}$ and the set of edges is $E = \{(1,2),(2,3)\}$.

Definition of an undirected pairwise graphical model (we will just call this a graphical model): An *undirected pairwise graphical model* for random variables $X_1,\ldots,X_n$ consists of an undirected graph with vertices $V = \{1,\ldots,n\}$ and edges E, and tables ϕ_i's and $\psi_{i,j}$'s that have nonnegative entries. The joint probability table of $X_1,\ldots,X_n$ is given by

$$p_{X_1,\ldots,X_n}(x_1,\ldots,x_n) = \frac{1}{Z}\prod_{i\in V}\phi_i(x_i)\prod_{(i,j)\in E}\psi_{ij}(x_i,x_j),$$

where Z is the normalization constant that ensures that the probability distribution actually sums to 1 (we'll give a concrete example shortly).

Note that in earlier examples, we didn't always specify that a random variable X_i needed to have a table ϕ_i. This is not a problem: we can just set $\phi_i(x_i) = 1$ for all values of x_i in this case.

Terminology:

Each table ϕ_i depends only on random variable X_i and is called the node potential function or node potential of node i.

Each table $\psi_{i,j}$ depends only on random variables X_i and X_j and is called the pairwise potential function or pairwise potential or edge potential of nodes i and j.

Important: We require that the potential tables consist of nonnegative entries **but each potential table does not have to sum to 1**. The constant Z will ensure that the joint probability table actually sums to 1.

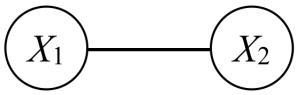
2.3.2 Trees

A tree is a graph for which there are no loops, and we can reach from any node to any other node (moving along edges in the graph). We'll be seeing trees quite a bit so here are some basics of trees.

Example:

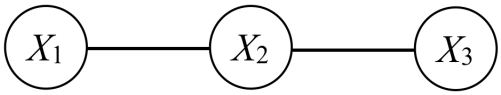
This graph is not a tree since there is no path from X_1 to X_2.

Example:



This graph is a tree since there are no loops and we can reach from any node to any other node.

Example:



This graph is a tree since there are no loops and we can reach from any node to any other node.

Theorem: For any graph that has n nodes, if the graph is a tree, then it will always have exactly $n-1$ edges.

Proof: We use induction.

Base case $n = 1$: There is only 1 node so there are no edges, so the claim clearly holds.

Inductive step: Suppose the claim holds for every tree of size (i.e., number of nodes) up to k. Thus, every tree of size k nodes has $k-1$ edges. Now consider a tree T with $k+1$ nodes. Take a leaf node v from T and note that the tree T with v removed is a tree T' of size k, which by the inductive hypothesis has $k-1$ edges. Since v is a leaf node though, it has exactly 1 neighbor, which means that the tree T has 1 more edge than the tree T', i.e., T has k edges. This finishes the inductive step. □

2.3.3 Practice Problem: Computing the Normalization Constant

It turns out that once we know the potential functions, the normalization constant Z becomes fixed since the distribution needs to sum to 1. Let's show this for a simple case. Consider a two node graphical model with an edge between the two nodes corresponding to

$$p_{X_1,X_2}(x_1,x_2) = \tfrac{1}{Z}\phi_1(x_1)\phi_2(x_2)\psi_{12}(x_1,x_2).$$

Suppose that we are given what the potential functions are. Show what Z is equal to as a function of ϕ_1, ϕ_2, and ψ_{12}.

Hint: Sum both sides over all values of x_1 and all values of x_2. What is $\sum_{x_1}\sum_{x_2} p_{X_1,X_2}(x_1,x_2)$ equal to?

Because knowing the potentials fixes what the value of Z is, often times we'll omit writing Z and instead write

$$p_{\underline{X}}(\underline{x}) \propto \prod_{i\in V}\phi_i(x_i)\prod_{(i,j)\in E}\psi_{ij}(x_i,x_j),$$

where "$\propto$" means "proportional to".

Solution: We have

$$\begin{aligned} 1 &= \sum_{x_1}\sum_{x_2} p_{X_1,X_2}(x_1,x_2) \\ &= \sum_{x_1}\sum_{x_2} \tfrac{1}{Z}\phi_1(x_1)\phi_2(x_2)\psi_{12}(x_1,x_2) \\ &= \tfrac{1}{Z}\sum_{x_1}\sum_{x_2} \phi_1(x_1)\phi_2(x_2)\psi_{12}(x_1,x_2), \end{aligned}$$

so

$$Z = \sum_{x_1}\sum_{x_2} \phi_1(x_1)\phi_2(x_2)\psi_{12}(x_1,x_2).$$

In general for a graphical model with graph $G = (V,E)$ and factorization

$$p_{X_1,\ldots,X_n}(x_1,\ldots,x_n) = \tfrac{1}{Z}\prod_{i\in V}\phi_i(x_i)\prod_{(i,j)\in E}\psi_{ij}(x_i,x_j),$$

using the same reasoning as above,

$$Z = \sum_{x_1}\cdots\sum_{x_n}\prod_{i\in V}\phi_i(x_i)\prod_{(i,j)\in E}\psi_{ij}(x_i,x_j).$$

2.4 Inference in Graphical Models - Marginalization

2.4.1 Two Fundamental Inference Tasks in Graphical Models

Now that we have introduced graphical models for representing probability distributions, we proceed to solving problems of inference. As we have seen, observations can be taken care of quite easily and once we account for them, we are left with a new graphical model that is only over the random variables that we don't get to observe, which are called "hidden" or "latent" random variables. Thus, our discussion to follow will be about graphical models where we don't explicitly mention conditioning, *with the idea that the conditioning has already happened!*

Given a graphical model with graph $G = (V,E)$ and its associated node and edge potentials, the two fundamental inference tasks we focus on are as follows:

- **(Marginalization)** Compute marginal probability table p_{X_i} for every $i \in V$.
- **(Most probable configuration)** Compute the most probable configuration $(\widehat{x}_1,\widehat{x}_2,\ldots,\widehat{x}_n)$ such that
$$(\widehat{x}_1,\widehat{x}_2,\ldots,\widehat{x}_n) = \arg\max_{x_1,x_2,\ldots,x_n} p_{X_1,X_2,\ldots,X_n}(x_1,x_2,\ldots,x_n).$$

Again, we are assuming conditioning has already happened. This means that if we observed random variable $Y = y$ and already accounted for it, then the graphical model that remains is only over random variables $X_1,\ldots,X_n$ and so in the first task above, the table p_{X_i} actually corresponds to $p_{X_i|Y}(\cdot \mid y)$ and the second task would compute the most probable configuration in the posterior distribution $p_{X_1,\ldots,X_n|Y}(\cdots \mid y)$. (Note that there was nothing special about us calling the random variables in a graphical model $X_1,\ldots,X_n$. Doing so was just to make the notation simple, but the random variables could be called whatever you want, including having X_i's and Y_i's, and also observing multiple random variables rather than just observing one random variable is handled the same way: fix the values for what has been observed, which corresponds to removing those nodes from the original graphical model and changing potential tables to account for the observations.)

We now focus on the inference task of marginalization in graphical models.

2.4.2 The Sum-Product Algorithm

What about graphical models that aren't tree-structured and that don't have loops? Think about why in such cases, the different disconnected pieces can be treated separately (and each of these pieces of the graph will be a tree)! For example, when computing the marginal distribution p_{x_i} for a specific node i, any node that isn't reachable from i doesn't matter in the computation. So we can always turn the problem into just looking at one tree at a time.

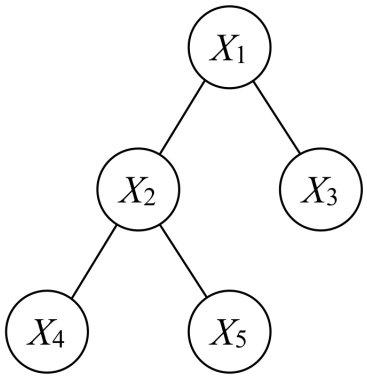
Marginalization: A Worked Example

Let's work through an example of how to marginalize in an undirected graphical model. This example will illustrate a general algorithm for marginalization, which we can write a computer program for.

Suppose random variables X_1, X_2, X_3, X_4, X_5 each take on values in the set $\mathscr{X}$ and have the following joint distribution:

$$\begin{aligned} p_{X_1,\ldots,X_n}(x_1,\ldots,x_n) &= \frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3)\phi_4(x_4)\phi_5(x_5) \\ &\quad \cdot \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\psi_{25}(x_2,x_5). \end{aligned}$$

This joint distribution corresponds to the following undirected graphical model:



Let's compute the marginal distribution p_{x_i}. This distribution is a table assigning a probability to each value of $\mathscr{X}$. We obtain the marginal distribution for X_1 by summing out the other random variables:

$$\begin{aligned} p_{X_1}(x_1) &= \sum_{x_2}\sum_{x_3}\sum_{x_4}\sum_{x_5} p_{X_1,\ldots,X_n}(x_1,\ldots,x_n) \\ &= \sum_{x_2}\sum_{x_3}\sum_{x_4}\sum_{x_5} \left\{ \frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3)\phi_4(x_4)\phi_5(x_5) \right. \\ &\qquad \left. \cdot \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\psi_{25}(x_2,x_5) \right\}. \end{aligned}$$

To compute the summation on the right-hand side efficiently on a computer, notice that we can push the summations around a bit taking advantage of the distributive property of arithmetic: $ab + ac = a(b+c)$ for any choice of constants a, b, c. For example, only two factors depend on x_5 (namely ϕ_5 and ψ_{25}) so we can push the summation over x_5 inward as follows:

$$\begin{aligned} p_{X_1}(x_1) &= \sum_{x_2}\sum_{x_3}\sum_{x_4}\sum_{x_5} \left\{ \frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3)\phi_4(x_4)\phi_5(x_5) \right. \\ &\qquad \left. \cdot \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\psi_{25}(x_2,x_5) \right\} \\ &= \sum_{x_2}\sum_{x_3}\sum_{x_4} \left\{ \frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3)\phi_4(x_4) \right. \\ &\qquad \left. \cdot \psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4) \underbrace{\sum_{x_5} \phi_5(x_5)\psi_{25}(x_2,x_5)}_{\triangleq m_{5\to 2}(x_2)} \right\}. \end{aligned}$$

Here we have introduced some notation: $m_{5\to 2}(x_2)$ is a value we get from summing out (and thus eliminating) x_5 where the result depends on x_2. Note that $m_{5\to 2}$ is a table: there is one entry in the table per value of $x_2 \in \mathscr{X}$. We use the letter "m" since we can interpret $m_{5\to 2}$ as a message that node 5 sends to node 2.

We can keep pushing sums around:

$$\begin{aligned}
p_{X_1}(x_1) &= \sum_{x_2}\sum_{x_3}\Bigg\{\frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3) \\
&\qquad \cdot\psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)m_{5\to 2}(x_2)\underbrace{\sum_{x_4}\phi_4(x_4)\psi_{24}(x_2,x_4)}_{\triangleq m_{4\to 2}(x_2)}\Bigg\} \\
&= \sum_{x_2}\sum_{x_3}\Bigg\{\frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3) \\
&\qquad \cdot\psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)m_{4\to 2}(x_2)m_{5\to 2}(x_2)\Bigg\} \\
&= \sum_{x_2}\Bigg\{\frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\psi_{12}(x_1,x_2) \\
&\qquad \cdot m_{4\to 2}(x_2)m_{5\to 2}(x_2)\underbrace{\sum_{x_3}\phi_3(x_3)\psi_{13}(x_1,x_3)}_{\triangleq m_{3\to 1}(x_1)}\Bigg\} \\
&= \frac{1}{Z}\phi_1(x_1)m_{3\to 1}(x_1)\underbrace{\sum_{x_2}\phi_2(x_2)\psi_{12}(x_1,x_2)m_{4\to 2}(x_2)m_{5\to 2}(x_2)}_{\triangleq m_{2\to 1}(x_1)} \\
&= \frac{1}{Z}\underbrace{\phi_1(x_1)m_{2\to 1}(x_1)m_{3\to 1}(x_1)}_{\triangleq \widetilde{p}_{X_1}(x_1)},
\end{aligned}$$

where in the last line, we define $\widetilde{p}_{X_1}(\cdot)$ to be a table that just corresponds to an unnormalized version of the marginal distribution of X_1. We can readily compute the normalization constant:

$$Z = \textstyle\sum_{x_1}\widetilde{p}_{X_1}(x_1)$$

Once we have computed this, we know the marginal distribution p_{X_1} for X_1.

Computing Another Marginal Distribution in the Graph

Now suppose that we want the marginal distribution p_{X_3} for X_3. We can approach solving this problem the same way as how we computed the marginal distribution p_{X_1} for X_1. But something nice happens, assuming that we've already computed the intermediate tables that we computed to obtain the marginal for X_1. In particular, we have

$$
\begin{aligned}
p_{X_3}(x_3) &= \sum_{x_1}\sum_{x_2}\sum_{x_4}\sum_{x_5}\Big\{\frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3)\phi_4(x_4)\phi_5(x_5) \\
&\qquad\qquad \cdot\psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\psi_{25}(x_2,x_5)\Big\} \\
&= \sum_{x_1}\sum_{x_2}\sum_{x_4}\Big\{\frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3)\phi_4(x_4) \\
&\qquad\qquad \cdot\psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)\psi_{24}(x_2,x_4)\quad \underbrace{\sum_{x_5}\phi_5(x_5)\psi_{25}(x_2,x_5)}_{m_{5\to 2}(x_2)\text{ - we already computed this}}\Big\} \\
&= \sum_{x_1}\sum_{x_2}\Big\{\frac{1}{Z}\phi_1(x_1)\phi_2(x_2)\phi_3(x_3) \\
&\qquad\qquad \cdot\psi_{12}(x_1,x_2)\psi_{13}(x_1,x_3)m_{5\to 2}(x_2)\quad \underbrace{\sum_{x_4}\phi_4(x_4)\psi_{24}(x_2,x_4)}_{m_{4\to 2}(x_2)\text{ - also already computed}}\Big\} \\
&= \sum_{x_1}\Big\{\frac{1}{Z}\phi_1(x_1)\phi_3(x_3) \\
&\qquad\qquad \cdot\psi_{13}(x_1,x_3)\underbrace{\sum_{x_2}\phi_2(x_2)\psi_{12}(x_1,x_2)m_{4\to 2}(x_2)m_{5\to 2}(x_2)}_{m_{2\to 1}(x_1)\text{ - already computed}}\Big\} \\
&= \frac{1}{Z}\phi_3(x_3)\underbrace{\sum_{x_1}\phi_1(x_1)\psi_{13}(x_1,x_3)m_{2\to 1}(x_1)}_{\triangleq m_{1\to 3}(x_3)} \\
&= \frac{1}{Z}\underbrace{\phi_3(x_3)m_{1\to 3}(x_3)}_{\triangleq \tilde{p}_{X_3}(x_3)}.
\end{aligned}
$$

The General Case: The Sum-Product Algorithm

We are now ready to state the general algorithm for computing marginal distributions for every node in an undirected graphical model. This algorithm specifically works when the corresponding graph is a tree, which means that it has no loops and we can reach from any node in the graph to any other node by traversing along edges. The resulting general algorithm for marginalization is called the *sum-product algorithm* (and popularly also goes by the name *belief propagation*):

- 1. Choose a root note (arbitrarily) and identify corresponding leaf nodes. (In our earlier example, we choose node 1 as the root node; the leaf nodes then are nodes 3, 4, and 5.)
- 2. Proceed from the leaf nodes to the root node computing required messages along the way. (In our earlier example, we computed messages in the order $m_{5\to 2}$, $m_{4\to 2}$, $m_{3\to 1}$, and $m_{2\to 1}$.)

  When the root node is reached, reverse direction and calculate messages that go back to the leaf nodes (In our earlier example, we then computed messages $m_{1\to 3}$ followed by $m_{1\to 2}$; to further get the marginals at X_4 and X_5 compute $m_{2\to 4}$ and $m_{2\to 5}$.)

  The equation for computing the table of messages is given by

$$m_{i\to j}(x_j) = \sum_{x_i}\left[\phi_i(x_i)\psi_{i,j}(x_i,x_j)\prod_{k\in\mathcal{N}(i)\text{ such that }k\neq j} m_{k\to i}(x_i)\right],$$

where $\mathcal{N}(i)$ denotes the neighboring nodes of node i in the graph.

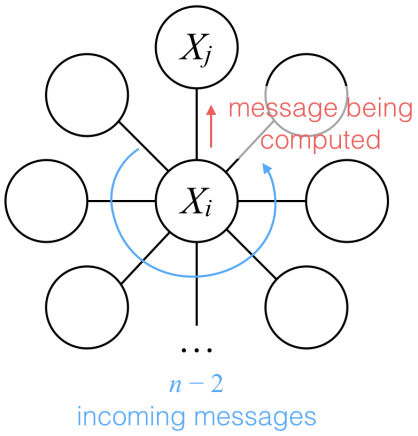
- 3. For each node i, use the incoming messages to compute the marginal distribution p_{X_i}:

$$p_{X_i}(x_i) = \frac{1}{Z}\underbrace{\phi_i(x_i)\prod_{j\in\mathcal{N}(i)} m_{j\to i}(x_i)}_{\tilde{p}_{X_i}(x_i)}.$$

  Typically, this marginal is computed by first computing the unnormalized table $\tilde{p}_{X_i}$ and then normalizing it; the normalization constant Z is not saved (although it could be).

2.4.3 Speeding Up Sum-Product

The sum-product algorithm naively implemented can take time that is more than linear in the number of nodes n. We can see this from the worst-case star graph:



In particular with node i referring to the center of the star graph, for every possible node j that is not i (there are $\mathcal{O}(n)$ choices for j), the computation of $m_{i\to j}$ requires taking the product of $\mathcal{O}(n)$ terms, so we get hit by a total running time that at least scales with n^2.

It turns out that with clever implementation, we can cut this running time down to linear in n. We walk through a way of doing this, which will treat the two passes of the sum-product slightly differently in terms of calculation. Note that this way of speeding up the calculation will require that the message tables always have strictly positive entries. There are ways around this assumption that still keeps the computation linear in n but to keep the exposition here simple we'll stick to strictly positive message table entries.

In what follows, let d_i denote the number of neighbors that node i has; in graph theory, d_i is called the degree of node i. As before, assume that every X_i takes on one of k possible values (note that even if the different X_i's took on a different number of values, we can take k to be the maximum alphabet size of any of the X_i's to get an upper bound).

Sum-Product: Message Passing from the Leaves to the Root

After choosing an arbitrary node to be the root of the tree, then we know what the leaves are, and every node (except the root node) has one unique parent, where if we look at any leaf node, take its parent node, then the parent node of that parent node, and so forth, eventually we will always reach the root node.

Let $\pi(i)$ denote the parent of non-root node i. Then when passing messages from leaves to the root node, the messages are of the form:

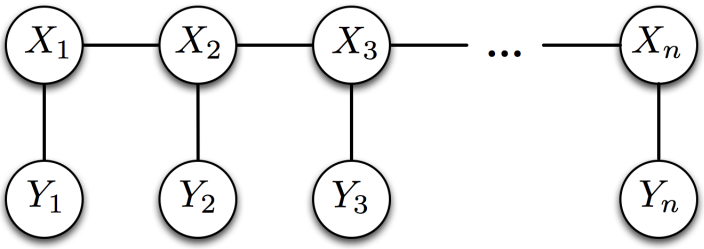
$$m_{i\to\pi(i)}(x_{\pi(i)}) = \sum_{x_i}\left[\psi_{i,\pi(i)}(x_i,x_{\pi(i)})\underbrace{\phi_i(x_i)\prod_{\ell\in\mathcal{N}(i)\text{ such that }\ell\neq\pi(i)} m_{\ell\to i}(x_i)}_{\text{define this to be }\xi_i(x_i)}\right],$$

The reason for defining a table ξ_i (which has one entry for every possible value in the alphabet of X_i) is that we will use ξ_i during message passing from the root node back to the leaves that will eliminate redundant calculation.

2.5 Special Case: Marginalization in Hidden Markov Models

2.5.1 Introduction to Hidden Markov Models (HMM's)

We now show the sum-product algorithm applied to the special case of a tree-structured graphical model called a hidden Markov model (HMM), for which we have hidden states $X_1, X_2, \ldots, X_n$ that we infer given observations $Y_1, Y_2, \ldots, Y_n$, and the graphical model looks as follows:



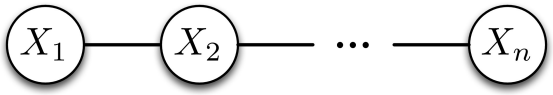
For example, in a basic robot localization problem setup, X_i is the robot's location at time point i, and Y_i is the robot's sensor readings at time point i. Then $p_{X_i|Y_1,\ldots,Y_n}(\cdot \mid y_1, \ldots, y_n)$ is the posterior distribution of the robot's location at time point i.

The graphical model is

$$\begin{aligned}
&p_{X_1,\ldots,X_n,Y_1,\ldots,Y_n}(x_1,\ldots,x_n,y_1,\ldots,y_n) \\
&\propto \textstyle\prod_{i=1}^{n} \left\{\phi_{X_i}(x_i)\phi_{Y_i}(y_i)\psi_{X_i,Y_i}(x_i,y_i)\right\} \prod_{i=1}^{n-1} \psi_{X_i,X_{i+1}}(x_i,x_{i+1}).
\end{aligned}$$

Note that by incorporating observations, we are left with a new graphical model that is only over the X_i's:

$$\begin{aligned}
&p_{X_1,\ldots,X_n|Y_1,\ldots,Y_n}(x_1,\ldots,x_n \mid y_1,\ldots,y_n) \\
&\propto \textstyle\prod_{i=1}^{n} \Big\{ \underbrace{\phi_{X_i}(x_i)\phi_{Y_i}(y_i)\psi_{X_i,Y_i}(x_i,y_i)}_{\triangleq \tilde{\phi}_i(x_i)} \Big\} \prod_{i=1}^{n-1} \underbrace{\psi_{X_i,X_{i+1}}(x_i,x_{i+1})}_{\triangleq \psi_{i,i+1}(x_i,x_{i+1})}.
\end{aligned}$$



We can run the sum-product algorithm choosing the right-most node X_n as the root, so the initial message passing happens from the single leaf node X_1 going rightward until reaching X_n (this is called the forward pass), and then we pass messages from the root node X_n all the way back to X_1 (this is called the backward pass). The resulting algorithm for this specific case of HMM's is called the forward-backward algorithm.

Hidden Markov models are widely used in practice for more than just robot localization (note that a robot could be, for instance, a self-driving car)! But a few other examples include recognizing speech, labeling the part of speech of each word in a sentence, aligning DNA sequences, detecting protein folds, and decoding a certain of code in digital communications called a convolutional code.

2.5.2 Hidden Markov Models: Three Ingredients

Often for HMM's, a problem is modeled with a prior distribution p_{X_1} on the initial state X_1, a transition model $p_{X_{i+1}|X_i}$ that is the same for all i (so we can denote it as just $p_{X_{\text{next}}|X_{\text{current}}}$), and an observation model $p_{Y_i|X_i}$ that is the

same for all i (so we can denote it as just $p_{Y|X}$), in which case

$$\begin{aligned}&p_{X_1,\ldots,X_n,Y_1,\ldots,Y_n}(x_1,\ldots,x_n,y_1,\ldots,y_n)\\&= p_{X_1}(x_1)\left\{\textstyle\prod_{i=1}^n p_{Y|X}(y_i \mid x_i)\right\}\left\{\textstyle\prod_{i=1}^{n-1} p_{X_{\text{next}}|X_{\text{current}}}(x_{i+1} \mid x_i)\right\}.\end{aligned}$$

Then as a graphical model, incorporating observations $Y_1 = y_1, \ldots, Y_n = y_n$,

$$\begin{aligned}&p_{X_1,\ldots,X_n|Y_1,\ldots,Y_n}(x_1,\ldots,x_n \mid y_1,\ldots,y_n)\\&\propto \underbrace{p_{X_1}(x_1)p_{Y|X}(y_1 \mid x_1)}_{\triangleq \widetilde{\phi}_1(x_1)} \textstyle\prod_{i=2}^n \left\{\underbrace{p_{Y|X}(y_i \mid x_i)}_{\triangleq \widetilde{\phi}_i(x_i)}\right\} \prod_{i=1}^{n-1} \underbrace{p_{X_{\text{next}}|X_{\text{current}}}(x_{i+1} \mid x_i)}_{\triangleq \psi_{i,i+1}(x_i,x_{i+1})}.\end{aligned}$$

The messages are:

$$\begin{aligned}&& m_{1\to 2}(x_2) &= \textstyle\sum_{x_1} \widetilde{\phi}_1(x_1)\psi_{1,2}(x_1,x_2)\\\text{For } i=2,\ldots,n-1: && m_{i\to i+1}(x_{i+1}) &= \textstyle\sum_{x_i} \widetilde{\phi}_i(x_i)\psi_{i,i+1}(x_i,x_{i+1})m_{i-1\to i}(x_i)\\&& m_{n\to n-1}(x_{n-1}) &= \textstyle\sum_{x_n} \widetilde{\phi}_n(x_n)\psi_{n-1,n}(x_{n-1},x_n)\\\text{For } i=2,\ldots,n-1: && m_{i\to i-1} &= \textstyle\sum_{x_i} \widetilde{\phi}_i(x_i)\psi_{i-1,i}(x_{i-1},x_i)m_{i+1\to i}(x_i)\end{aligned}$$

The posterior distributions for each X_i given all the observations $Y_1 = y_1, \ldots, Y_n = y_n$ are:

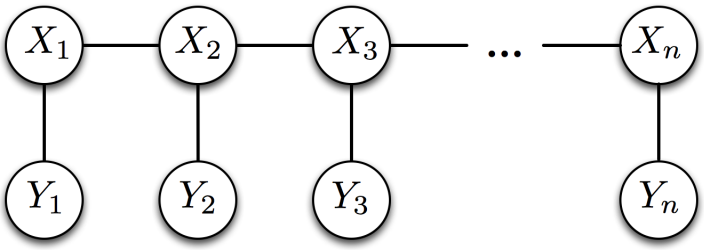
$$\begin{aligned}&p_{X_1|Y_1,\ldots,Y_n}(x_1 \mid y_1,\ldots,y_n) &\propto\ & \widetilde{\phi}_1(x_1)m_{2\to 1}(x_1)\\\text{For } i=2,\ldots,n-1: \quad &p_{X_i|Y_1,\ldots,Y_n}(x_1 \mid y_1,\ldots,y_n) &\propto\ & \widetilde{\phi}_i(x_i)m_{i-1\to i}(x_i)m_{i+1\to i}(x_i)\\&p_{X_n|Y_1,\ldots,Y_n}(x_1 \mid y_1,\ldots,y_n) &\propto\ & \widetilde{\phi}_n(x_n)m_{n-1\to n}(x_n)\end{aligned}$$

We will use these conventions:

- The forward messages are denoted by $\alpha_{i\to i+1}(x_{i+1}) \triangleq m_{i\to i+1}(x_{i+1})$
- The backward messages are denoted by $\beta_{i+1\to i}(x_i) \triangleq m_{i+1\to i}(x_i)$
- Since $\psi_{i,i+1}$ is actually the same table/function for all i, we just call it ψ, i.e., $\psi(x_i,x_{i+1}) = p_{X_{\text{next}}|X_{\text{current}}}(x_{i+1} \mid x_i)$
- Since $\widetilde{\phi}_i$ above is actually the same for all $i \geqslant 2$, we just call it $\widetilde{\phi}$, i.e., $\widetilde{\phi}(x_i) = p_{Y|X}(y_i \mid x_i)$; we actually extend this to the case $i = 1$ as well with the following notational convenience: we define $\alpha_{0\to 1}(x_1) \triangleq p_{X_1}(x_1)$, so that the forward messages now can be expressed in 1 formula:
$$\alpha_{i\to i+1}(x_{i+1}) = \textstyle\sum_{x_i} \widetilde{\phi}(x_i)\psi(x_i,x_{i+1})\alpha_{i-1\to i}(x_i) \qquad \text{for all } i = 1,2,\ldots,n-1$$
- As another notational convenience, we define $\beta_{n+1\to n}(x_n) = 1$ for all x_n, so that the backward messages now can be expressed in one formula:
$$\beta_{i\to i-1} = \textstyle\sum_{x_i} \widetilde{\phi}(x_i)\psi(x_{i-1},x_i)\beta_{i+1\to i}(x_i) \qquad \text{for all } i = 2,3,\ldots,n$$

Clarification: In terms of edges in the graphical model, any particular state variable X_k has edges only to X_{k-1} (what came before it), X_{k+1} (what comes after it), and Y_k (the observation associated with it). When we condition on all three of these, X_k becomes independent of everything else in the graph. *However, when we don't condition on anything, then from any node we can reach any other node, so at least from looking at the graph, it could very well be that nodes far apart depend on each other. Even if we condition on all the Y_i's and nothing else, in general, all the X_i's can be related to each other!*

As a reminder, here is the graphical model for an HMM:

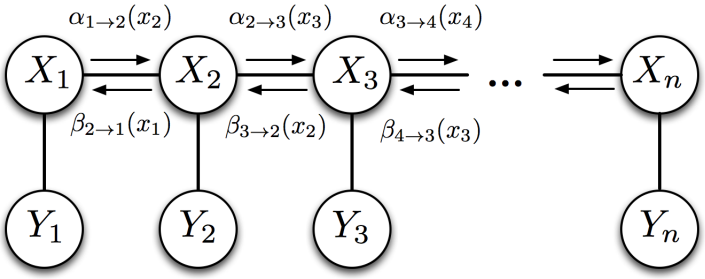


Here's a quick recap of the important facts:

- We observe Y_1 through Y_n, which we model as coming from hidden states X_1 through X_n
- The goal of the forward-backward algorithm is to find the conditional distribution over hidden states given the data.
- In order to specify an HMM, we need three quantities:
  - A *transition distribution*, $p_{X_{k+1}|X_k}(x_{k+1}|x_k)$, which describes the distribution for the next state given the current state. This is often represented as a matrix that we'll call A. Rows of A correspond to the current state, columns correspond to the next state, and each entry corresponds to the transition probability. That is, the entry at row i and column j, A_{ij}, is $p_{X_{k+1}|X_k}(j|i)$.
  - An *observation distribution* (also called an "emission distribution") $p_{Y_k|X_k}(y_k|x_k)$, which describes the distribution for the output given the current state. We'll represent this with matrix B. Here, rows correspond to the current state, and columns correspond to the observation. So, $B_{ij} = p_{Y_k|X_k}(j|i)$: the probability of observing output j from state i is B_{ij}. Since the number of possible observations isn't necessarily the same as the number of possible states, B won't necessarily be square.
  - An *initial state distribution* $p_{X_1}(x_1)$, which describes the starting distribution over states. We'll represent this with a vector called π_0, where item i in the vector represents $p_{X_1}(i)$.
- We compute forward and backwards messages as follows:

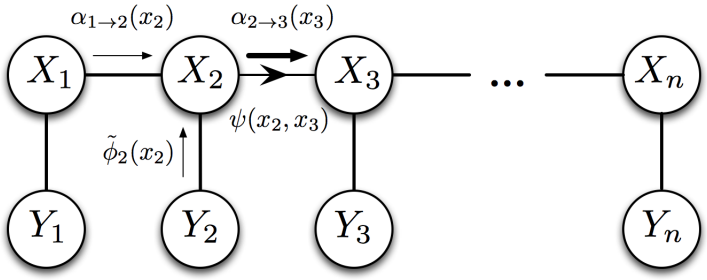
$$\begin{aligned}
\alpha_{(k-1)\to k}(x_k) &= \sum_{x_{k-1}} \overbrace{\alpha_{(k-2)\to(k-1)}(x_{k-1})}^{\text{previous message}} \overbrace{\tilde{\phi}(x_{k-1})}^{\text{obs.}} \overbrace{\psi(x_{k-1},x_k)}^{\text{transition}} \\
\beta_{(k+1)\to k}(x_k) &= \sum_{x_{k+1}} \underbrace{\beta_{(k+2)\to(k+1)}(x_{k+1})}_{\text{previous message}} \underbrace{\tilde{\phi}(x_{k+1})}_{\text{obs.}} \underbrace{\psi(x_k,x_{k+1})}_{\text{transition}}
\end{aligned}$$

  These messages are illustrated below:



  Here, $\tilde{\phi}(x_k) = p_{Y_k|X_k}(y_k|x_k)$ and $\psi(x_k,x_{k+1}) = p_{X_{k+1}|X_k}(x_k|x_{k+1})$. The first forward message $\alpha_{0\to1}(x_1)$ is initialized to $\pi_0(x_1) = p_{X_1}(x_1)$. The first backward message $\beta_{(n+1)\to n}(x_n)$ is initialized to uniform (this is equivalent to not including it at all).

  The picture below illustrates the computation of one forward message $\alpha_{2\to3}(x_3)$.



In order for node 2 to summarize its belief about X_3, it must incorporate the previous message $\alpha_{1\to2}(x_2)$, its observation $\tilde{\phi}(x_2)$, and the relationship $\psi(x_2,x_3)$ between X_2 and X_3.

- To obtain a marginal distribution for a particular state given all the observations, $p_{X_k|Y_1,\ldots,Y_n}$, we simply multiply the incoming messages together with the observation term, and then normalize:

$$\alpha_{(k-1)\to k}(x_k)\beta_{(k+1)\to k}(x_k)\tilde{\phi}(x_k) \quad \propto \quad \alpha_{(k-1)\to k}(x_k)\beta_{(k+1)\to k}(x_k)\tilde{\phi}(x_k)$$

Here, the symbol $\propto$ means "is proportional to", and indicates that we must normalize at the end so that the answer sums to 1.

Technical note: Traditionally, the backward messages β are computed as described here, but the forward messages α are computed as $\alpha'_{2\to3}(x_3) = \alpha_{2\to3}(x_3)\tilde{\phi}(x_3)$. In general, this means that the forward message to X_k would also include the observation Y_k. How would you modify the equations we have presented to use messages $\alpha'_{2\to3}(x_3) = \alpha_{2\to3}(x_3)\tilde{\phi}(x_3)$ and β instead and still produce the correct marginals?

2.5.3 Formulating HMM's

Suppose you send your robot, Inquisition, to Mars. Unfortunately, it gets stuck in a canyon while landing and most of its sensors break. You know the canyon has 3 areas. Areas 1 and 3 are sunny and hot, while Area 2 is cold. You decide to plan a rescue mission for the robot from Area 3, knowing the following things about the robot:

- Every hour, it tries to move forward by one area (i.e. from Area 1 to Area 2, or Area 2 to Area 3). It succeeds with probability 0.75 and fails with probability 0.25. If it fails, it stays where it is. If it is in Area 3, it always stays there (and waits to be rescued).
- The temperature sensor still works. Every hour, we get a binary reading telling us whether the robot's current environment is hot or cold.
- We have no idea where the robot initially got stuck.

Let's construct an HMM for this problem, which amounts to coming up with the transition matrix A, observation matrix B, and initial state distribution π_0.

We'll start with the transition matrix. Remember that each row corresponds to the current state, and each column corresponds to the next state. We'll use 3 states, each corresponding to one area.

- If the robot is in Area 1, it stays where it is with probability 0.25, moves to Area 2 with probability 0.75, and can't move to Area 3.
- Similarly, if the robot is in Area 2, it stays where it is with probability 0.25, can't move back to Area 1, and moves to Area 3 with probability 0.75.
- If the robot is in Area 3, it always stays in Area 3.

Each item above gives us one row of A. Putting it all together, we obtain

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0.25 & 0.75 & 0 \\ 0 & 0.25 & 0.75 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

Next, let's look at the observation matrix. There are two possible observations, hot and cold. Areas 1 and 3 always produce "hot" readings while Area 2 always produces a "cold" reading:

$$B = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cc} hot & cold \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \end{array}$$

Last but not least, since we have no idea where the robot starts, our initial state distribution will be uniform:

$$\pi_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \\ \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} \end{array}$$

2.5.4 Forward and Backward Messages for HMM's

Before doing any computation, we see that the sequence (hot,cold,hot) could only have been observed from the hidden state sequence (1,2,3). Make sure you convince yourself this is true before continuing!

We'll start with the forward messages.

$$\sum_{x_1} \underbrace{\alpha_{0\to 1}(x_1)\tilde{\phi}(x_1)}_{\text{depends only on } x_1} \psi(x_1, x_2)$$

The output message should have three different possibilities, one for each value of x_2. We can therefore represent it as a vector indexed by x_2. For each term in the sum (i.e. each possible value of x_1):

- $\alpha_{0\to 1}$ comes from from the initial distribution. Normally it would come from the previous message, but our first forward message is always set to initial state distribution.
- $\tilde{\phi}$ comes from the column of B corresponding to our observation $y_1 = \text{hot}$.
- ψ comes from a row of A: we are fixing x_1 and asking about possible values for x_1, which corresponds exactly to the transition distributions given in the rows of A (remember that the rows of A correspond to the current state and the columns correspond to the next state).

So, we obtain

$$\begin{aligned} \alpha_{1\to 2} &= \overbrace{\frac{1}{3}\cdot 1\cdot \begin{pmatrix} .25 \\ .75 \\ 0 \end{pmatrix}}^{x_1=1} + \overbrace{\frac{1}{3}\cdot 0\cdot \begin{pmatrix} 0 \\ .25 \\ .75 \end{pmatrix}}^{x_1=2} + \overbrace{\frac{1}{3}\cdot 1\cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}}^{x_1=3} \\ &\propto \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix} \end{aligned}$$

Since our probabilities are eventually computed by multiplying messages and normalizing, we can arbitrary renormalize at any step to make the computation easier.

For the second message, we perform a similar computation:

$$\begin{aligned}
\alpha_{2\to 3} &= \sum_{x_2} \alpha_{1\to 2}(x_2)\tilde{\phi}(x_2)\psi(x_2,x_3) \\
&= \overbrace{1\cdot 0\cdot \begin{pmatrix}.25\\.75\\0\end{pmatrix}}^{x_2=1} + \overbrace{3\cdot 1\cdot \begin{pmatrix}0\\.25\\.75\end{pmatrix}}^{x_2=2} + \overbrace{4\cdot 0\cdot \begin{pmatrix}0\\0\\1\end{pmatrix}}^{x_2=3} \\
&\propto \begin{pmatrix}0\\1\\3\end{pmatrix}
\end{aligned}$$

The backwards messages are computed using a similar formula:

$$\beta_{3\to 2} = \sum_{x_3} \underbrace{\beta_{4\to 3}(x_3)\tilde{\phi}(x_3)}_{\text{depends only on } x_3} \psi(x_2,x_3)$$

The first backwards message, $\beta_{4\to 3}(x_3)$, is always initialized to uniform since we have no information about what the last state should be. Note that this is equivalent to not including that term at all.

For each value of x_3, the transition term $\psi(x_2,x_3)$ is now drawn from a column of A, since we are interested in the probability of arriving at x_3 from each possible state for x_2. We compute the messages as:

$$\begin{aligned}
\beta_{3\to 2} &= \overbrace{1\cdot \begin{pmatrix}.25\\0\\0\end{pmatrix}}^{x_3=1} + \overbrace{0\cdot \begin{pmatrix}.75\\.25\\0\end{pmatrix}}^{x_3=2} + \overbrace{1\cdot \begin{pmatrix}0\\.75\\1\end{pmatrix}}^{x_3=3} \\
&\propto \begin{pmatrix}1\\3\\4\end{pmatrix}
\end{aligned}$$

Similarly, the second backwards message is:

$$\begin{aligned}
\beta_{2\to 1} &= \overbrace{1\cdot 0\cdot \begin{pmatrix}.25\\0\\0\end{pmatrix}}^{x_2=1} + \overbrace{3\cdot 1\cdot \begin{pmatrix}.75\\.25\\0\end{pmatrix}}^{x_2=2} + \overbrace{4\cdot 0\cdot \begin{pmatrix}0\\.75\\1\end{pmatrix}}^{x_2=3} \\
&\propto \begin{pmatrix}3\\1\\0\end{pmatrix}
\end{aligned}$$

Notice from the symmetry of the problem that our forwards messages and backwards messages were the same.

To compute the marginal distribution for X_2 given the data, we multiply the messages and the observation:

$$\begin{aligned} p_{X_2|Y_1,\ldots,Y_n}(x_2|y_1,\ldots,y_n) &\propto \alpha_{1\to 2}(x_2)\beta_{3\to 2}(x_2)\tilde{\phi}(x_2) \\ &\propto \begin{pmatrix}1\\3\\4\end{pmatrix}\cdot\begin{pmatrix}1\\3\\4\end{pmatrix}\cdot\begin{pmatrix}0\\1\\0\end{pmatrix} \\ &= \begin{pmatrix}0\\1\\0\end{pmatrix} \end{aligned}$$

Notice that in this case, because of our simplified observation model, the observation "cold" allowed us to determine the state. This matches up with our earlier conclusion that the robot must have been in Area 2 during the second hour.

2.6 Inference with Graphical Models - Most Probable Configuration

2.6.1 Most Probable Configurations in Graphical Models

Recall that for tree-structured undirected graphical models, the sum-product algorithm allowed us to compute marginal distributions $p_{X_i}(\cdot)$. Of course, we could fold in observations as well and compute $p_{X_i|Y_1,\ldots,Y_N}(\cdot|y_1,\ldots,y_n)$, with y_i's treated as constants. Applied to hidden Markov models (HMMs), this led to the forward-backward algorithm.

For tree-structured undirected graphical models, we now ask for the most probable configuration:

$$(\hat{x}_1,\ldots,\hat{x}_n) = \arg\max_{x_1,\ldots,x_n} p_{X_1,\ldots,X_n}(x_1,\ldots,x_n).$$

Being able to compute this efficiently would, as before, enable us to compute the most probable configuration given an observation (or even many observations): $\arg\max_{x_1,\ldots,x_n} p_{X_1,\ldots,X_n|Y}(x_1,\ldots,x_n|y)$. Recall from the first part of the course that the most probable configuration given observation(s) is precisely the MAP estimate!

Note that in these problems, we're interested in the arg max (i.e., the value that achieves the maximum) rather than the actual maximum value itself.

2.6.2 The Max-Product Algorithm

The Key Idea Behind Sum-Product and Max-Product

In deriving sum-product, the key idea was to push in summations. As a simple example, this amounts to the following:

$$\textstyle\sum_{x_2} f(x_1)g(x_1,x_2) = f(x_1)\sum_{x_2} g(x_1,x_2),$$

for arbitrary functions f and g.

A similar idea applies when we want the maximum:

$$\max_{x_1,x_2}\{f(x_1)g(x_1,x_2)\} = \max_{x_1}\left[\max_{x_2} f(x_1)g(x_1,x_2)\right] = \max_{x_1}\left[f(x_1)\max_{x_2} g(x_1,x_2)\right],$$

where the second equality holds provided that $f(x_1)$ is nonnegative for all x_1. (Why would a negative value kill the equality?)

For the equation above, think of the right-hand side function $f(x_1)\max_{x_2} g(x_1,x_2)$ as some 1-dimensional table with a number associated with each value of x_1. (In particular, suppose we've precomputed what $\max_{x_2} g(x_1,x_2)$ is for every value of x_1.) Then the argument $\hat{x}_1$ achieving the maximum is whichever x_1 has the highest value in the table. In other words, the highest value is

$$f(\hat{x}_1)\max_{x_2} g(\hat{x}_1, x_2),$$

from which we see that to get the optimal value of x_2, we just need to figure out which x_2 maximizes the function $g(\hat{x}_1, x_2)$.

So we first found the optimal value $\hat{x}_1$ for x_1. Then we traced backward and found the best value for $\hat{x}_2$ given $\hat{x}_1$, where we could readily answer this query if when we first compute $\max_{x_2} g(x_1, x_2)$, we also store what value of x_2 achieves the maximum for each value of x_1.

The intuition for this two-variable example carries over to the case when we have many variables represented by a tree-structured undirected graphical model and we want the most probable configuration. The resulting algorithm is the max-product algorithm.

Max-Product

As a reminder of notation, for random variables $X_1, \ldots, X_n$ corresponding to a tree $G = (V, E)$,

$$p_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = \frac{1}{Z}\left(\prod_{i\in V} \phi_i(x_i)\right)\left(\prod_{(i,j)\in E} \psi_{i,j}(x_i, x_j)\right).$$

The max-product algorithm is as follows. First, we pick a root node r and compute messages going from the leaves to the root:

$$m_{i\to j}(x_j) = \max_{x_i}\left[\phi_i(x_i)\psi_{i,j}(x_i, x_j)\prod_{k\in\mathcal{N}(i)\setminus\{j\}} m_{k\to i}(x_i)\right].$$

Note that this is formula is the same as that of sum-product except that the sum has been replaced by a max.

As with our simple two-variable example, we want to keep track of which argument achieves the maximum. Otherwise, all we'd be computing is a scaled version of the probability of the most probable configuration. (Remember that in general we don't know the normalization constant Z, in which case we won't actually find out the probability of the most probable configuration! Luckily, the argument achieving the maximum doesn't care what the value of Z is.)

To keep track of which argument achieves the maximum, we compute *traceback messages*:

$$t_{j\to i}(x_j) = \arg\max_{x_i}\left[\phi_i(x_i)\psi_{ij}(x_i, x_j)\prod_{k\in\mathcal{N}(i)\setminus\{j\}} m_{k\to i}(x_i)\right].$$

After computing the messages and traceback messages going from the leaves to the root, we're ready to find the most probable configuration, starting from the root and going to the leaves. This is like how we figured out the optimal value for x_1 in our two-variable example first (i.e., node 1 was the root).

For root node r, we compute the optimal value:

$$\hat{x}_r = \arg\max_{x_r}\left[\phi_r(x_r)\prod_{k\in\mathcal{N}(r)} m_{k\to r}(x_r)\right].$$

We then work backwards to the leaves, using the traceback messages to assign values:

$$\hat{x}_i = t_{j\to i}(\hat{x}_j).$$

2.6.3 Practice Problem: A Case When Traceback Tables Aren't Needed - Each Node Max-Marginal Has a Unique Most Probable Value

Recall that node max-marginal $\overline{p}_{X_i}$ satisfies

$$\overline{p}_{X_i}(x_i) \propto \underbrace{\max_{x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n}}_{\text{maximize over everything except } x_i} p_{X_1, X_2, \ldots, X_n}(x_1, x_2, \ldots, x_n).$$

Suppose that we knew that every node max-marginal $\overline{p}_{X_i}$ had a unique maximum probable value x_i^*, i.e.,

$$x_i^* = \arg\max_{x_i} \overline{p}_{X_i}(x_i), \qquad \text{and} \qquad \overline{p}_{X_i}(x_i) < \overline{p}_{X_i}(x_i^*) \quad \text{for all } x_i \neq x_i^*.$$

Show: The joint probability table $p_{X_1,X_2,\ldots,X_n}$ has a unique most probable configuration given by $x \triangleq (x_1^*, x_2^*, \ldots, x_n^*)$ (i.e., we put together the unique maximum probable values for each of the node max-marginals).

Hint: Start by supposing that there is a configuration $\widehat{x} \triangleq (\widehat{x}_1, \widehat{x}_2, \ldots, \widehat{x}_n)$ that is a most probable configuration, i.e.,

$$p_{X_1,\ldots,X_n}(\widehat{x}_1, \ldots, \widehat{x}_n) \geqslant p_{X_1,\ldots,X_n}(x_1, \ldots, x_n) \qquad \text{for all } x_1, \ldots, x_n,$$

which also means that

$$p_{X_1,\ldots,X_n}(\widehat{x}_1, \ldots, \widehat{x}_n) \geqslant p_{X_1,\ldots,X_n}(x_1^*, \ldots, x_n^*).$$

You'll want to use the definition of a node max-marginal.

Practical implication: If you run the max-marginal variant of the max-product algorithm that more closely resembles the sum product algorithm, and you check the node max-marginals to find that the most probable value for each of these max-marginals is unique, then you don't need traceback messages and you also don't need to follow backpointers.

Solution: Following the hint, let $\widehat{x} \triangleq (\widehat{x}_1, \widehat{x}_2, \ldots, \widehat{x}_n)$ be a most probable configuration, which means that

$$p_{X_1,\ldots,X_n}(\widehat{x}_1, \ldots, \widehat{x}_n) \geqslant p_{X_1,\ldots,X_n}(x_1, \ldots, x_n) \qquad \text{for all } x_1, \ldots, x_n.$$

In particular,

$$p_{X_1,\ldots,X_n}(\widehat{x}_1, \ldots, \widehat{x}_n) \geqslant p_{X_1,\ldots,X_n}(x_1^*, \ldots, x_n^*).$$

Then

$$\begin{aligned}
& \overline{p}_{X_i}(\widehat{x}_i) \\
(\text{since } \widehat{x} \text{ is a most probable configuration}) \quad & = p_{X_1,\ldots,X_n}(\widehat{x}_1, \ldots, \widehat{x}_n) \\
(\text{since } \widehat{x} \text{ is a most probable configuration}) \quad & = \max_{x_1,\ldots,x_n} p_{X_1,\ldots,X_n}(x_1, \ldots, x_n) \\
& = \max_{x_i} \max_{x_1,\ldots,x_{i-1},x_{i+1},x_n} p_{X_1,\ldots,X_n}(x_1, \ldots, x_n) \\
& = \max_{x_i} \overline{p}_{X_i}(x_i).
\end{aligned}$$

Look at the very left-most side of this equation and the very right-most side: what the equation says is that $\widehat{x}_i$ achieves the highest possible value in the node max-marginal p_{X_i}. And by assumption there is only one possible value for X_i that achieves this highest node max-marginal value: x_i^*. So we conclude that indeed $\widehat{x}_i = x_i^*$.

2.6.4 Numerical Stability Issues: Max-Product to Min-Sum

This note presents an important implementation detail that is often helpful. As you noticed in mini-project 1 on movie recommendations, working in the log domain helps with numerical stability.

Messages are computed by multiplying many potential function values and then propagating these products onwards to compute more messages. Meanwhile, potential functions can often consist of small numbers. Multiplying many small numbers may lead to underflow issues on computers, where multiplication just yields 0 when the product isn't actually 0.

A popular fix to this issue is to work with negative logs. Log turns small values into large values since values in $(0, 1]$ get mapped to $(-\infty, 0]$, and negating then turns these numbers positive. Large potential values aren't affected as much.

So taking the negative log of our messages, we get the following equations:

$$
\begin{aligned}
m'_{i\to j}(x_j) &= \min_{x_i}\left[-\log\phi_i(x_i) - \log\psi_{ij}(x_i,x_j) + \sum_{k\in\mathscr{N}(i)\setminus\{j\}} m'_{k\to i}(x_i)\right],\\
t_{j\to i}(x_j) &= \arg\min_{x_i}\left[-\log\phi_i(x_i) - \log\psi_{ij}(x_i,x_j) + \sum_{k\in\mathscr{N}(i)\setminus\{j\}} m'_{k\to i}(x_i)\right],\\
\hat{x}_r &= \arg\min_{x_r}\left[-\log\phi_r(x_r) + \prod_{k\in\mathscr{N}(r)} m'_{k\to r}(x_r)\right],\\
\hat{x}_i &= t_{j\to i}(\hat{x}_j).
\end{aligned}
$$

Following the usual lack of imagination in naming these algorithms, this algorithm is called the *min-sum algorithm* since it takes minimums of sums.

2.7 The Viterbi Algorithm

Max-product or min-sum specialized to HMMs results in the Viterbi algorithm, proposed by MIT alum Andrew Viterbi back in 1967, which pre-dates both the forward-backward algorithm (1980) and the sum-product algorithm (1982). The Viterbi algorithm produces an MAP estimate for the sequence of hidden states given the observations of an HMM.

Let's work through an example. I have two coins, one fair and one biased, and I keep picking a coin and flipping it. You get to observe whether each flip was heads or tails, and from that information, you have to guess the sequence of coins that I used.

At first, I choose either the fair coin or the biased coin uniformly at random. I prefer not to switch between flips: that is, after each flip, my next flip will use the same coin with probability 3/4, and switch coins with probability 1/4. The fair coin is equally likely to be heads or tails, and the biased coin has probability 3/4 of coming up tails and probability 1/4 of coming up heads.

If you observe the sequence $HHTTT$ (where H is heads and T is tails), what's the most likely sequence of coins that I used?

We reuse notation from how we formulated HMM's when we presented the forward-backward algorithm. In particular, for this HMM with 5 hidden states, we define the following potentials:

$$
\begin{aligned}
\widetilde{\phi}_i(x_i) &= \begin{cases} p_{X_1}(x_1)p_{Y_1|X_1}(y_1|x_1) & \text{for } i=1,\\ p_{Y_i|X_i}(y_i|x_i) & \text{for } i=2,\ldots,5,\end{cases}\\
\psi(x_{i-1},x_i) &= p_{X_i|X_{i-1}}(x_i \mid x_{i-1}) \quad \text{for } n=2,\ldots,5.
\end{aligned}
$$

Thus, we have (importantly, observations $HHTTT$ have been folded into the node potentials!):

ψ

		x_i: fair	x_i: biased
x_{i-1}	fair	3/4	1/4
	biased	1/4	3/4

$\widetilde{\phi}_1$

x_1	fair	$(1/2)\times(1/2)$
	biased	$(1/2)\times(1/4)$

$\widetilde{\phi}_2$

x_2	fair	1/2
	biased	1/4

$\widetilde{\phi}_i,\ i=3,4,5$

x_i	fair	1/2
	biased	3/4

Defining $\gamma \triangleq \log_2(3) \approx 1.6$, we have:

$-\log_2 \psi$

		x_i: fair	x_i: biased
x_{i-1}	fair	$2-\gamma$	2
	biased	2	$2-\gamma$

$-\log_2 \widetilde{\phi}_1$

x_1	fair	2
	biased	3

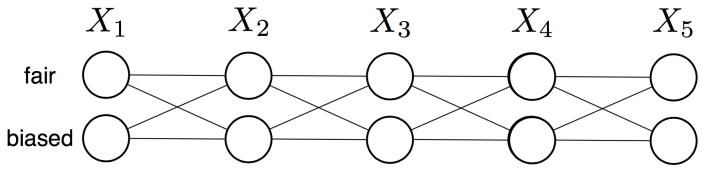
$-\log_2 \widetilde{\phi}_2$

x_2	fair	1
	biased	2

$-\log_2 \widetilde{\phi}_i,\ i = 3,4,5$

x_i	fair	1
	biased	$2-\gamma$

Below, we show what is called the *trellis diagram* for this problem: note that while this diagram has nodes and edges, it is not a probabilistic graphical model in that the nodes do not correspond to random variables, and the nodes/edges aren't associated with potential tables, etc. Instead, the i-th column consists of all the possible states that X_i can be, and the edges denote all possible transitions, so that a path going from left to right (a path in a trellis diagram can't go rightward and then leftward; it can only go rightward) corresponds to a specific possible sequence of states that $X_1, \ldots, X_n$ can take on.



The trellis diagram for our example

We will see that the Viterbi algorithm just finds a path along this trellis.

For a length-n HMM, the min-sum messages are

$$
\begin{aligned}
m'_{1\to 2}(x_2) &= \min_{x_1}\Big[-\log_2 \widetilde{\phi}_1(x_1) - \log_2 \psi(x_1,x_2)\Big], \\
t_{2\to 1}(x_2) &= \arg\min_{x_1}\Big[-\log_2 \widetilde{\phi}_1(x_1) - \log_2 \psi(x_1,x_2)\Big], \\
m'_{(i-1)\to i}(x_i) &= \min_{x_{i-1}}\Big[-\log_2 \widetilde{\phi}_{i-1}(x_{i-1}) - \log_2 \psi(x_{i-1},x_i) + m'_{(i-2)\to(i-1)}(x_{i-1})\Big] \qquad \text{for } i = 3,\ldots,n, \\
t_{i\to(i-1)}(x_i) &= \arg\min_{x_{i-1}}\Big[-\log_2 \widetilde{\phi}_{i-1}(x_{i-1}) - \log_2 \psi(x_{i-1},x_i) + m'_{(i-2)\to(i-1)}(x_{i-1})\Big] \qquad \text{for } i = 3,\ldots,n.
\end{aligned}
$$

Let's determine the messages for our length $n = 5$ HMM.

We compute the first message:

$$
\begin{aligned}
m_{1\to 2}(\text{fair}) &= \min_{x_1}\big[-\log_2 \widetilde{\phi}_1(x_1) - \log_2(\psi(x_1,\text{fair}))\big] \\
&= \min\Big\{\overbrace{-\log_2 \widetilde{\phi}_1(\text{fair}) - \log_2(\psi(\text{fair},\text{fair}))}^{x_1=\text{fair}}, \\
&\qquad \underbrace{-\log_2 \widetilde{\phi}_1(\text{biased}) - \log_2(\psi(\text{biased},\text{fair}))}_{x_1=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{2+(2-\gamma)}_{x_1=\text{fair}}, \underbrace{3+2}_{x_1=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{4-\gamma}_{x_1=\text{fair}}, \underbrace{5}_{x_1=\text{biased}}\Big\} \\
&= 4-\gamma,
\end{aligned}
$$

where the arg min is $x_1 = fair$, so $t_{2\to1}(\text{fair}) = \text{fair}$.

Next:

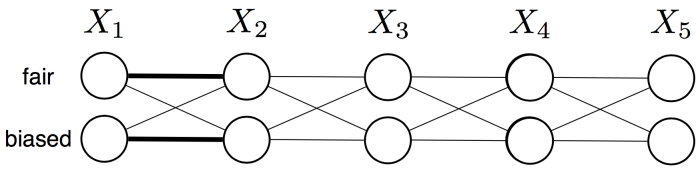
$$\begin{aligned}
m_{1\to2}(\text{biased}) &= \min_{x_1}\big[-\log_2\widetilde{\phi}_1(x_1) - \log_2(\psi(x_1,\text{biased}))\big] \\
&= \min\Big\{\overbrace{-\log_2\widetilde{\phi}_1(\text{fair}) - \log_2(\psi(\text{fair},\text{biased}))}^{x_1=\text{fair}}, \\
&\qquad \underbrace{-\log_2\widetilde{\phi}_1(\text{biased}) - \log_2(\psi(\text{biased},\text{biased}))}_{x_1=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{2+2}_{x_1=\text{fair}}, \underbrace{3+(2-\gamma)}_{x_1=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{4}_{x_1=\text{fair}}, \underbrace{5-\gamma}_{x_1=\text{biased}}\Big\} \\
&= 5-\gamma,
\end{aligned}$$

where the arg min is $x_1 = biased$, so $t_{2\to1}(\text{biased}) = \text{biased}$.

Our message and traceback tables are then:

$$\begin{aligned}
m'_{1\to2}(x_2) &= \begin{cases} 4-\gamma & \text{if } x_2 = \text{fair} \\ 5-\gamma & \text{if } x_2 = \text{biased} \end{cases} \\
t_{2\to1}(x_2) &= \begin{cases} \text{fair} & \text{if } x_2 = \text{fair} \\ \text{biased} & \text{if } x_2 = \text{biased} \end{cases}
\end{aligned}$$

This traceback table tells us how to select x1 once we decide on x2. This is illustrated in the first-step trellis diagram below:



The first traceback

The second message is computed similarly:

$$\begin{aligned}
m'_{2\to3}(\text{fair}) &= \min_{x_2}\big[-\log_2\widetilde{\phi}_2(x_2) - \log_2(\psi(x_2,\text{fair})) + m'_{1\to2}(x_2)\big] \\
&= \min\Big\{\overbrace{-\log_2\widetilde{\phi}_2(\text{fair}) - \log_2(\psi(\text{fair},\text{fair})) + m'_{1\to2}(\text{fair})}^{x_2=\text{fair}}, \\
&\qquad \underbrace{-\log_2\widetilde{\phi}_2(\text{biased}) - \log_2(\psi(\text{biased},\text{fair})) + m'_{1\to2}(\text{biased})}_{x_2=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{1+(2-\gamma)+(4-\gamma)}_{x_2=\text{fair}}, \underbrace{2+2+(5-\gamma)}_{x_2=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{7-2\gamma}_{x_2=\text{fair}}, \underbrace{9-\gamma}_{x_2=\text{biased}}\Big\} \\
&= 7-2\gamma,
\end{aligned}$$

where the arg min is $x_2 = fair$, so $t_{3\to 2}(\text{fair}) = \text{fair}$.

Next:

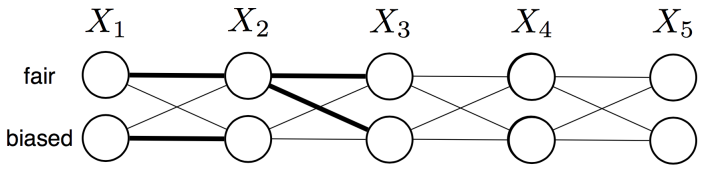
$$\begin{aligned}
m'_{2\to 3}(\text{biased}) &= \min_{x_2}\big[-\log_2\widetilde{\phi}_2(x_2) - \log_2(\psi(x_2,\text{biased})) + m'_{1\to 2}(x_2)\big] \\
&= \min\Big\{\overbrace{-\log_2\widetilde{\phi}_2(\text{fair}) - \log_2(\psi(\text{fair},\text{biased})) + m'_{1\to 2}(\text{fair})}^{x_2=\text{fair}}, \\
&\qquad \underbrace{-\log_2\widetilde{\phi}_2(\text{biased}) - \log_2(\psi(\text{biased},\text{biased})) + m'_{1\to 2}(\text{biased})}_{x_2=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{1+2+(4-\gamma)}_{x_2=\text{fair}}, \underbrace{2+(2-\gamma)+(5-\gamma)}_{x_2=\text{biased}}\Big\} \\
&= \min\Big\{\underbrace{7-\gamma}_{x_2=\text{fair}}, \underbrace{9-2\gamma}_{x_2=\text{biased}}\Big\} \\
&= 7-\gamma,
\end{aligned}$$

where the arg min is $x_2 = fair$, so $t_{3\to 2}(\text{biased}) = \text{fair}$.

Our message and traceback tables are then:

$$\begin{aligned}
m'_{2\to 3}(x_3) &= \begin{cases} 7-2\gamma & \text{if } x_3 = \text{fair} \\ 7-\gamma & \text{if } x_3 = \text{biased} \end{cases} \\
t_{3\to 2}(x_3) &= \begin{cases} \text{fair} & \text{if } x_3 = \text{fair} \\ \text{fair} & \text{if } x_3 = \text{biased} \end{cases}
\end{aligned}$$

The traceback is illustrated below:



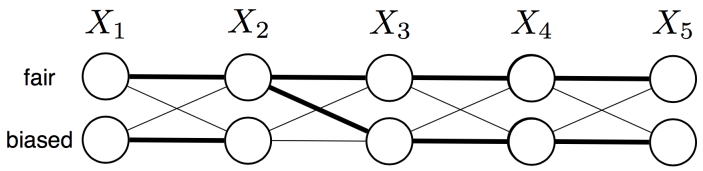
The second traceback

We see that after observing wo heads in a row, we will eventually pick $x_2 = \text{fair}$ no matter what x_3 is.

Make sure that you can also compute these messages! The rest of the message and traceback tables are as follows:

$$\begin{aligned}
m'_{3\to 4}(x_4) &= \begin{cases} 10-3\gamma & \text{if } x_4 = \text{fair} \\ 11-3\gamma & \text{if } x_4 = \text{biased} \end{cases} \\
t_{4\to 3}(x_4) &= \begin{cases} \text{fair} & \text{if } x_4 = \text{fair} \\ \text{biased} & \text{if } x_4 = \text{biased} \end{cases} \\
m'_{4\to 5}(x_5) &= \begin{cases} 13-4\gamma & \text{if } x_5 = \text{fair} \\ 15-5\gamma & \text{if } x_5 = \text{biased} \end{cases} \\
t_{5\to 4}(x_5) &= \begin{cases} \text{fair} & \text{if } x_5 = \text{fair} \\ \text{biased} & \text{if } x_5 = \text{biased} \end{cases}
\end{aligned}$$

The traceback is illustrated below:

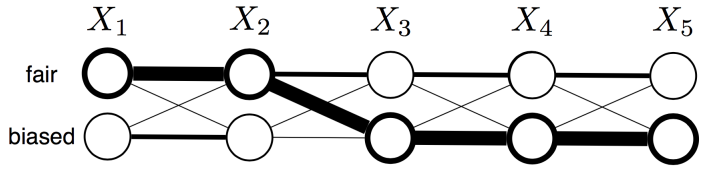


The final traceback

Finally, we can compute the most likely value for X_5:

$$\hat{x}_5 = \arg\min\Big(\underbrace{1+(13-4\gamma)}_{x_5=\text{fair}}, \underbrace{(2-\gamma)+(15-5\gamma)}_{x_5=\text{biased}}\Big) = \text{biased}$$

We can see from the trellis diagram that as soon as we decide on a value for X_5, we just have to follow our "trail of breadcrumbs" from the traceback messages to get the path highlighted below:



The result of the Viterbi algorithm

We can do the same thing with the traceback messages:

$$\begin{aligned}
\hat{x}_4 &= t_{5\to4}(\hat{x}_5) = \text{biased} \\
\hat{x}_3 &= t_{4\to3}(\hat{x}_4) = \text{biased} \\
\hat{x}_2 &= t_{3\to2}(\hat{x}_3) = \text{fair} \\
\hat{x}_1 &= t_{2\to1}(\hat{x}_2) = \text{fair}
\end{aligned}$$

We conclude that the MAP estimate for the sequence of hidden states given observed sequence $HHTTT$ is "fair, fair, biased, biased, biased".

Chapter 3

Learning a Probabilistic Model from Data

3.1 Introduction to Learning Probabilistic Models

As we have seen, graphical models are a powerful mechanism for capturing structure in distributions that enables efficient inference. We're now ready to talk about how to choose such models. This is the problem of model selection. Invariably, domain knowledge guides aspects of the model we choose – for example, the underlying physics, biology, economics, et cetera. However, in practice, some aspects of the model we must often learn directly from data.

So now let's turn our attention to how to learn models. We'll begin with a discussion of a rather general framework of learning based on the time-honored principle of maximum likelihood. We will then put it to work to help us learn a wide range of tree-structured graphical models, including hidden Markov models and what are referred to as naive Bayes models – which, despite their name, are surprisingly powerful.

In particular, we'll show how to efficiently learn the conditional distribution tables that characterize a tree model. This is called parameter estimation. And we will see that the concept of empirical distributions or histograms plays an important role.

Finally, we will show how to solve the problem of finding a tree that best fits the data, which is referred to as structure learning. Amazingly, even though there are a super exponential number of possible trees, we will see how to solve the problem with only quadratic search complexity using the celebrated Chow-Liu algorithm, which is based on measuring empirical mutual information. Exciting applications for the learning methodology we develop are everywhere, from natural language processing and automatic face recognition to phylogenetics and well beyond. So let's begin.

3.1.1 Learning Probabilistic Models: Notation and Outline

Thus far in the course, we've been given a probabilistic model of the uncertain world, from which we produced predictions given observations. But where do these probabilistic models come from? We now turn to the problem of learning such models (also referred to as model selection since we are selecting which model to use).

As a concrete example that we'll build on later in this section: We don't actually know the underlying probability distribution for what makes an email considered spam vs. "ham" (i.e., not spam). However, we do have access to plenty of training examples of what emails are spam or ham, where a user has manually flagged her or his incoming emails as spam or not. From all these training examples, we could learn a model for what spam emails look like and what ham emails look like.

There are two levels of learning we consider:

- Parameter learning: Suppose we know what the edges are in an undirected graphical model but we don't know

what the table entries should be for the potentials – how do we estimate these entries?

- Structure learning: What if we know neither the parameters nor which edges are present in an undirected graphical model? In this case, we could first figure out what edges are present. After we decide on which edges are present, then the problem reduces to the first problem of parameter learning.

In both cases, the high-level setup is the same: there is some underlying probability distribution p that we don't know details for but want to learn. The distribution p has some parameter (or a set of parameters) θ. We will assume that we can collect n independent samples $X^{(1)}, \ldots, X^{(n)}$ from the distribution p (these n samples are often referred to as "training data"). Given these samples, we aim to estimate θ using what's called "maximum likelihood", which tries to learn a model that in some sense best fits the training data we have available. Along the way, we will see how information theory plays a crucial role in helping us not only develop computer programs to learn probability distributions but it also provides an interpretation of what the programs are doing.

We begin with parameter learning, starting with a single node graphical model and working our way to richer graphical models whose potential table entries we aim to learn:

1. A single node undirected graphical model corresponding to a single finite random variable
2. A graphical model called the naive Bayes model which can be used for tasks like email spam detection and handwritten digit recognition
3. General tree-structured undirected graphical models

We then turn toward structure learning, proceeding directly to the most general class of graphical models we consider: tree-structured undirected graphical models, where we learn which edges to include in the tree and what potential tables to assign to the edges.

3.2 Introduction to Parameter Learning - Maximum Likelihood and MAP Estimation

3.2.1 Introduction to Maximum Likelihood

We consider a single binary random variable X, that for simplicity can be thought of as a (possibly biased) coin flip, taking on values in the set $\mathscr{X} = \{\text{heads}, \text{tails}\}$. While this setup is extremely simple, understanding how parameter learning works here will already give us most of the intuition for our parameter learning coverage! We'll generalize to the finite (can be non-binary) random variable case later.

The issue is that the probability of heads, which denote θ, is unknown, and we'd like to estimate (or "learn") this probability.

The probability table is:

		Prob.
X	heads	θ
	tails	$1-\theta$

Notation: Recall that previously we would write the probability table of random variable X as p_X or $p_X(\cdot)$. However, now to make it explicit that we don't know θ, which we aim to estimate, we will denote the probability table as $p_X(\cdot;\theta)$. The semi-colon is used to say that everything after the semi-colon in the parentheses refers to parameter(s) of the model. The probability that $X = x$ is denoted as $p_X(\cdot;\theta)$.

To estimate parameter θ, we assume we have flipped the coin n times to get outcomes $X^{(1)}, X^{(2)}, \ldots, X^{(n)}$ which are i.i.d. samples from the same distribution as X. In other words,

$$p_{X^{(1)},X^{(2)},\ldots,X^{(n)}}(x^{(1)}, x^{(2)}, \ldots, x^{(n)}; \theta) = \prod_{i=1}^{n} p_X(x^{(i)}; \theta).$$

The above probability is called the "likelihood" of the data – it is the probability of seeing the observed data as a function of the unknown parameter *theta*. Note that the observed data are treated as fixed constants! We denote the likelihood as $L(\theta)$.

As an example, if we observed the sequence heads, tails, heads, then $n = 3$, $X^{(1)} =$ heads, $X^{(2)} =$ tails, and $X^{(3)} =$ heads, and the likelihood is

$$L(\theta) = \underbrace{\theta}_{\text{heads}} \cdot \underbrace{(1-\theta)}_{\text{tails}} \cdot \underbrace{\theta}_{\text{heads}} = \theta^2(1-\theta).$$

Next, to estimate (or learn) θ, we will use "maximum likelihood" which maximizes the likelihood function L over possible values of the parameter θ. Put another way, we find whichever probability of heads θ makes the probability of seeing the samples $X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \ldots, X^{(n)} = x^{(n)}$ as high as possible.

Formally, the maximum likelihood estimate $\widehat{\theta}$ for parameter θ is the solution to the following optimization problem:

$$\widehat{\theta} = \arg\max_{\theta \in [0,1]} \underbrace{\prod_{i=1}^{n} p_X(x^{(i)}; \theta)}_{\text{likelihood}}.$$

It turns out the answer in this case is quite simple.

Claim: The maximum likelihood estimate $\widehat{\theta}$ for the true probability of heads θ is simply the fraction of times we see heads in the observed sequence $x^{(1)}, \ldots, x^{(n)}$, i.e.,

$$\widehat{\theta} = \frac{\text{number of times heads appears}}{n} = \frac{\sum_{i=1}^{n} \mathbf{1}\{x^{(i)} = \text{heads}\}}{n}.$$

For example, if we observed the sequence heads, tails, heads, then the maximum likelihood estimate for the probability of heads is 2/3 since 2/3 of the tosses were heads.

Let's prove the claim.

First, by how the problem is set up:

$$p_X(x^{(i)}; \theta) = \begin{cases} \theta & \text{if } x^{(i)} = \text{heads}, \\ 1-\theta & \text{if } x^{(i)} = \text{tails}. \end{cases}$$

Next, letting $n_{\text{heads}} \triangleq \sum_{i=1}^{n} \mathbf{1}\{x^{(i)} = \text{heads}\}$ be the number of times heads occurred, and $n_{\text{tails}} \triangleq \sum_{i=1}^{n} \mathbf{1}\{x^{(i)} = \text{tails}\}$ be the number of times tails occurred, we have, due to independence of the coin flips and because the coin flips all have the same distribution:

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{n} p_X(x^{(i)}; \theta) \\ &= \Big(\underbrace{\theta \times \cdots \times \theta}_{n_{\text{heads}} \text{ times}}\Big) \times \Big(\underbrace{(1-\theta) \times \cdots \times (1-\theta)}_{n_{\text{tails}} \text{ times}}\Big) \\ &= \theta^{n_{\text{heads}}}(1-\theta)^{n_{\text{tails}}}. \end{aligned}$$

Our next goal is to maximize the likelihood. Mathematically it'll be easier to work with the log of the likelihood. Note that the value of θ that maximizes the likelihood is the same as the value of θ that maximizes the log of the likelihood, because the log function is strictly increasing, so:

$$\begin{aligned}
\widehat{\theta} &= \arg\max_{\theta\in[0,1]} L(\theta) \\
&= \arg\max_{\theta\in[0,1]} \log(L(\theta)) \\
&= \arg\max_{\theta\in[0,1]} \log\left(\theta^{n_{\text{heads}}}(1-\theta)^{n_{\text{tails}}}\right) \\
&= \arg\max_{\theta\in[0,1]} \Big\{ \underbrace{n_{\text{heads}}\log\theta + n_{\text{tails}}\log(1-\theta)}_{\triangleq \ell(\theta)} \Big\}.
\end{aligned}$$

Let's examine the log likelihood function ℓ that we've just defined and that we're aiming to maximize:

$$\ell(\theta) = n_{\text{heads}}\log\theta + n_{\text{tails}}\log(1-\theta).$$

From calculus you may remember that we can optimize this function by looking at what value for θ makes $\frac{d\ell(\theta)}{d\theta} = 0$. It turns out that this will be sufficient and solving this optimization (which we'll spell out shortly) will yield that the optimal $\widehat{\theta}$ is the fraction of heads observed in the n flips.

However, for just this binary random variable case, we'll be a little bit more rigorous and give all the details, which should help you connect what you've seen in calculus to what is happening here in maximum likelihood estimation.

Without further ado, when we maximize $\ell(\theta)$, we break the problem up into three cases:

- If $n_{\text{tails}} = 0$, then
$$\ell(\theta) = n_{\text{heads}}\log\theta$$
is maximized when $\theta = 1$ since $\log\theta$ increases as θ ranges from 0 to 1.
- If $n_{\text{heads}} = 0$, then
$$\ell(\theta) = n_{\text{tails}}\log(1-\theta)$$
is maximized when $\theta = 0$ since $\log(1-\theta)$ decreases as we increase θ from 0 to 1.
- If neither n_{heads} nor n_{tails} is 0, which means that both are positive (they can't be negative, and they can't both be 0 since that would mean we didn't have any observations), then note that ℓ is a differentiable real-valued function defined on the domain $0 < \theta < 1$. To see why this is the case:
  - Note that we can't have $\theta < 0$ or $\theta > 1$ because in both of these cases, we'd have a term in $\ell(\theta)$ that is the log of a negative number, which isn't defined for real-valued functions.
  - When $\theta = 0$ or $\theta = 1$, we encounter a term $\log 0 = -\infty$, which is not a real number, and so $\ell(\theta)$ in this case is not a real number.
  - When $\theta \in (0,1)$, then $\log\theta$ and $\log(1-\theta)$ are both real numbers, and so is $\ell(\theta)$.

Thus, ℓ is only a real-valued function on the interval $\theta \in (0,1)$. In fact, within this interval, we can compute its first and second derivatives (which we'll do shortly). At this point we recall from calculus that to optimize ℓ, it's sufficient to do two steps:

1. Find when $\frac{d\ell(\theta)}{d\theta} = 0$. Supposing that this happens for only one value of θ, we'll call this best value $\widehat{\theta}$.
2. Check that the second derivative $\frac{d^2\ell}{d\theta^2}$ evaluated at $\widehat{\theta}$ is negative, which implies that the point we found in the first step corresponds to the unique maximum.

Let's do step 1:

$$\begin{aligned}
0 &= \left[\frac{d\ell}{d\theta}\right]_{\theta=\hat{\theta}} \\
&= \left[\frac{d}{d\theta}\left\{n_{\text{heads}}\log\theta + n_{\text{tails}}\log(1-\theta)\right\}\right]_{\theta=\hat{\theta}} \\
&= \left[n_{\text{heads}}\frac{1}{\theta} - n_{\text{tails}}\frac{1}{1-\theta}\right]_{\theta=\hat{\theta}} \\
&= n_{\text{heads}}\frac{1}{\widehat{\theta}} - n_{\text{tails}}\frac{1}{1-\widehat{\theta}}.
\end{aligned}$$

Multiplying through by $\widehat{\theta}(1-\widehat{\theta})$, we get:

$$\begin{aligned}
0 &= n_{\text{heads}}(1-\widehat{\theta}) - n_{\text{tails}}\widehat{\theta} \\
&= n_{\text{heads}} - n_{\text{heads}}\widehat{\theta} - n_{\text{tails}}\widehat{\theta} \\
&= n_{\text{heads}} - (n_{\text{heads}} + n_{\text{tails}})\widehat{\theta}.
\end{aligned}$$

In other words,

$$\widehat{\theta} = \tfrac{n_{\text{heads}}}{n_{\text{heads}}+n_{\text{tails}}} = \tfrac{n_{\text{heads}}}{n}.$$

Step 2: The second derivative of ℓ with respective to θ is

$$\tfrac{d^2\ell}{d\theta^2} = -n_{\text{heads}}\tfrac{1}{\theta^2} - n_{\text{tails}}\tfrac{1}{(1-\theta)^2},$$

which is always negative since θ^2 and $(1-\theta)^2$ are always positive when $\theta \in (0,1)$.

Lastly, we also check the boundary, i.e., we make sure that indeed the log likelihood ℓ evaluated at $\widehat{\theta}$ is larger than each of $\ell(0)$ and $\ell(1)$. We have

$$\begin{aligned}
\ell(0) &= \underbrace{n_{\text{heads}}\log 0}_{-\infty} + \underbrace{n_{\text{tails}}\log 1}_{0} = -\infty, \\
\ell(1) &= \underbrace{n_{\text{heads}}\log 1}_{0} + \underbrace{n_{\text{tails}}\log 0}_{-\infty} = -\infty, \\
\ell(\widehat{\theta}) &= n_{\text{heads}}\underbrace{\log\frac{n_{\text{heads}}}{n}}_{>-\infty} + n_{\text{tails}}\underbrace{\log\frac{n_{\text{tails}}}{n}}_{>-\infty} > -\infty,
\end{aligned}$$

so indeed $\ell(\widehat{\theta}) > \ell(0)$ and $\ell(\widehat{\theta}) > \ell(1)$.

To summarize:

- If $n_{\text{tails}} = 0$, then we set $\widehat{\theta} = 1$
- If $n_{\text{heads}} = 0$, then we set $\widehat{\theta} = 0$
- If $n_{\text{heads}} > 0$ and $n_{\text{tails}} = 0$, then we set

  $$\widehat{\theta} = \tfrac{n_{\text{heads}}}{n},$$

  from which we see that this same formula holds true even for the cases of $n_{\text{tails}} = 0$ or $n_{\text{heads}} = 0$. The only reason why we were careful to look at the first two cases separately is because the log likelihood function ℓ is neither real-valued nor differentiable at $\theta = 0$ or $\theta = 1$, corresponding to the $n_{\text{heads}} = 0$ and $n_{\text{tails}} = 0$ cases.

Putting together the pieces, the maximum likelihood estimate for the probability of heads is equal to the fraction of heads we see amongst our n coin flips:

$$\widehat{\theta} = \frac{n_{\text{heads}}}{n}.$$

Remark: If we didn't take the log, we could still proceed with the same calculus tools but immediately the derivatives get messy due to the product rule of derivatives! Note though that it's not always the case that taking the log makes sense, as you'll see in one of the problems.

3.2.2 Practice Problem: The German Tank Problem

Suppose the Germans have k tanks, numbered $1, 2, \ldots, k$. The Allies observe $n < k$ tanks with numbers $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$. Assume that these numbers are drawn uniformly without replacement from $\{1, 2, \ldots, k\}$. The objective is to get an estimate $\widehat{k}$ of the total number of German tanks.

- **(a)** What is the ML estimate for k?

  Hints: Keep in mind that each $x^{(i)}$ is a value in $\{1, \ldots, k\}$. Also, this is one of those maximum likelihood problems where taking the log isn't helpful.

- **(b)** What is the bias of the ML estimate $\widehat{k}_{\text{ML}}(X^{(1)}, \ldots, X^{(n)})$ for k?

- **(b)** Determine an unbiased estimator for k based on the maximum likelihood estimate $\widehat{k}_{\text{ML}}$ for k.

Hint: Start by using your answer from (b) and setting the bias equal to 0 and solving for k.

Solutions **(a)** There are $\binom{k}{n}$ possible choices of which tanks we observe. Each of these is equally likely, so the likelihood is

$$p_X(x;\theta) = \frac{\mathbf{1}\{1 \leqslant x^{(1)} \leqslant k, 1 \leqslant x^{(2)} \leqslant k, \ldots, 1 \leqslant x^{(n)} \leqslant k\}}{\binom{k}{n}}.$$

The numerator says that each $x^{(i)}$ (i.e., the number the tank is labeled with) must be between 1 and k, the maximum number.

Now notice that with n treated as fixed, then to maximize the likelihood, we want the denominator to be as small as possible, which means that we want k to be as small as possible. However, the smallest k can be so that the numerator is nonzero is when $k = \max(x^{(1)}, \ldots, x^{(n)})$. Thus, the ML estimator for k is given by

$$\widehat{k}_{\text{ML}} = \max(x^{(1)}, \ldots, x^{(n)}).$$

(b) We want to evaluate the expectation of

$$\widehat{k}_{\text{ML}}(X^{(1)}, \ldots, X^{(n)}) = \max(X^{(1)}, X^{(2)}, \ldots, X^{(n)}).$$

The probability that the maximum tank number/label observed is m is $\frac{\binom{k-1}{n-1}}{\binom{k}{n}}$ since there are $\binom{k}{n}$ possible ways to observe n tanks, and $\binom{m-1}{n-1}$ ways in which we definitely picked the tank with number m and among the tanks with smaller numbers (i.e., among $m-1$ tanks) we choose $n-1$ of them. Then

$$\mathbb{E}[\widehat{k}_{\text{ML}}(X^{(1)}, \ldots, X^{(n)})] = \sum_{m=n}^{k} m\mathbb{P}(\text{max number observed is } m) = \sum_{m=n}^{k} m\frac{\binom{m-1}{n-1}}{\binom{k}{n}} = \frac{n(k+1)}{n+1},$$

where in the last step, we evaluate the expression using a calculator. Thus, the bias of the ML estimator for k in this case is

$$\mathbb{E}[\widehat{k}_{\text{ML}}(X^{(1)}, \ldots, X^{(n)})] = \sum_{m=n}^{k} m\mathbb{P}(\text{max number observed is } m) = \sum_{m=n}^{k} m\frac{\binom{m-1}{n-1}}{\binom{k}{n}} = \frac{n(k+1)}{n+1},$$

where as a reminder, note that $n \leqslant k$. This means that the ML estimator on average guesses a value of k that is lower than the true value. This makes sense since if we just take the maximum of the numbers on the tanks we've seen, then unless we see the tank with number k, the maximum we observe is going to be smaller than the true maximum. Of course, if we see all the tanks, i.e., if $n = k$, then we definitely see the tank with max number k so the bias in this case would be 0.

(c) The bias is given by

$$\mathbb{E}[\widehat{k}_{\text{ML}}] - k = \tfrac{n-k}{n+1} = \tfrac{n}{n+1} - \tfrac{k}{n+1}.$$

Thus,

$$\mathbb{E}[\widehat{k}_{\text{ML}}] - \tfrac{n}{n+1} = \underbrace{\left(1 - \frac{1}{n+1}\right)}_{\frac{n}{n+1}} k.$$

So

$$k = \underbrace{\frac{n+1}{n}}_{1+\frac{1}{n}} \mathbb{E}[\widehat{k}_{\text{ML}}] - 1.$$

Thus, an estimator

$$\widehat{k} = \left(1 + \tfrac{1}{n}\right)\widehat{k}_{\text{ML}} - 1$$

will be unbiased since, by linearity of expectation,

$$\mathbb{E}[\widehat{k}] = \left(1 + \tfrac{1}{n}\right)\mathbb{E}[\widehat{k}_{\text{ML}}] - 1.$$

3.2.3 The Bayesian Approach to Learning Parameters

TODO – add notes from video

3.2.4 Parameter Learning: What's Next

The single-variable calculus tools for showing what the maximum likelihood or even what the MAP estimate is are not easily going to generalize when we have a large number of parameters that we want to estimate, especially to justify why a particular estimate is actually the argmax. We will soon turn to deriving maximum likelihood estimates using information theoretic measures. Before heading there though, we will be covering a simple way of doing predictions with what's called the naive Bayes classifier, which you'll be using to build an email spam detector.

3.3 Parameter Learning - Naive Bayes Classification

3.3.1 The Naive Bayes Classifier: Introduction

Previously, we saw how maximum likelihood estimation works for some simple cases. Now, we look at how it works for a more elaborate setup, specifically in the problem of email spam detection. Note that in this section, for simplicity, in showing the maximum likelihood estimates, we will just be setting derivatives equal to 0 without checking second derivatives and boundaries.

We want to build a classifier that, given an email, classifies it as either spam or ham. How do we go about doing the classification? There are many ways to classify data. Today, we're going to talk about one such way called the *naive Bayes classifier*, which uses a simple classification model and has two algorithms to go with it: the first algorithm learns the naive Bayes model parameters from training data, and the second algorithm, given the parameters learned, predicts whether a new email is spam or ham.

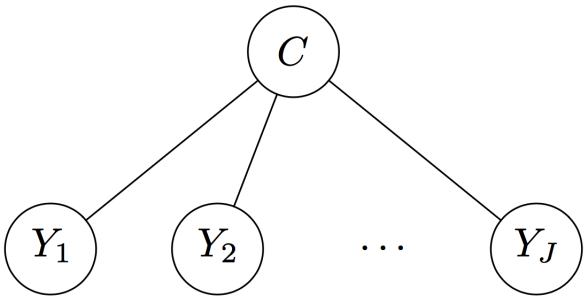
Suppose we have n training emails. Email i has a known label $c^{(i)} \in \{\text{spam}, \text{ham}\}$. Also, from email i, we extract J features $y^{(i)} = (y_1^{(i)}, y_2^{(i)}, \ldots, y_J^{(i)})$. In particular, for simplicity, we shall assume that each $y_j^{(i)} \in \{0,1\}$indicates the presence of the j-th word in some dictionary of J words. (Of course, you could use much fancier features such as

vector-valued features or even features with non-numerical values, but we'll stick to 0's and 1's that indicate presence of certain words.) For example, maybe the first word in the dictionary is "viagra" in which case y1(i) is 1 if email i contains the word "viagra" and 0 otherwise. In summary, our training data consists of $y^{(1)}, y^{(2)}, \ldots, y^{(n)} \in \{0,1\}^J$ with respective labels $c^{(1)}, c^{(2)}, \ldots, c^{(n)} \in \{\text{spam}, \text{ham}\}$.

Next, we specify a probabilistic model that explains how an email is hypothetically generated:

1. Sample a random label C that is equal to spam with probability s and equal to ham with probability $1-s$. (For example, you could flip a coin with probability of heads s, and if the coin comes up heads, you assign $C = spam$, and otherwise you assign $C = ham$.)
2. For $j = 1, 2, \ldots, J$: Sample $Y_j \sim \text{Ber}(q_j)$ if $C = spam$. Otherwise, sample $Y_j \sim \text{Ber}(p_j)$ if $C = ham$.

Note that the chance of a word from the dictionary occurring depends on whether the email is spam or ham, much like how you'd imagine that if you see the word "viagra" in an email, the email's probably spam rather than ham. The above recipe for generating features for a hypothetical email is called a *generative process*, and its corresponding graphical model is as follows:



Important observation: The Y_i's are independent given C. This assumption of the model is not actually true for emails since certain words may be more likely to co-occur. Also, the model does not account for the ordering of the words or whether a word occurs multiple times. While the naive Bayes classifier does make these assumptions, in practice, it is often applied to data that violates these assumptions, yet the performance of the classifier is still quite good! To quote statistician George Box, "All models are wrong but some are useful."

We need to estimate the parameters $\theta = \{s, p_1, p_2, \ldots, p_J, q_1, q_2, \ldots, q_J\}$. We shall assume that our training data $(c^{(i)}, y^{(i)})$ for each i are generated i.i.d. according to the above generative process. Then, to learn the parameters, we find θ that maximizes the likelihood, i.e.:

$$\widehat{\theta} = \arg\max_\theta \prod_{i=1}^n p_{C,Y_1,\ldots,Y_J}(c^{(i)}, y_1^{(i)}, \ldots, y_J^{(i)}; \theta).$$

3.3.2 The Naive Bayes Classifier: Training

Note: "log" means natural log in these notes.

Practice problem: Show that the log likelihood for our email spam detection setup can be written as

$$\log\left(\prod_{i=1}^n p_{C,Y_1,\ldots,Y_J}(c^{(i)}, y_1^{(i)}, \ldots, y_J^{(i)}; \theta)\right) = f(s) + \sum_{j=1}^J g_j(p_j) + \sum_{j=1}^J h_j(q_j)$$

for some functions $f, g_1, g_2, \ldots, g_J, h_1, h_2, \ldots, h_J$. What are these functions? Note that what the above equation is saying is that the log likelihood decouples into functions where each function depends on just one of the parameters. Thus, when we want to maximize over θ, what we can do is maximize over each parameter in θ separately! For example, to find what the ML estimate for s, we only need to look at (s)

Hint: To show the above equation, you may find it helpful that we can write:

$$\begin{aligned} p_C(c;\theta) &= s^{\mathbf{1}\{c=\text{spam}\}}(1-s)^{1-\mathbf{1}\{c=\text{spam}\}} \qquad \text{for } c \in \{\text{spam}, \text{ham}\} \\ p_{Y_j|C}(y_j \mid \text{ham};\theta) &= p_j^{y_j}(1-p_j)^{1-y_j} \qquad \text{for } y_j \in \{0,1\} \\ p_{Y_j|C}(y_j \mid \text{spam};\theta) &= q_j^{y_j}(1-q_j)^{1-y_j} \qquad \text{for } y_j \in \{0,1\} \end{aligned}$$

Practice problem: After you figure out what the functions f, g_j's, and h_j's are, obtain the ML estimate for each of the parameters $s, p_1, \ldots, p_J, q_1, \ldots, q_J$ by setting derivatives equal to 0.

Hint: You may find it helpful that for nonzero constants A and B,

$$\tfrac{d}{dt}\{A\log t + B\log(1-t)\} = 0 \qquad \text{when} \qquad t = \tfrac{A}{A+B}.$$

Simplifying the log likelihood: The log likelihood is given by

$$\begin{aligned} &\log\left(\prod_{i=1}^{n} p_{C,Y_1,\ldots,Y_J}(c^{(i)}, y_1^{(i)}, \ldots, y_J^{(i)};\theta)\right) \\ &= \log\left(\prod_{i=1}^{n}\left[p_C(c^{(i)};\theta)\prod_{j=1}^{J} p_{Y_j|C}(y_j^{(i)}|c^{(i)};\theta)\right]\right) \\ &= \sum_{i=1}^{n}\left[\log p_C(c^{(i)};\theta) + \sum_{j=1}^{J}\log p_{Y_j|C}(y_j^{(i)}|c^{(i)};\theta)\right] \\ &= \underbrace{\sum_{i=1}^{n}\log p_C(c^{(i)};\theta)}_{(*)} + \underbrace{\sum_{i=1}^{n}\sum_{j=1}^{J}\log p_{Y_j|C}(y_j^{(i)}|c^{(i)};\theta)}_{(**)}. \end{aligned}$$

We next simplify the expressions (*) and (**).

First let's simplify term (*):

$$\begin{aligned} (*) &= \sum_{i=1}^{n}\log p_C(c^{(i)};\theta) \\ &= \sum_{i=1}^{n}[\mathbf{1}\{c = \text{“spam”}\}\log s + \mathbf{1}\{c = \text{“ham”}\}\log(1-s)] \\ &= \left[\sum_{i=1}^{n}\mathbf{1}\{c = \text{“spam”}\}\right]\log s + \left[\sum_{i=1}^{n}\mathbf{1}\{c = \text{“ham”}\}\right]\log(1-s) \\ &\triangleq f(s). \end{aligned}$$

Next, we simplify (**), splitting it up as to decouple p_j and q_j. To do this, we can split the summation over i into two sums, one accounting for all the ham emails and one accounting for all the spam emails:

$$
\begin{aligned}
&(**)\\
&= \sum_{i=1}^{n}\sum_{j=1}^{J} \log p_{Y_j|C}(y_j^{(i)}|c^{(i)};\theta)\\
&= \sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} \sum_{j=1}^{J} \log p_{Y_j|C}(y_j^{(i)}|c^{(i)};\theta)\\
&\quad + \sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} \sum_{j=1}^{J} \log p_{Y_j|C}(y_j^{(i)}|c^{(i)};\theta)\\
&= \sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} \sum_{j=1}^{J} \left[y_j^{(i)} \log p_j + (1-y_j^{(i)}) \log(1-p_j) \right]\\
&\quad + \sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} \sum_{j=1}^{J} \left[y_j^{(i)} \log q_j + (1-y_j^{(i)}) \log(1-q_j) \right]\\
&= \sum_{j=1}^{J} \underbrace{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} \left[y_j^{(i)} \log p_j + (1-y_j^{(i)}) \log(1-p_j) \right]}_{\triangleq g_j(p_j)}\\
&\quad + \sum_{j=1}^{J} \underbrace{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} \left[y_j^{(i)} \log q_j + (1-y_j^{(i)}) \log(1-q_j) \right]}_{\triangleq h_j(q_j)}.
\end{aligned}
$$

In summary:

$$
\begin{aligned}
f(s) &= \left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} \right] \log s + \left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} \right] \log(1-s),\\
g_j(p_j) &= \left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} y_j^{(i)} \right] \log p_j + \left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} (1-y_j^{(i)}) \right] \log(1-p_j),\\
h_j(q_j) &= \left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} y_j^{(i)} \right] \log q_j + \left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} (1-y_j^{(i)}) \right] \log(1-q_j).
\end{aligned}
$$

Setting derivatives to 0: The ML estimate for s is $\widehat{s} = \arg\max_{s\in[0,1]} f(s)$, which occurs when $\frac{df}{ds} = 0$. Using the hint, we see that

$$
f(s) = \underbrace{\left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} \right]}_{A} \log s + \underbrace{\left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} \right]}_{B} \log(1-s)
$$

has derivative equal to 0 when

$$
\widehat{s} = \frac{A}{A+B} = \frac{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\}}{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} + \sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\}} = \frac{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\}}{n}.
$$

This result is intuitive – it's the number of emails labeled “spam” divided by the total number of emails.

The ML estimate for p_j is $\widehat{p}_j = \arg\max_{p_j\in[0,1]} g_j(p_j)$, which occurs when $\frac{dg_j}{dp_j} = 0$. Again using the hint, we see that

$$g_j(p_j) = \underbrace{\left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} y_j^{(i)}\right]}_{A} \log p_j + \underbrace{\left[\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\}(1 - y_j^{(i)})\right]}_{B} \log(1 - p_j)$$

has derivative equal to 0 when

$$\begin{aligned}
\widehat{p}_j &= \frac{A}{A+B} \\
&= \frac{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} y_j^{(i)}}{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} y_j^{(i)} + \sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\}(1 - y_j^{(i)})} \\
&= \frac{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\} y_j^{(i)}}{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“ham”}\}}.
\end{aligned}$$

This result is also intuitive – it's the number of times word j occurred in an email labeled “ham” divided by the total number of emails labeled “ham”.

Finally, by pattern-matching, the ML estimate for q_j is

$$\widehat{q}_j = \frac{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\} y_j^{(i)}}{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = \text{“spam”}\}}.$$

Wonderful, now we can write up an algorithm that computes all those ML estimates above. Once we learn the parameters θ, we can treat them as fixed and start doing prediction.

3.3.3 The Naive Bayes Classifier: Prediction

Practice problem: Once we learn the parameters θ, we can treat them as fixed and start doing prediction. Let's now look at classifying whether a new email that's not in our training data is spam or ham. This new email has random, unobserved label C, which we would like to infer, but we only get to see its features $Y_1 = y_1, Y_2 = y_2, \ldots, Y_J = y_J$. Assuming that θ is known and fixed, figure out what the MAP estimate for label C is given $Y_1 = y_1, Y_2 = y_2, \ldots, Y_J = y_J$.

Unleashing Bayes' rule,

$$\begin{aligned}
&p_{C|Y_1,\ldots,Y_J}(\text{“spam”}|y_1,\ldots,y_J) \\
&= \frac{p_C(\text{“spam”}) p_{Y_1,\ldots,Y_J|C}(y_1,\ldots,y_J|\text{“spam”})}{p_{Y_1,\ldots,Y_J}(y_1,\ldots,y_J)} \\
&= \frac{p_C(\text{“spam”}) p_{Y_1,\ldots,Y_J|C}(y_1,\ldots,y_J|\text{“spam”})}{p_C(\text{“spam”}) p_{Y_1,\ldots,Y_J|C}(y_1,\ldots,y_J|\text{“spam”}) + p_C(\text{“ham”}) p_{Y_1,\ldots,Y_J|C}(y_1,\ldots,y_J|\text{“ham”})} \\
&= \frac{p_C(\text{“spam”}) \prod_{j=1}^{J} p_{Y_j|X}(y_j|\text{“spam”})}{p_C(\text{“spam”}) \prod_{j=1}^{J} p_{Y_j|C}(y_j|\text{“spam”}) + p_C(\text{“ham”}) \prod_{j=1}^{J} p_{Y_j|X}(y_j|\text{“ham”})} \\
&= \frac{s \prod_{j=1}^{J} q_j^{y_j} (1-q_j)^{1-y_j}}{s \prod_{j=1}^{J} q_j^{y_j} (1-q_j)^{1-y_j} + (1-s) \prod_{j=1}^{J} p_j^{y_j} (1-p_j)^{1-y_j}},
\end{aligned}$$

where for simplicity we've dropped the hats on the parameters even though the parameter values we use are estimated from training data.

Of course,

$$
\begin{aligned}
&p_{C|Y_1,\ldots,Y_J}(\text{“ham”}|y_1,\ldots,y_J) \\
&= 1 - p_{C|Y_1,\ldots,Y_J}(\text{“spam”}|y_1,\ldots,y_J) \\
&= \frac{(1-s)\prod_{j=1}^J p_j^{y_j}(1-p_j)^{1-y_j}}{s\prod_{j=1}^J q_j^{y_j}(1-q_j)^{1-y_j} + (1-s)\prod_{j=1}^J p_j^{y_j}(1-p_j)^{1-y_j}}.
\end{aligned}
$$

The MAP estimate for C is

$$
\widehat{C}_{\text{MAP}} = \begin{cases} \text{“spam”} & \text{if } p_{C|Y_1,\ldots,Y_J}(\text{“spam”}|y_1,\ldots,y_J) \geqslant p_{C|Y_1,\ldots,Y_J}(\text{“ham”}|y_1,\ldots,y_J) \\ \text{“ham”} & \text{otherwise.} \end{cases}
$$

Note that here we're breaking ties in favor of spam. The above is equivalent to looking at whether the odds ratio

$$
\frac{p_{C|Y_1,\ldots,Y_J}(\text{“spam”}|y_1,\ldots,y_J)}{p_{C|Y_1,\ldots,Y_J}(\text{“ham”}|y_1,\ldots,y_J)}
$$

is at least 1, or whether the log odds ratio

$$
\log \frac{p_{C|Y_1,\ldots,Y_J}(\text{“spam”}|y_1,\ldots,y_J)}{p_{C|Y_1,\ldots,Y_J}(\text{“ham”}|y_1,\ldots,y_J)}
$$

is at least 0. In practice the log odds ratio can be much more numerically stable to compute since, pushing in the log, we end up taking sums and differences of log probabilities rather than multiplying a large number of probabilities.

3.3.4 The Naive Bayes Classifier: Laplace Smoothing

Practice problem: Assume that a particular word, say word 1, did not appear in any of the training data. Now for the email we're trying to predict the label for, suppose that we observe that $y_1 = 1$. What is $p_{Y_1,\ldots,Y_J}(y_1,\ldots,y_J)$? What went wrong? Can you think of a way to modify the prediction step to address this issue?

Assume that a particular word, say word 1, did not appear in any of the training data. Now for the email we're trying to predict the label for, suppose that we observe that $y_1 = 1$. We would have learned that $p_1 = q_1 = 0$ (we're leaving hats off to keep notation from getting cluttered). Using the result of the prediction phase, we see that

$$
p_{Y_1,\ldots,Y_J}(y_1,\ldots,y_J) = s\prod_{j=1}^J q_j^{y_j}(1-q_j)^{1-y_j} + (1-s)\prod_{j=1}^J p_j^{y_j}(1-p_j)^{1-y_j} = 0
$$

since $y_1 = 1$ while $p_1 = q_1 = 0$. In other words, the observation is impossible given the model learned! Hence, computing $\mathbb{P}(C = \text{spam}|Y_1 = 1, Y_2 = y_2, \ldots, Y_J = y_J)$ doesn't make sense since we're conditioning on an event that has 0 probability.

Thus, ML estimates aren't robust in this case to words that we don't encounter in training data.

One way to resolve this is to take a Bayesian approach to parameter estimation (we saw this earlier when we put a prior on the bias of a coin) and, in particular, introduce pseudocounts. For example, we could set:

$$
\begin{aligned}
\widehat{s} &= \frac{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“spam”}\} + 1}{n+2}, \\
\widehat{p}_j &= \frac{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“ham”}\} y_j^{(i)} + 1}{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“ham”}\} + 2}, \\
\widehat{q}_j &= \frac{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“spam”}\} y_j^{(i)} + 1}{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“spam”}\} + 2}.
\end{aligned}
$$

This ensures that none of the parameters are estimated as 0 or 1.

Note that introducing a pseudocount for each possible outcome has a special name: Laplace smoothing, also called additive smoothing. For $\widehat{s}$, this means introducing 1 pseudocount for spam and 1 pseudocount for ham. For $\widehat{p}_j$, this means introducing 1 pseudocount for word j appearing in ham and 1 pseudocount for word j not appearing in ham. You could, of course, also introduce ℓ pseudocounts for each possible outcome instead of just one, i.e.,

$$\begin{aligned}
\widehat{s} &= \frac{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“spam”}\} + \ell}{n + 2\ell}, \\
\widehat{p}_j &= \frac{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“ham”}\} y_j^{(i)} + \ell}{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“ham”}\} + 2\ell}, \\
\widehat{q}_j &= \frac{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“spam”}\} y_j^{(i)} + \ell}{\sum_{i=1}^n \mathbf{1}\{c^{(i)} = \text{“spam”}\} + 2\ell}.
\end{aligned}$$

3.4 Parameter Learning - Finite Random Variables and Trees

3.4.1 Generalizing Parameter Learning

This section generalizes the ideas we've seen with parameter learning for a biased coin and for naive Bayes, specifically for maximum likelihood (so, no Laplace smoothing now). Specifically we look at learning parameters for a general finite random variable (for which estimating the bias of a coin is a special case) and learning all the potential tables in a tree-structured graphical model (for which naive Bayes is a special case).

For the finite random variable case, our derivation will revisit information measures from earlier in the course: entropy and information divergence. We'll see that maximum likelihood estimation relates to what we've seen earlier in the course on histograms – frequencies in which we see outcomes occur.

Learning all the potential tables in a tree-structured graphical model can be phrased in terms of learning parameters for a collection of finite random variables.

Unlike the bulk of our previous coverage of parameter learning, we won't be taking derivatives in this section!

3.4.2 Parameter Learning for a Finite Random Variable

Let's build off of our coin tossing example, where we had an underlying distribution p_X with parameter θ, the probability of heads. We now explicitly make there be two parameters θ_{heads} and θ_{tails}. In the more general case when there are many outcomes, it will be easier to write out a parameter for every value a random variable can take on rather than a parameter for all but one of them. Thus, using the same representation trick we saw earlier:

$$p_X(x;\theta) = \theta_{\text{heads}}^{\mathbf{1}\{x=\text{heads}\}} \theta_{\text{tails}}^{\mathbf{1}\{x=\text{tails}\}} = \begin{cases} \theta_{\text{heads}} & \text{if } x = \text{heads}, \\ \theta_{\text{tails}} & \text{if } x = \text{tails}. \end{cases}$$

In particular, in the general case when random variable X has alphabet $\mathscr{X}$, then

$$p_X(x;\theta) = \textstyle\prod_{a\in\mathscr{X}} \theta_a^{\mathbf{1}\{x=a\}}.$$

(In the coin tossing case, $\mathscr{X} = \{\text{heads}, \text{tails}\}$.)

The training data $X^{(1)}, \ldots, X^{(n)}$ are again assumed to be drawn i.i.d. from the distribution $p_X(\cdot;\theta)$, so that the likelihood for observed data $X^{(1)} = x^{(1)}, \ldots, X^{(n)} = x^{(n)}$ is

$$\textstyle\prod_{i=1}^n p_X(x^{(i)};\theta) = \prod_{i=1}^n \left\{ \prod_{a\in\mathscr{X}} \theta_a^{\mathbf{1}\{x^{(i)}=a\}} \right\}.$$

The log likelihood is thus

$$\log \prod_{i=1}^n p_X(x^{(i)};\theta) = \log \prod_{i=1}^n \left\{ \prod_{a\in\mathscr{X}} \theta_a^{\mathbf{1}\{x^{(i)}=a\}} \right\} = \sum_{i=1}^n \sum_{a\in\mathscr{X}} \mathbf{1}\{x^{(i)}=a\} \log \theta_a.$$

Now we exchange the ordering of the summations on the right-hand side to get

$$\text{log likelihood} = \sum_{a\in\mathscr{X}} \left[\sum_{i=1}^n \mathbf{1}\{x^{(i)}=a\} \right] \log \theta_a.$$

Note that the maximum likelihood estimate $\widehat{\theta}$ of all the parameters θ (now θ has a parameterθ_a for each possible value $a \in \mathscr{X}$) is precisely the solution to the following constrained optimization problem:

$$\widehat{\theta} = \arg \max_{\theta=\{\theta_a \text{ for } a\in\mathscr{X}\}} \left\{ \sum_{a\in\mathscr{X}} \left[\sum_{i=1}^n \mathbf{1}\{x^{(i)}=a\} \right] \log \theta_a \right\}$$
$$\text{subject to } \sum_{a\in\mathscr{X}} \theta_a = 1, \text{ and } \theta_a \geqslant 0 \text{ for all } a \in \mathscr{X}.$$

(Technical detail: If we didn't have the inequality constraints, then it's possible to solve this using a single Lagrange multiplier to enforce the equality constraint that $\sum_{a\in\mathscr{X}} \theta_a = 1$. If you do this, you will in fact get the correct answer, but it's not straightforward showing why it is definitely the unique global maximum.)

Let's see how information-theoretic measures can help us. Let $\widehat{p}_X$ refer to the empirical distribution (this is the histogram of frequencies) of X that we get from our training data $x^{(1)},\ldots,x^{(n)}$, so that $\widehat{p}_X(x)$ is precisely the fraction of times we saw x in the training data:

$$\widehat{p}_X(x) = \tfrac{1}{n} \sum_{i=1}^n \mathbf{1}\{x^{(i)}=x\}.$$

Then the log likelihood is

$$\begin{aligned}
&\text{log likelihood} \\
&= \sum_{a\in\mathscr{X}} \underbrace{\left[\sum_{i=1}^n \mathbf{1}\{x^{(i)}=a\} \right]}_{n\cdot\widehat{p}_X(a)} \log \underbrace{\theta_a}_{p_X(a;\theta)} \\
&= \sum_{a\in\mathscr{X}} n\cdot\widehat{p}_X(a) \log p_X(a;\theta) \\
&= \sum_{a\in\mathscr{X}} n\cdot\widehat{p}_X(a) \log \Big(p_X(a;\theta) \frac{\widehat{p}_X(a)}{\widehat{p}_X(a)} \Big) \\
&= \sum_{a\in\mathscr{X}} n\cdot\widehat{p}_X(a) \Big(\log \widehat{p}_X(a) + \log \frac{p_X(a;\theta)}{\widehat{p}_X(a)} \Big) \\
&= n \Bigg[\sum_{a\in\mathscr{X}} \widehat{p}_X(a) \log \widehat{p}_X(a) + \sum_{a\in\mathscr{X}} \widehat{p}_X(a) \log \frac{p_X(a;\theta)}{\widehat{p}_X(a)} \Bigg] \\
&= n\Big[-H(\widehat{p}_X) - D\big(\widehat{p}_X \parallel p_X(\cdot;\theta)\big) \Big] \\
&= -n\Big[H(\widehat{p}_X) + D\big(\widehat{p}_X \parallel p_X(\cdot;\theta)\big) \Big].
\end{aligned}$$

Note that here we are using natural log with entropy and information divergence instead of log base 2 – this is actually commonly done and just results in the overall quantity being scaled by a constant (since $\log_2 x = \frac{\log x}{\log 2}$). Recall that when we used log base 2, we measured information content in terms of bits. When using natural log, information content is measured in terms of what are called nats.

Importantly, note that to maximize the likelihood, the only expression that depends on θ here is the divergence, and in particular, maximizing the likelihood is the same as (because of the minus sign in the last expression) minimizing

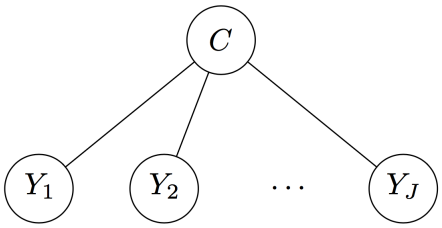
the divergence $D\big(\widehat{p}_X \,\|\, p_X(\cdot;\theta)\big)$. Here, since we treat our training data as fixed, then $\widehat{p}_X$ is a fixed distribution, and so minimizing the divergence means choosing a distribution $p_X(\cdot;\theta)$. But we know from Gibbs' inequality what the best choice of $p_X(\cdot;\theta)$ is! In particular, the divergence is the smallest possible and, in particular, 0 when $p_X(\cdot;\theta)$ is set to be equal to the empirical distribution $\widehat{p}_X$, so we set

$$\underbrace{p_X(a;\theta)}_{\theta_a} = \widehat{p}_X(a) \qquad \text{for all } a \in \mathscr{X}.$$

So the maximum likelihood estimate $\widehat{\theta}_a$ for parameter θ_a is the fraction of times we see a in the training data $x^{(1)},\ldots,x^{(n)}$. Note that Gibbs' inequality tells us that we cannot do better. The unique global maximum is to choose $\widehat{\theta}_a = \widehat{p}_X(a)$ for all $a \in \mathscr{X}$. There is no need to check second derivatives, boundaries, etc. (Technical note: When we have more than a single variable, justifying whether a point is a local maximum in general requires more work than in the single variable calculus case since there could be saddle points. However, in this case, Gibbs' inequality tells us that there's a unique maximum.)

3.4.3 Parameter Learning for an Undirected Tree-Structured Graphical Model

We now look at how to learn potential tables for a tree-structured graphical model. Recall that for the exact same distribution, the way in which we specify potential tables is not unique (i.e., there could be multiple way to specify potential tables to represent the same distribution). This is fine. We can still learn potential tables systematically with maximum likelihood. In fact, we did this already when we trained a naive Bayes classifier! Let's build some intuition from the naive Bayes classifier case (again, we stick to maximum likelihood here – no Laplace smoothing). The graphical model was as follows:



How we learned the parameters was that we treated node C as the root node. Then the factorization was

$$\underbrace{p_C(c;\theta)}_{\text{1 potential table}} \underbrace{\prod_{j=1}^{J} p_{Y_j|C}(y_j \mid c;\theta)}_{J \text{ potential tables}}.$$

Each of the potential tables is actually specified in terms of different parameters. In particular, p_C only depends on parameter s, and $p_{Y_j|C}$ only depends on p_j and q_j.

What we did was we estimated $p_C(\cdot;\theta)$ with the empirical distribution

$$\widehat{p}_C(c) = \text{fraction of times in training data we see label } c = \tfrac{1}{n}\textstyle\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = c\}.$$

When c = spam, we called this parameter s. Of course when c = ham, then this parameter is just $1-s$, which we didn't have to separately store.

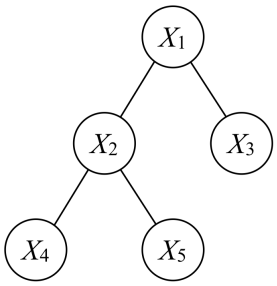
We estimated $p_{Y_j|C}(\cdot \mid c;\theta)$ with the empirical conditional distribution

$$\begin{aligned}\widehat{p}_{Y_j|C}(y_j \mid c) &= \text{fraction of times in training data we see } y_j \text{ amongst emails with label } c \\ &= \frac{\sum_{i=1}^{n} \mathbf{1}\{y_j^{(i)} = y_j, c^{(i)} = c\}}{\sum_{i=1}^{n} \mathbf{1}\{c^{(i)} = c\}}.\end{aligned}$$

When c = ham, we called this parameter p_j. When c = spam, we called this parameter q_j.

For the general case of learning maximum likelihood parameters for a given tree-structured graphical model, we can choose an arbitrary root node (which has a node potential table corresponding to the probability of the root node)

and then treat the edge potential tables as conditional probability distributions. For the root node, we estimate its node potential table with the empirical distribution for just that node. For the other potential tables, we estimate them using empirical conditional distributions. Let me walk through a specific example. Consider the following graphical model:



We assume we have access to training data

$$\underbrace{X_1^{(1)}, X_2^{(1)}, X_3^{(1)}, X_4^{(1)}, X_5^{(1)}}_{\text{first training data point}}, \quad \underbrace{X_1^{(2)}, X_2^{(2)}, X_3^{(2)}, X_4^{(2)}, X_5^{(2)}}_{\text{second training data point}}, \quad \ldots, \quad \underbrace{X_1^{(n)}, X_2^{(n)}, X_3^{(n)}, X_4^{(n)}, X_5^{(n)}}_{n\text{-th training data point}}.$$

The likelihood is

$$\textstyle\prod_{i=1}^{n} p_{X_1,X_2,X_3,X_4,X_5}(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, x_4^{(i)}, x_5^{(i)}; \theta).$$

By choosing node 1 arbitrarily as the root, then we write $p_{X_1,X_2,X_3,X_4,X_5}(\cdot;\theta)$ as

$$\underbrace{p_{X_1,X_2,X_3,X_4,X_5}}_{\text{parameters } \theta} = \underbrace{p_{X_1}}_{\text{parameters } \theta_1} \underbrace{p_{X_2|X_1}}_{\theta_{2|1}} \underbrace{p_{X_3|X_1}}_{\theta_{3|1}} \underbrace{p_{X_4|X_2}}_{\theta_{4|2}} \underbrace{p_{X_5|X_2}}_{\theta_{5|2}},$$

where we use $\theta_1, \theta_{2|1}, \theta_{3|1}, \theta_{4|2}, \theta_{5|2}$ to refer to parameters for each of the tables that we aim to learn. The notation here is going to be a bit messy. We'll use the notation

$$\theta_{1;a} \triangleq p_{X_1}(a;\theta)$$
$$\theta_{2|1;a|b} \triangleq p_{X_2|X_1}(a \mid b;\theta_{2|1})$$

and so forth.

We assume these parameters to be separate from each other (so that similar to the naive Bayes case, we can learn each of these tables separately). Then the log likelihood is

$$\begin{aligned}
&\log \prod_{i=1}^{n} p_{X_1,X_2,X_3,X_4,X_5}(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, x_4^{(i)}, x_5^{(i)}; \theta) \\
&= \log \prod_{i=1}^{n} \big\{ p_{X_1}(x_1^{(i)};\theta_1) p_{X_2|X_1}(x_2^{(i)} \mid x_1^{(i)};\theta_{2|1}) p_{X_3|X_1}(x_3^{(i)} \mid x_1^{(i)};\theta_{3|1}) \\
&\qquad p_{X_4|X_2}(x_4^{(i)} \mid x_2^{(i)};\theta_{4|2}) p_{X_5|X_2}(x_5^{(i)} \mid x_2^{(i)};\theta_{5|2}) \big\} \\
&= \text{log likelihood for } X_1 \text{ with parameter } \theta_1 \\
&\quad + \sum_a \mathbf{1}\{x_1^{(i)} = a\} \cdot (\text{log likelihood for } X_2 \mid X_1 = a \text{ with parameter } \theta_{2|1}) \\
&\quad + \sum_a \mathbf{1}\{x_1^{(i)} = a\} \cdot (\text{log likelihood for } X_3 \mid X_1 = a \text{ with parameter } \theta_{3|1}) \\
&\quad + \sum_a \mathbf{1}\{x_2^{(i)} = a\} \cdot (\text{log likelihood for } X_4 \mid X_2 = a \text{ with parameter } \theta_{4|2}) \\
&\quad + \sum_a \mathbf{1}\{x_2^{(i)} = a\} \cdot (\text{log likelihood for } X_5 \mid X_2 = a \text{ with parameter } \theta_{5|2})
\end{aligned}$$

To maximize the overall log likelihood, we maximize the log likelihood for each of the five tables separately, where the root node corresponds to learning the parameter for a single finite random variable, whereas for each of the edges, we learn a finite random variable for every possible value that we are conditioning on. In both of these cases, the result from parameter learning for a finite random variable (for maximum likelihood) says that the ML estimate corresponds to fitting an empirical distribution.

Thus, for the root node, we have

$$\widehat{\theta}_{1;a} = \widehat{p}_{X_1}(a) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{x_1^{(i)} = a\},$$

and for the conditional probability tables, we have

$$\widehat{\theta}_{i|j;a|b} = \widehat{p}_{X_i|X_j}(a \mid b) = \frac{\sum_{\ell=1}^{n} \mathbf{1}\{x_i^{(\ell)}=a, x_j^{(\ell)}=b\}}{\sum_{\ell=1}^{n} \mathbf{1}\{x_j^{(\ell)}=b\}}.$$

Note that if we chose the root node to be some other node, of course, the factorization we get would be different, which means that the tables we are learning will be different but will represent the same distribution.

3.5 Structure Learning – Trees

3.5.1 Structure Learning for an Undirected Tree-Structured Graphical Model: The Chow-Liu Algorithm

At this point we've covered parameter learning for undirected trees, where we assume we know the tree structure. But what if we don't know what tree to even use? We now look at an algorithm called the Chow-Liu algorithm that learns which tree to use from training data, again using maximum likelihood. Once more, information measures appear, where mutual information plays a pivotal role. Recall that mutual information tells us how far two random variables are from being independent. A key idea here is that to determine whether an edge between two random variables should be present in a graphical model, we can look at mutual information.

The idea is that we have random variables X_1 up through X_k. These you can think of as nodes in a undirected graphical model. But we don't know which edges should be in this graphical model. And restricting ourself to trees, the question is, which tree should we use? To help us answer this question, we're going to have access to training data. We see n i.i.d. samples from this graphical model, where for each of the samples, of course, we see everything, $x_1^{(i)}, \ldots, x_k^{(i)}$. So this is the i-th training data point and $i = 1 \ldots n$. Just as a comment, the number of possible trees here is humongous, actually k^{k-2}. This from a result called Cayley's formula, which grows even faster than exponential, it's super exponential. Somehow, we're going to search this huge space of possible trees to select the best tree. It turns out, doing that can be done very quickly. To select the tree we're going to use maximum likelihood. We're going to pick the tree T here.

$$\widehat{T} = \arg\max_{T} \left\{ \max_{\theta_T} \log \prod_{i=1}^{n} p_{x_1 \ldots x_k}(x_1^{(i)}, \ldots, x_k^{(i)}; T, \theta_T) \right\}$$

In this case, we know that there are k nodes, but where are the edges? You can think of T as just specifying which edges are present that result in a tree across k nodes, with $k-1$ edges. So we're trying to find the best tree such that once we fix a tree, then we can do maximum likelihood for that specific tree T.

To solve this optimization problem, it turns out there's a very clean algorithm. To get to it, we're going to simplify what's in the curly braces:

For specific tree T :

$$\begin{aligned}
&\max_{\theta_T} \log \prod_{i=1}^{n} p_{x_1 \ldots x_k}(x_1^{(i)}, \ldots, x_k^{(i)}; T, \theta_T) \\
= \quad & \log \prod_{i=1}^{n} \left[\widehat{p}(x_r^{(i)}) \prod_{j \neq r} \widehat{p}(x_r^{(i)} | x_{\pi(j)}^{(i)}) \right] \\
= \quad & \sum_{i=1}^{n} \log \widehat{p}(x_r^{(i)}) + \sum_{j \neq r} \sum_{i=1}^{n} \log \widehat{p}(x_r^{(i)} | x_{\pi(j)}^{(i)})
\end{aligned}$$

We have the log likelihood across the training data. Here, we're parameterizing by both what the tree is and then, once we know the tree, some set of parameters θ_T for the tree. We can arbitrarily choose a root r and drop the subscripts just to save some writing. So the factorization is $p(x_r^{(i)})$ times the product from all the nodes that are not equal to the root $p(x_r^{(i)} | x_{\pi(j)}^{(i)})$. We want to take the maximum over all the θ_T's, meaning we want to plug in the best possible values of the parameters into this expression. But we know from the parameter learning video that the best choice of θ is going to correspond to the empirical distributions. All we need to do is replace the probability distributions p with the empirical versions $\widehat{p}$, which give us the desired max. As a reminder, $\widehat{p}(x_r^{(i)})$ is the empirical distribution of random variable X_r. Evaluated at some value a this is just equal to the fraction of times a appears in the training data. So $\widehat{p}(x_r^{(i)}) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\{x_r^{(i)} = a\}$. This is just an example of how you get these empirical distributions. The empirical conditional probability table, as a reminder, is going to be the fraction of times we see that these two random variables in our training data jointly take on two specific values divided by the number of times we see this random variable taking on a specific value in our training data. That's what these $\widehat{p}$'s are.

$$\begin{aligned}
& \sum_{i=1}^{n} \log \widehat{p}(x_r^{(i)}) + \sum_{j \neq r} \sum_{i=1}^{n} \log \widehat{p}(x_r^{(i)} | x_{\pi(j)}^{(i)}) \\
= \quad & \sum_{a} \left[\sum_{i=1}^{n} \mathbf{1}\{x_r^{(i)} = a\} \right] \log \widehat{p}_{x_r}(a) + \sum_{j \neq r} \sum_{a,b} \left[\sum_{i=1}^{n} \mathbf{1}\{x_j^{(i)} = a, x_{\pi(j)}^{(i)} = b\} \right] \log \widehat{p}_{x_j | x_{\pi(j)}}(a|b)
\end{aligned}$$

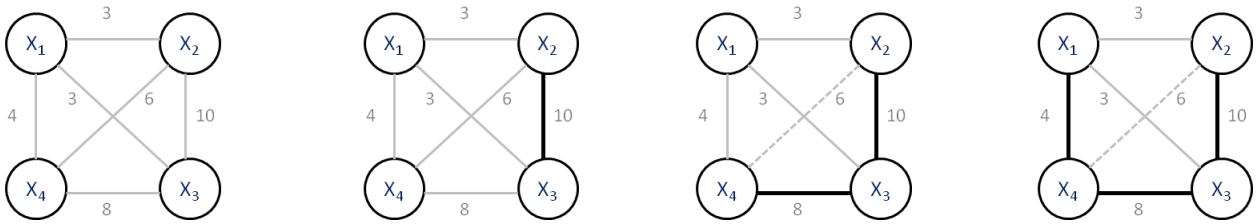
Next we introduce a different summation to split up what the possible values of this random variable are. We were doing this for the conditional distributions earlier. But now, we're going to do it for all of them, counting up how many times we see each particular value.

$$\begin{aligned}
& \sum_{a} \underbrace{\left[\sum_{i=1}^{n} \mathbf{1}\{x_r^{(i)} = a\} \right]}_{n p_{x_r}(a)} \log \widehat{p}_{x_r}(a) + \sum_{j \neq r} \sum_{a,b} \underbrace{\left[\sum_{i=1}^{n} \mathbf{1}\{x_j^{(i)} = a, x_{\pi(j)}^{(i)} = b\} \right]}_{n \widehat{p}_{x_j, x_{\pi(j)}}(a,b)} \log \widehat{p}_{x_j | x_{\pi(j)}}(a|b) \\
= \quad & n \Bigg[\underbrace{\sum_{a} p_{x_r}(a) \log \widehat{p}_{x_r}(a)}_{-H(\widehat{p}_{x_r})} + \underbrace{\sum_{j \neq r} \sum_{a,b} \widehat{p}_{x_j, x_{\pi(j)}}(a,b) \log \frac{\widehat{p}_{x_j, x_{\pi(j)}}(a,b)}{\widehat{p}_{x_{\pi(j)}}(b) \widehat{p}_{x_j}(a)}}_{\sum_{j \neq r} D(\widehat{p}_{x_j, x_{\pi(j)}})} \widehat{p}_{x_j}(a) \Bigg] \\
= \quad & n \Bigg[-H(\widehat{p}_{x_r}) + \sum_{j \neq r} \underbrace{D(\widehat{p}_{x_j, x_{\pi(j)}})}_{\widehat{I}(x_j, x_{\pi(j)})} + \underbrace{\sum_{a} \widehat{p}_{x_j}(a) \log \widehat{p}_{x_j}(a)}_{-H(\widehat{p}_{x_j}} \Bigg] \\
= \quad & n \Bigg[-\sum_{j \in V} H(\widehat{p}_{x_j}) + \sum_{(i,j) \in E} \widehat{I}(x_i; x_j) \Bigg]
\end{aligned}$$

The first term is related to the negative entropy. The log is natural log, but that's OK. We can still talk about the entropy, it's just in terms of "nats" not bits. The second term we multiply top and bottom by $\widehat{p}_{x_j}(a)$, which is just multiplying by 1. That gives us a term which is the information divergence $D(\widehat{p}_{x_j,x_{\pi(j)}})$. The basic idea is we're looking at how far the empirical joint distribution is from a product of two empirical marginal distributions. You can think of this as saying, how far is this empirical joint distribution from being independent. This is called the mutual information between these two random variables. But it's not actually these two random variables, it's the empirical versions of them. So we'll denote this as the empirical mutual information between x_j and $x_{\pi(j))}$. Again, the mutual information would be this exact same thing except with the true distributions, rather than the empirical distributions. Now, when we're looking at the empirical distributions, we just get this empirical mutual information instead. So that accounts for one piece of the second term. We have another piece, and this will become a minus entropy of the empirical distribution of X_j. Once we put all these pieces together, there is a minus entropy of the root node and a minus entropy for all the other nodes. So we're going to have a sum over all the nodes V of minus entropy of the empirical distribution of X_j. We're also going to have the empirical mutual information $\widehat{I}(x_i;x_j)$ for (i,j) within the edges in the tree.

The bottom line is this is equal to the maximum likelihood across parameters for a specific tree T. Note that to choose the best possible tree, the nodes aren't going to change, so the negative entropy term is going to be the same across any tree. The only thing that changes is the mutual information term that depends on the edges in the tree. So to figure out what the best tree is possible we just want to maximize this $\sum_{(i,j)\in E}\widehat{I}(x_i;x_j)$ term.

Now we want an algorithm that maximizes this across all possible trees. The basic idea of the algorithm is to start with a graph with no edges and incrementally add edges until we get the best tree according to maximum likelihood. For the first edge we add, we're going to look at all pairs of random variables to find which one has the highest empirical mutual information, and we're going to add that edge. Then we're going to repeat that process – we're going to find the pair of random variables with the second-highest empirical mutual information and so forth where we make sure that whenever we add an edge we don't introduce a cycle. And after we add enough edges we're going to get a tree and that's actually the best tree. This is called the Chow-Liu algorithm, illustrated with a simple four-node example.



Here we have $k = 4$ for $X_1 \ldots X_4$, and in gray are all the edges you could possibly have. For each of these edges we have a gray number, the empirical mutual information quantities. In practice these would be computed based on your training data. In the Chow-Liu algorithm we ask, what is the highest empirical mutual information? It's 10, so that is first edge to be added. The second highest edge is 8, which we add. Then in the next step, the third highest one is 6, but if we add this edge it will form a cycle, so skip that and move on to the next highest, which is 4, and we add that. At this point the algorithm would stop because adding anything else would form a cycle. So at this point the Chow-Liu algorithm would output a tree, denoted by the solid black lines, and that would be the maximum likelihood estimate for the best possible tree we could use.

The general idea is very simple. We start with a graph with no edges. And then for all pairs of random variables (i,j) we compute the empirical mutual information quantities $\widehat{I}(X_i;X_j)$, the gray numbers. Next we sort these empirical mutual information quantities from highest to lowest. And then starting from the highest and going all the way to the lowest we're going to add edges, except we skip adding an edge if it results in a cycle. And this algorithm will always terminate because for a graph with k nodes, any tree for that graph is going to have $k-1$ edges. For example in this case, any tree for this graph has 3 edges. That is the Chow-Liu algorithm. Given training data we can learn what is the maximum likelihood estimate for the tree that best explains the data.

3.5.2 Correctness and Running Time of the Chow-Liu Algorithm

These notes fill in details for why the Chow-Liu algorithm is correct and what its running time is. As a reminder, these notes are inherently more advanced and meant for those who are interested in the gory details.

We already justified why the maximum likelihood tree is the solution to the optimization problem

$$\widehat{T} = \arg\max_{T \text{ that is a tree over nodes } \{1,\ldots,k\}} \sum_{(i,j)\in T} \widehat{I}(X_i;X_j).$$

Here, we treat T as a set of edges, where edge (i,j) is the same as edge (j,i) so for simplicity we ask that $i < j$. As illustrated in the video, we can think of the above optimization problem as the following:

1. For every pair of nodes i and j with $i < j$, there is a possible edge (i,j) that can belong to the best tree $\widehat{T}$. If edge (i,j) is in $\widehat{T}$, then it contributes a "reward" of $\widehat{I}(X_i;X_j)$ to the sum that we are maximizing over.
2. Thus, the question is: What is the tree composed of edges that have the highest overall reward?

It turns out that this problem has been widely studied in computer science and amounts to finding what is called the "minimum weight spanning tree" (often shortened to just minimum spanning tree or MST). Note that we can easily turn the problem into one of getting the tree with the minimum overall weight (instead of maximum overall empirical mutual information) by associating the weight $w\big((i,j)\big) \triangleq -\widehat{I}(X_i;X_j)$ for each edge (i,j), and then solving the equivalent optimization problem

$$\widehat{T} = \arg\min_{T \text{ that is a tree over nodes } \{1,\ldots,k\}} \sum_{(i,j)\in T} w\big((i,j)\big).$$

The solution $\widehat{T}$ is called a minimum weight spanning tree because it is a tree that reaches (and thus "spans") all the possible nodes (so from any node, we can reach any other node by traversing edges along the tree), and if we sum up the weights associated with all the edges in the tree, we get the minimum possible overall weight.

Thus, the correctness of the Chow-Liu algorithm hinges on the correctness of established algorithms for solving the MST problem. One of the most popular algorithms for solving the MST problem is Kruskal's algorithm, which is what we covered in the Chow-Liu video:

- Start with no edges and repeatedly add the next lowest weight edge that doesn't introduce a cycle. (Again, lower weight means higher empirical mutual information here.)

What remains is to show why the Chow-Liu algorithm is correct (which given our analysis thus far amounts to just showing why Kruskal's algorithm is correct) and how long it takes to run. After all, anyone can write an algorithm that produces a wrong answer in $\mathcal{O}(1)$ time, and moreover, even if the algorithm is correct, if it has to loop through all K^{k-2} possible trees, then the algorithm would be practically useless for large values of k!

Bibliographic note. In what follows, we are largely following the exposition of Kruskal's algorithm in Dasgupta, Papadimitriou, and Vazirani's textbook "Algorithms" McGraw-Hill 2006 (Section 5.1).

3.5.3 Correctness of Kruskal's Algorithm

As we have already established previously in the course, a tree on k nodes has exactly $k-1$ edges. The intuition for the proof of this is that starting from a fully disconnected graph, by adding edges, every time we are merging exactly two connected components (a connected component is a collection of nodes that are all reachable from each other and that has no edges to any other nodes outside of the connected component). By starting with k connected components, after merging two connected components exactly $k-1$ times, we are left with a single connected component, and adding any other edge would result in a cycle (since all the nodes are already in the same connected component). Note that this way in which a tree is built up is precisely how Kruskal's algorithm builds up a tree: each time an edge is added, exactly two different connected components are merged (it cannot be that the edge added is between two nodes in the same connected component since that would mean that adding it would produce a cycle

as the nodes are already connected via some other path). After $k-1$ edges are added, all the nodes are in the same connected component, meaning that the edges added so far correspond to a graph that is connected (i.e., any node can reach any other node by traversing edges in the graph) and that also covers every node.

In fact, a key property of trees is that any graph over k nodes that is both connected and has $k-1$ edges must be a tree. The reason why this is true is to note that if we take a graph over k nodes that is connected and has $k-1$ edges, then if there were cycles, we could remove any cycles to end up with a tree that is still connected over the k nodes. However, we know that any tree over k nodes has $k-1$ edges, so it must mean that we didn't remove any edges, i.e., the graph was indeed a tree to begin with. So we know that the resulting graph built by Kruskal's algorithm is a tree that spans all k nodes.

The main technical challenge is justifying that the spanning tree produced by Kruskal's algorithm is of minimum weight. To show this, the high level idea is to show that every time Kruskal's algorithm adds an edge, the resulting graph (which is not yet a tree until we add $k-1$ edges) is a subset of a possible MST. (Note: In general, there could be multiple MST's for the same set of nodes and choice of weights on the possible edges.) Thus, after adding $k-1$ edges, the resulting graph is still part of an MST over k nodes, but this means that the graph at this point must itself be an MST since the graph is already a spanning tree over k nodes.

To show this result, we do a graph induction. Let E_ℓ be the set of the first ℓ edges added by Kruskal's algorithm. We aim to prove the proposition that for each $\ell = 0, 1, \ldots, k-1$, there exists an MST (represented as a set of edges) that contains E_ℓ as a subset of edges.

Base case ($\ell = 0$): At this point $E_0 = \varnothing$, and clearly any MST has the empty set as a subset of the edges.

Inductive step: Suppose the proposition above holds for $\ell = m$ for $0 \leqslant m < k-1$. We want to show that it also holds for $\ell = m+1$. Let T^* be the MST that contains E_m. Let e be the edge that Kruskal's algorithm adds, so that $E_{m+1} = E_m \cup \{e\}$. There are two cases to consider.

- If $E_{m+1} \subseteq T^*$, then we're done: E_{m+1} is also contained within an MST, namely T^*.
- If instead $E_{m+1} \nsubseteq T^*$, then now we need to show that E_{m+1} is instead contained in a different MST. Since $E_m \subseteq T^*$, then it must be that the edge added e is not in T^*. Thus, $T^* \cup \{e\}$ contains a cycle C which involves the edge e. This cycle C must contain another edge e' that is not in E_m. (Otherwise, it means that all the other edges in $C \backslash \{e\}$ are in E_m in which case Kruskal's algorithm could not have added edge e as it would have created the cycle C in producing E_{m+1}.) Then consider the set of edges $T^\dagger \triangleq T^* \cup \{e\} \backslash \{e'\}$, i.e., we take the MST T^* that contained E_m, add edge e, and remove edge e'. Note that
$$E_{m+1} \subseteq T^\dagger.$$

This holds since $E_m \subseteq T^*$, edge e' is neither in E_m nor E_{m+1}, and we added edge e to get from Em to E_{m+1}.

The final claim here is that the set of edges $T^\dagger$ actually corresponds to an MST. First off, note that $T^\dagger$ is connected and spans all the nodes. The reason why this is the case: We produce $T^\dagger$ by taking the MST T^*, adding edge e (which produces cycle C), and then removing edge e' (which is in cycle C). Note that for any graph that has a cycle, removing an edge from that cycle cannot disconnect the graph. In particular, all k of the nodes remain connected in $T^\dagger$ (i.e., there is a path to reach from any node to any other node), and there are $k-1$ edges, so $T^\dagger$ is again a tree that spans all k nodes.

As to why $T^\dagger$ has minimum weight, the key observation is that edges e and e' are both edges that are not in E_m, and so since Kruskal's algorithm added edge e to go from E_m to E_{m+1}, it must be that the weight $w(e')$ associated with e' is at least equal to the weight $w(e)$ of e. (If $w(e')$ had actually been less than $w(e)$, then since $E_m \subseteq T^*$ and $e' \in T^*$, then Kruskal's algorithm would have added edge e' instead since it certainly wouldn't have introduced a cycle.) We conclude that

$$\text{weight of } T^\dagger = \text{weight of } T^* - \underbrace{w(e')}_{\geqslant w(e)} + w(e) \leqslant \text{weight of } T^*,$$

so the weight of $T^\dagger$ is upper-bounded by the weight of T^*. But since T^* is an MST, then it must be that the weight of $T^\dagger$ cannot actually be lower so it too is an MST.

This completes the proof of the inductive step for why the edges E_{m+1} are also contained in an MST.

We thus conclude that for each $\ell = 0, 1, \ldots, k-1$, the intermediate edge set E_ℓ is contained in an MST. Then since E_{k-1} corresponds to a connected graph over nodes $\{1, \ldots, k\}$ with $k-1$ edges, it is a spanning tree, and as it is contained in an MST, it must itself be an MST since an MST over a graph of k nodes has exactly $k-1$ edges.

3.5.4 Running Time of the Chow-Liu Algorithm

The running time of Chow-Liu is the running time of computing all the empirical mutual information quantities followed by the running time of Kruskal's algorithm, where the tricky part in analyzing the running time of Kruskal's algorithm is looking at how long it takes to detect cycles.

Running time of computing empirical mutual information quantities. Let's first look at the time complexity of computing $\widehat{I}(X_i; X_j)$ for all $i < j$. We compute the empirical marginal distributions

$$\widehat{p}_{X_i}(a) = \tfrac{1}{n} \textstyle\sum_{\ell=1}^{n} \mathbf{1}\{x_i^{(\ell)} = a\},$$

for every $i \in \{1, \ldots, k\}$ and every item in the alphabet; let's suppose that the alphabet size for all the X_i's is $|\mathscr{X}|$. Then the running time to do this is $\mathscr{O}(k|\mathscr{X}|n)$.

Next we compute all the empirical pairwise distributions

$$\widehat{p}_{X_i,X_j}(a,b) = \tfrac{1}{n} \textstyle\sum_{\ell=1}^{n} \mathbf{1}\{x_i^{(\ell)} = a, x_j^{(\ell)} = b\}$$

for every $i < j$ and every pair of items in the alphabets. The running time to do this is $\mathscr{O}(k^2|\mathscr{X}|^2 n)$, where we recall that there are $\binom{k}{2} = \mathscr{O}(k^2)$ possible pairs.

Finally, we compute all $\binom{k}{2}$ empirical mutual information quantities

$$\widehat{I}(X_i; X_j) = \textstyle\sum_a \sum_b \widehat{p}_{X_i,X_j}(a,b) \log \frac{\widehat{p}_{X_i,X_j}(a,b)}{\widehat{p}_{X_i}(a)\widehat{p}_{X_j}(b)}.$$

Given that we have already computed the empirical distributions, this takes time $\mathscr{O}(k^2|\mathscr{X}|^2)$.

Thus, all together, computing the empirical distributions and the empirical mutual information quantities takes a total of $\mathscr{O}(k^2|\mathscr{X}|^2 n)$ operations (the complexity is dominated by computing the empirical pairwise distributions).

Running time of Kruskal's algorithm. Let's be clear about what the algorithm's steps are. We will use something called a Union-Find data structure to help us prevent cycles from being added. This data structure has three parts to it:

- To initialize the Union-Find data structure, every node is set to be its own connected component in the graph. Initializing every node to be its own connected component corresponds to when all the nodes are disconnected. This initialization takes time $\mathscr{O}(k)$.
- Given a node i, there is a function $find(i)$ that finds which connected component i belongs to. Each call to find takes time $\mathscr{O}(\log k)$.
- Given two nodes i and j, there is a function $union(i, j)$ that merges the connected components of i and j into a single connected component. Each call to union takes time $\mathscr{O}(\log k)$.

Details on the Union-Find data structure are given later.

We can now state Kruskal's algorithm as follows:

1. Set the edges added to the tree so far to be $T = \varnothing$. (Takes $\mathscr{O}(1)$ time.)

2. Initialize the Union-Find data structure, setting every node to be its own connected component. (Takes $\mathcal{O}(k)$ time.)

3. Sort all the possible edges (i.e., all $\binom{k}{2}$ pairs) by their weights. (Using quicksort or mergesort, this takes $\mathcal{O}(k^2 \log k^2) = \mathcal{O}(2k^2 \log k) = \mathcal{O}(k^2 \log k)$ time.)

4. For each pair of nodes (i, j) with $i < j$ and in increasing order of weight: (Worst case: $\binom{k}{2}$ total iterations)

   - If $find(i) \neq find(j)$, meaning that i and j are in different connected components: (this check takes $\mathcal{O}(\log k)$ time)
     (a) Set $T = T \cup \{(i, j)\}$. (Takes $\mathcal{O}(1)$ time.)
     (b) Call $union(i, j)$ to merge the connected components of i and j. (Takes $\mathcal{O}(\log k)$ time.)

Thus, putting together the pieces, the worst-case running time for Kruskal's algorithm is

$$\underbrace{\mathcal{O}(1) + \mathcal{O}(k)}_{\text{initialization}} + \underbrace{\mathcal{O}(k^2 \log k)}_{\text{sorting}} + \underbrace{\binom{k}{2}\left(\mathcal{O}(\log k) + \mathcal{O}(1) + \mathcal{O}(\log k)\right)}_{\text{adding edges}} = \mathcal{O}(k^2 \log k).$$

Recall that computing all the empirical mutual information quantities takes time $\mathcal{O}(k^2 |\mathcal{X}|^2 n)$, where $|\mathcal{X}|$ is the alphabet size of each variable (or you can treat this as the maximum alphabet size of any variable), and n is the number of training data. Thus, the overall time complexity of the Chow-Liu algorithm is

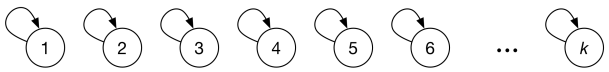
$$\mathcal{O}(k^2 |\mathcal{X}|^2 n + k^2 \log k).$$

This running time is certainly a massive improvement over trying to iterate through all k^{k-2} possible trees!

3.5.5 Details on the Union-Find Data Structure

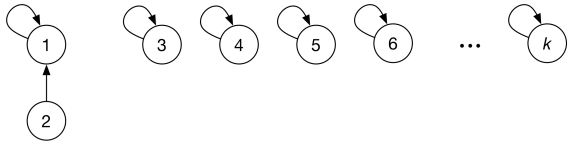
We first sketch out with an example how the Union-Find data structure works. The basic idea of the Union-Find data structure is to represent all the connected components in a *directed* graph over the k nodes. A directed graph, unlike an undirected graph, has orientation associated with each edge. A connected component is encoded as all nodes where if you keep following the arrows, you reach the same node that is its own parent. We'll illustrate what this means shortly.

Initialization involves setting each node to be its own parent:

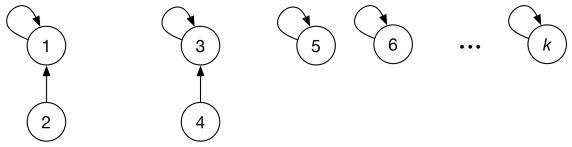


Here, notice that since each node is its own parent, it means that there are k different connected components.

Next, consider what happens when we add an edge, say $(1,2)$. This means that nodes 1 and 2 should now be in the same connected component, which we get by calling $union(1,2)$ with the Union-Find data structure, and what happens is that we're going to change the parent of one of the nodes 1 or 2 to point to the other node. For instance, we change the parent of node 2 to be node 1:



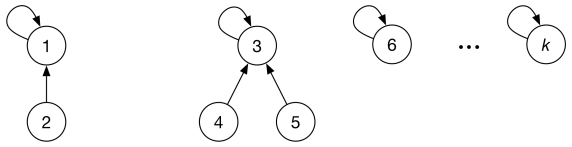
As another example, if we do $union(3,4)$, we get:



Thus, now nodes 1 and 2 belong to the same connected component since for either of the nodes, if we keep following the directed edges (the arrows), we end up at the same node 1 that is its own parent. Similarly, nodes 3 and 4 belong to the same connected component. In particular, this illustrates how the $find$ function works: $find(i)$ simply goes to node i and keeps following the directed arrows until it hits a node that is its own parent: nodes that are their own parents are the unique representatives of each connected component. If you want, you can think of each connected component as a group where there is a leader of the group, which is the node that is its own parent, and $find(i)$ returns whoever this leader node is. Thus, at this point $find(1) = 1$, $find(2) = 1$, $find(3) = 3$, $find(4) = 3$, etc.

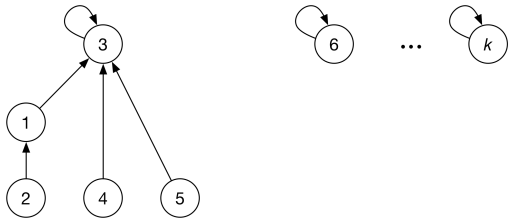
Importantly, what's going to cause $find$ to be slow is if we have to follow a lot of arrows. To prevent the graphs formed from in some sense becoming "unbalanced", the *union* function needs to be careful with how it combines connected components.

Consider if we want to do $union(3,5)$. A bad idea would be to have node 5's parent be changed to be node 4. This makes it so that $find(5)$ would have to move along 2 edges before reaching 3. A better idea is to have node 5's parent be changed to node 3:

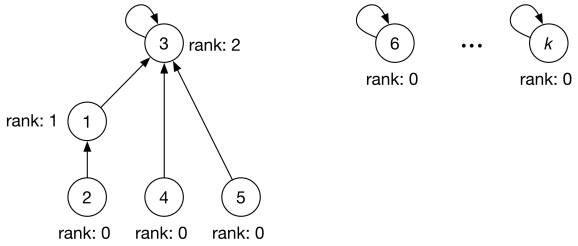


In particular, the basic idea is that we take the shorter tree and set its root to be the root of the taller tree.

Of course, some times the heights are the same, so we're forced to just make a taller tree. For example, $union(2,5)$ yields the following:



To help with figuring out which tree is higher, for every node, the Union-Find data structure stores a variable *rank* that says how tall that node is. For example, below we have written out all the ranks:



We are now ready to provide detail for the three operations of the Union-Find data structure.

Initialization.

For each node $i \in \{1,\ldots,k\}$:

Set $\pi(i) = i$. (Set the parent of node i to be i.)

Set $rank(i) = 0$. (Set the height of node i to be 0.)

find(i)**:**

While $i \neq \pi(i)$: (While we have not reached the node that is its own parent:)

Set $i = \pi(i)$. (Follow the directed edge upward.)

Return i. (Return the "leader" of the connected component which is the node that is its own parent.)

union(i, j)**:**

Set $r_i = find(i)$. (Find the root of the connected component of i.)

Set $r_j = find(j)$. (Find the root of the connected component of j.)

If $r_i = r_j$: (If the connected components are the same:)

Return. (There's nothing to do since i and j are already in the same connected component.)

Else:

If $rank(r_i) > rank(r_j)$: (If the tree rooted at r_i is taller than the tree rooted at r_j:)

Set $\pi(r_j) = r_i$. (Change the parent of r_j to be the root r_i of the taller tree.)

Else:

Set $\pi(r_i) = r_j$. (Change the parent of r_i to be the root r_j of the same height or taller tree.

If $rank(r_i) = rank(r_j)$: (If they were actually the same height:)

Set $rank(r_j) = rank(r_j) + 1$. (Change the height of the now taller root node to be 1 taller.)

Running time for the three pieces: It should be clear why initialization takes time $\mathcal{O}(k)$ i.e., linear in the number of nodes. A more careful argument is needed to justify why the rank of any node is at most $\log_2 k$, which would immediately imply that each $find$ call takes $\mathcal{O}(\log k)$ time (since the rank is how high a node in a tree is and the running time of $find$ just scales with the maximum height of a tree, which is the maximum rank). Meanwhile, each call to *union* involves calling $find$ twice followed by a series of operations that takes time $\mathcal{O}(1)$, so the overall running time for each union call is the same as that of find, namely $\mathcal{O}(\log k)$.

The question that remains is why the rank of any node is at most $\log_2 k$. To answer this, we make a series of observations that build on each other:

- As we move upward along any directed edge, the rank strictly increases. In other words, $rank(i) < rank(\pi(i))$.
- The only time we ever increment a rank of a root node is when we just did a union of two trees that have the same height (for which for exactly one of the original root nodes, we increase its rank by 1). This means that the very first time we do union (for example, when we start from everything disconnected and we do our very first *union*), the smallest possible two trees we are merging each has size 1, and the resulting tree ends up having rank 1 and 2 nodes. Thus, we need at least 2 of these rank 1 trees with at least 2 nodes each to form a bigger tree with rank 2, at which point we at least have 4 nodes. This pattern continues: The (sub)tree rooted at a node with rank ℓ must have at least 2^ℓ nodes in it.
- How large could the largest rank possibly be? Well, since a rank ℓ node has at least 2^ℓ nodes in it, consider when $\ell = \log_2 k$, which means that this tree as at least k nodes in it, but this is indeed the total number of nodes! Thus, we cannot have a higher rank than $\ell = \log_2 k$; otherwise, we would need more than k nodes.

This completes our analysis of the Union-Find data structure.

Path compression. We end by stating a remarkable result. It turns out that there's a small modification to find that cuts down the running time for each call to find to take *average* running time $\mathcal{O}(\log_2^* k)$ instead of $\mathcal{O}(\log k)$, where $\log_2^*$ (pronounced log star) is the iterated logarithm, i.e., how many times you iteratively apply log base 2 until you

get a number at most 1. Note that $\log_2^*$ grows extremely slowly: $\log_2^*(2^{65536}) = 5$, and 2^{65536} is many orders of magnitude larger than the number of atoms in the observable universe ($\approx 10^{80}$). As such, for practical purposes, $\mathcal{O}(\log_2^* k)$ might as well be constant time.

The trick is that when we call $find(i)$, for every node we encounter as we work our way to the "leader"/root node, we change all the parents of nodes we encounter to directly point to the "leader"/root node. This makes it so that *rank* no longer corresponds to tree height, but it turns out to not affect correctness. Now the tree height gets compressed quite substantially. This modification is called "path compression".

The only change is in the $find$ procedure:

find(i):

If $i \neq \pi(i)$: (If node i is not a "leader"/root node:)

Set $\pi(i) = find(\pi(i))$. (Change the parent of i to be the "leader"/root node.)

Return $\pi(i)$. (Return the "leader"/root node.)

We can readily unravel the recursion: if i is a "leader"/root node, we skip the if statement and just return the node itself since it is its own parent. Otherwise, we recursively work our way up the directed tree changing parents of the nodes we encounter to all point to the "leader"/root node.

The running time analysis here requires looking at an "amortized" or averaged case over a sequence of $find$ calls, which we will omit details for.

The bottom line is that the average case running time of Kruskal's algorithm with this path compression modification makes it so that the $find$ and *union* calls (again each *union* call does two $find$ calls, which dominates its running time) each takes average time $\mathcal{O}(\log_2^* k)$, yielding an overall average running time of

$$\underbrace{\mathcal{O}(1) + \mathcal{O}(k)}_{\text{initialization}} + \underbrace{\mathcal{O}(k^2 \log k)}_{\text{sorting}} + \underbrace{\binom{k}{2}\left(\mathcal{O}(\log_2^* k) + \mathcal{O}(1) + \mathcal{O}(\log_2^* k)\right)}_{\text{adding edges}} = \mathcal{O}(k^2 \log k + k^2 \log_2^* k).$$

The sorting operation still dominates the running time, but if the edge weights can be converted to limited precision unsigned integers without sacrificing accuracy (i.e., by converting floating point to integers, so long as the different edges still correspond to different integers and no two edge weights get quantized to be the same integer when they were different weights before), then a linear-time radix sort can be used, bringing the average running time of this modified Kruskal's algorithm to

$$\mathcal{O}(pk^2 + k^2 \log_2^* k),$$

where p is an integer precision parameter, which you can think of as scaling with the maximum number of bits needed to store the integers used to represent the edge weights. With this modification, the overall Chow-Liu algorithm with path compression and radix sort has running time

$$\mathcal{O}(k^2 |\mathcal{X}|^2 n + pk^2 + k^2 \log_2^* k),$$

cut down from the original Chow-Liu running time of

$$\mathcal{O}(k^2 |\mathcal{X}|^2 n + k^2 \log k),$$

where k is the number of random variables in the graph, $|\mathcal{X}|$ is the (maximum) alphabet size of any of the random variables, and n is the number of training data points, Again, we need the additional assumption on the precision of the edge weights being sufficiently limited to enable radix sort.

Chapter 4

Epilogue

4.1 What's Next?

In 6.008.1x, we only scratched the surface of what's out there for probability and inference. We took a computational approach, where almost every topic we encountered had Python code to go with it, where initially we supplied much of the code and then by parts 2 and 3 of the course, you were providing most of the code in the form of projects. We also sketched out how to derive the different algorithms we encountered (thus providing some justification for why they are correct), and how to analyze their running times. Of course, we also saw how these algorithms or their variants can be used to solve a variety of real-world problems (well ... and problems involving wizards, aliens, and other standbys).

There is an enormous amount of material that builds off what we've covered in 6.008.1x. Below are just a few examples. This listing is hardly exhaustive!

Part 1: Probability and Inference. We mainly focused on finite probability spaces and finite random variables and briefly mentioned discrete probability spaces and discrete random variables. Of course, there are also continuous random variables (such as the normal/Gaussian distribution informally called the "bell curve", where the sample space is the entire real number line), and more general random variables (for example, a random variable where with some probability it takes on the value of a discrete random variable and with some probability it takes on the value of a continuous random variable).

As for inference, we only provided a cursory introduction with the help of Bayes' theorem, skirting discussion on the two major schools of thought of frequentist and Bayesian inference, where the former views the quantity we are inferring as fixed but unknown, and the latter views the quantity we are inferring as random. These two schools of thought lead to very different interpretations of inference procedures!

Part 2: Inference in Graphical Models. Undirected graphical models can have loops, which introduces a variety of complications that in turn can be addressed by a variety of algorithms. Here, exact inference algorithms can be extremely computationally expensive, demanding that we settle for approximate inference algorithms, such as algorithms based on random sampling. A famous example of this is Markov chain Monte Carlo.

Also, we can have graphical models for continuous random variables. When the random variables are all Gaussian (and specifically what are called jointly-distributed Gaussian random variables), the Sum-Product algorithm can be written in a simple way to produce an algorithm called "Gaussian belief propagation", which specialized to Gaussian hidden Markov models leads to the Kalman filter (which just does the forward pass of the forward-backward algorithm).

Part 3: Learning Probabilistic Models. There are many complications to learning probabilistic models. For example, we have always assumed that every training data point has an observed value for every random variable in the graphical model of interest. What if certain random variables are never observed? For example, for HMM's, the latent variables are generally never observed, not even in the training data. The algorithm we presented for

learning parameters of a graphical model would not be able to handle this case. What we can do is "fill-in" missing values. A general algorithm for doing this procedure is the expectation-maximization (EM) algorithm, which finds approximate maximum likelihood estimates when certain random variables aren't observed. Of course, there are many other possible complications, such as different variables being missing across different training data points, the training data points not being i.i.d., or even us having control over which training data to collect (i.e., we get to query for what the next training data point should be that someone then manually labels, and we repeatedly do this to build up a training dataset and to learn a model).

Separately, there is a question of what we can say about guarantees on learning procedures. For example, how much training data should we collect before we can achieve some tolerated probability of error? One of the major trends in analysis of inference algorithms is to trade off running time, quality of inference (for example, in a classification task, the quality of inference could be measured by misclassification rate), and amount of training data. For example, in a classification task, if we don't have much training data, maybe we are okay with having the running time of an inference procedure be very large to achieve a small misclassification rate. There are a variety of tools for providing theoretical performance guarantees on inference algorithms, much of which draws on ideas from information theory. A subset of the results here are referred to as "asymptotic" guarantees on inference procedures in that certain quantities (such as the number of training data) are taken to go to infinity.

Courses. At MIT, the residential version of 6.008 covers not only the material in 6.008.1x but also basics of asymptotic analysis of inference procedures, random sampling for approximate inference, and an introduction to inference with continuous random variables (which is largely a re-hash of everything else in the class now with a slight twist as we use continuous random variables instead of discrete random variables).

A few other courses at MIT that build on 6.008 and that are available online:

- 6.041 Probabilistic Systems Analysis (available on edX and OCW): a much more thorough introduction to part 1 of 6.008.1x although without the coding
- 6.436 Fundamentals of Probability (available on OCW): a graduate-level measure-theoretic introduction to probability that more rigorously/formally explains why probability theory works; less emphasis than 6.041 on solving engineering/word problems and more emphasis on proofs/theoretical analysis
- 6.438 Algorithms for Inference (available on OCW): a graduate-level introduction to probabilistic graphical models (heavily builds on parts 2 and 3 of 6.008.1x)
- 6.867 Machine Learning (available on OCW): a graduate-level introduction to machine learning (covers a variety of topics, only partly overlapping with 6.008.1x)
- 9.520 Statistical Learning Theory and Applications (available on OCW): a graduate-level introduction to statistical learning theory (largely about theoretical guarantees on learning procedures)

A few textbooks that may be of interest (again, not exhaustive):

- Dimitri P. Bertsekas and John N. Tsitsiklis. *Introduction to Probability*. 2nd edition. Athena Scientific 2008.
- Geoffrey R. Grimmett and David R. Stirzaker. *Probability and Random Processes*. 3rd edition. Oxford University Press 2001.
- Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer 2007.
- Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press 2009.
- David MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Pressa 2003. (Freely available off the official textbook website.)

- Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R*. Springer-Verlag 2013. (Freely available off the official textbook website.)
- Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag 2009. (Freely available off the official textbook website.)

Appendix A

Notation Summary

Typically we use a capital letter like X to denote a random variable, a script (or calligraphic) letter $\mathscr{X}$ to denote a set (or an event), and a lowercase letter like x to refer to a nonrandom variable. Occasionally we will also use capital letters to refer to a constant that is not varying throughout the problem (in contrast to using a lowercase letter like x that can be a "dummy" variable such as in a summation $\sum_x p_X(x)$, for which lowercase x refers to a specific constant value but we are varying what x is and it is effectively a temporary variable that we do not need after computing the summation).

p_X or $p_X(\cdot)$	probability table/probability mass function (PMF)/probability distribution/marginal distribution of random variable X
$p_X(x)$ or $\mathbb{P}(X = x)$	probability that random variable X takes on value x
$p_{X,Y}$ or $p_{X,Y}(\cdot,\cdot)$	joint probability table/joint PMF/joint probability distribution of random variables X and Y
$p_{X,Y}(x,y)$ or $\mathbb{P}(X = x, Y = y)$	probability that X takes on value x and Y takes on value y
$p_{X\mid Y}(\cdot \mid y)$	conditional probability table/conditional PMF/conditional probability distribution of X given Y takes on value y
$p_{X\mid Y}(x \mid y)$ or $\mathbb{P}(X = x \mid Y = y)$	probability that X takes on value x given that Y takes on value y
$X \sim p$ or $X \sim p(\cdot)$	X is distributed according to distribution p
$X \perp Y$	X and Y are independent
$X \perp Y \mid Z$	X and Y are independent given Z

We will also of course be dealing with many events or many random variables. For example, $\mathbb{P}(\mathscr{A}, \mathscr{B}, \mathscr{C} \mid \mathscr{D}, \mathscr{E})$ would be the probability that events $\mathscr{A}$, $\mathscr{B}$, and $\mathscr{C}$ all occur, given that both events $\mathscr{D}$ and $\mathscr{E}$ occur, which by the definition of conditional probability would be

$$\mathbb{P}(\mathscr{A}, \mathscr{B}, \mathscr{C} \mid \mathscr{D}, \mathscr{E}) = \frac{\mathbb{P}(\mathscr{A}, \mathscr{B}, \mathscr{C}, \mathscr{D}, \mathscr{E})}{\mathbb{P}(\mathscr{D}, \mathscr{E})}.$$

Similarly, $p_{X,Y,Z|V,W}$ would refer to a joint conditional distribution of random variables X, Y, and Z given both V and W taking on specific values together:

$$p_{X,Y,Z|V,W}(x,y,z \mid v,w) = \frac{p_{X,Y,Z,V,W}(x,y,z,v,w)}{p_{V,W}(v,w)}.$$

When we have a collection of random variables, e.g., W,X,Y,Z, if we say that they are independent (without specifying what type of independence), then what we mean is mutual independence, which means that the joint distribution factorizes into the marginal distributions:

$$p_{W,X,Y,Z}(w,x,y,z) = p_W(w)p_X(x)p_Y(y)p_Z(z) \qquad \text{for all } w,x,y,z.$$

Appendix B

Supplementary Information

B.1 External Resources

"Missed Expections" chapted from MIT 6.042 course on OCW

B.2 Hints from the Discussion Forum

B.2.1 Law of Total Expectation as Matrices

Posted by Community TA Derek_edX, approximately October 5th, 2016.

In some sense the law of Total Expectation is just playing around with associativity. The below easily covers all finite cases. (If we take limits we could go beyond... but that's outside the scope.)

In any case, here is a different way to look at the law of total expectation using matrices.

– setup–

a matrix, A, has all payoffs of X, weighted by their respective probabilities. For simplicity, let A be a square matrix. (Note that if A is not 'naturally' square, we can always add more zeros until it is square)

$$h = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Thus if we sum up all entries in A, we get the expected value of X. That is : $E[X] = h^T A h$

v= vector with $E[X \mid \mathscr{B}_i]$ in the ith spot

p= a vector with $P(\mathscr{B}_i)$ in the ith spot

Much as we partition a sample space, all we need to do now is partition A so that its columns correspond to events $\mathscr{B}_0, \ldots, \mathscr{B}_{n-1}$ (For the avoidance of doubt, A should be set up in advance so that each entry in column j corresponds to a probability times payoff associated with B_j, or else such entry is an 'artificially' added zero.)

$$A = \left[\begin{array}{c|c|c|c} A_0 & A_1 & \cdots & A_{n-1} \end{array} \right]$$

$$Y = \left[\begin{array}{c|c|c|c} \frac{1}{p(B_0)} A_0 & \frac{1}{p(B_1)} A_1 & \cdots & \frac{1}{p(B_{n-1})} A_{n-1} \end{array} \right]$$

$$D = \begin{bmatrix} p(B_0) & 0 & 0 & 0 \\ 0 & p(B_1) & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & p(B_{n-1}) \end{bmatrix}$$

Thus: $A = YD$

–main argument–

By definition: $E[X] = h^T Ah$

$E[X] = h^T \big(YD\big)h$

by associativity: $E[X] = \big(h^T Y\big)\big(Dh\big)$

$E[X] = \big(v\big)^T \big(p\big) = v^T p$

note $v^T p = \sum_{i=0}^{n-1} \mathbb{E}[X \mid \mathscr{B}_i]\mathbb{P}(\mathscr{B}_i)$