CHAPTER 13

# Hidden Markov Models

There are many situations where one must work with sequences. Here is a simple, and classical, example. We see a sequence of words, but the last word is missing. I will use the sequence "I had a glass of red wine with my grilled xxxx". What is the best guess for the missing word? You could obtain one possible answer by counting word frequencies, then replacing the missing word with the most common word. This is "the", which is not a particularly good guess because it doesn't fit with the previous word. Instead, you could find the most common pair of words matching "grilled xxxx", and then choose the second word. If you do this experiment (I used Google Ngram viewer, and searched for "grilled *"), you will find mostly quite sensible suggestions (I got "meats", "meat", "fish", "chicken", in that order). If you want to produce random sequences of words, the next word should depend on some of the words you have already produced. A model with this property that is very easy to handle is a Markov chain (defined below).

It is really common to see a noisy sequence, and want to recover the noise free version. You should think of this recipe in a very broad way. So, for example, the recipe applies when one hears sound and would like to turn it into text. Here, the sound is the noisy sequence, and the text is the noise-free version. You might see handwriting (noisy sequence) and want to recover text (noise free version). You might see video of a person moving (noisy sequence) and want to recover joint angles (noise free version). The standard model for this recipe is a hidden markov model. We assume the noise free sequence is generated by a known Markov model, and the procedure that produced observed items from the sequence is known. In this case, a straightforward inference algorithm yields the noise free sequence from the observations. Furthermore, a hidden Markov model can be learned from examples using EM.

## 13.1 MARKOV CHAINS

A sequence of random variables $X_n$ is a **Markov chain** if it has the property that,

$$P(X_n = j|\text{values of all previous states}) = P(X_n = j|X_{n-1}),$$

or, equivalently, only the last state matters in determining the probability of the current state. The probabilities $P(X_n = j|X_{n-1} = i)$ are the **transition probabilities**. We will always deal with discrete random variables here, and we will assume that there is a finite number of states. For all our Markov chains, we will assume that

$$P(X_n = j|X_{n-1} = i) = P(X_{n-1} = j|X_{n-2} = i).$$

Formally, we focus on *discrete time, time homogenous Markov chains in a finite state space*. With enough technical machinery one can construct many other kinds of Markov chain.

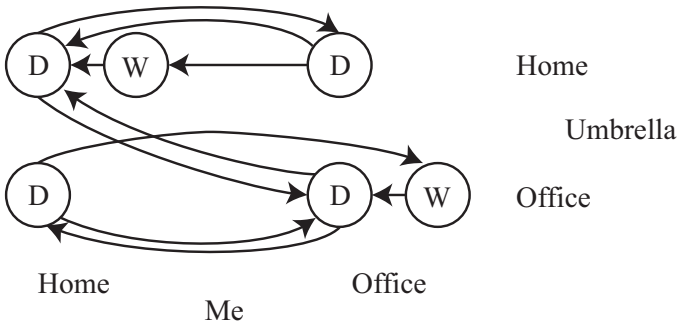


FIGURE 13.1: *A directed graph representing the umbrella example. Notice you can't arrive at the office wet with the umbrella at home (you'd have taken it), and so on. Labelling the edges with probabilities is left to the reader.*

One natural way to build Markov chains is to take a finite directed graph and label each directed edge from node $i$ to node $j$ with a probability. We interpret these probabilities as $P(X_n = j|X_{n-1} = i)$ (so the sum of probabilities over *outgoing* edges at any node must be 1). The Markov chain is then a **biased random walk** on this graph. A bug (or any other small object you prefer) sits on one of the graph's nodes. At each time step, the bug chooses one of the outgoing edges at random. The probability of choosing an edge is given by the probabilities on the drawing of the graph (equivalently, the transition probabilities). The bug then follows that edge. The bug keeps doing this until it hits an end state.

**Worked example 13.1** *Umbrellas*

I own one umbrella, and I walk from home to the office each morning, and back each evening. If it is raining (which occurs with probability $p$, and my umbrella is with me), I take it; if it is not raining, I leave the umbrella where it is. We exclude the possibility that it starts raining while I walk. Where I am, and whether I am wet or dry, forms a Markov chain. Draw a state machine for this Markov chain.

**Solution:** Figure 13.1 gives this chain. A more interesting question is with what probability I arrive at my destination wet? Again, we will solve this with simulation.

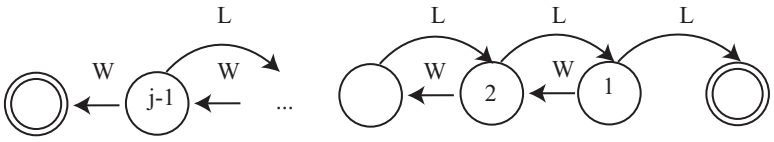


FIGURE 13.2: *A directed graph representing the gambler's ruin example. I have labelled each state with the amount of money the gambler has at that state. There are two end states, where the gambler has zero (is ruined), or has $j$ and decides to leave the table. The problem we discuss is to compute the probability of being ruined, given the start state is $s$. This means that any state except the end states could be a start state. I have labelled the state transitions with "W" (for win) and "L" for lose, but have omitted the probabilities.*

**Worked example 13.2** *The gambler's ruin*

Assume you bet 1 a tossed coin will come up heads. If you win, you get 1 and your original stake back. If you lose, you lose your stake. But this coin has the property that $P(H) = p < 1/2$. You have $s$ when you start. You will keep betting until either (a) you have 0 (you are ruined; you can't borrow money) or (b) the amount of money you have accumulated is $j$, where $j > s$. The coin tosses are independent. The amount of money you have is a Markov chain. Draw the underlying state machine. Write $P(\text{ruined, starting with s}|p) = p_s$. It is straightforward that $p_0 = 1$, $p_j = 0$. Show that

$$p_s = pp_{s+1} + (1-p)p_{s-1}.$$

**Solution:** Figure 13.2 illustrates this example. The recurrence relation follows because the coin tosses are independent. If you win the first bet, you have $s+1$ and if you lose, you have $s-1$.

Notice an important difference between examples 13.1 and 13.2. For the gambler's ruin, the sequence of random variables can end (and your intuition likely tells you it should do so reliably). We say the Markov chain has an **absorbing state** – a state that it can never leave. In the example of the umbrella, there is an infinite sequence of random variables, each depending on the last. Each state of this chain is **recurrent** – it will be seen repeatedly in this infinite sequence. One way to have a state that is not recurrent is to have a state with outgoing but no incoming edges.

The gambler's ruin example illustrates some points that are quite characteristic of Markov chains. You can often write recurrence relations for the probability of various events. Sometimes you can solve them in closed form, though we will not pursue this thought further. It is often very helpful to think creatively about what the random variable is (example 13.3).

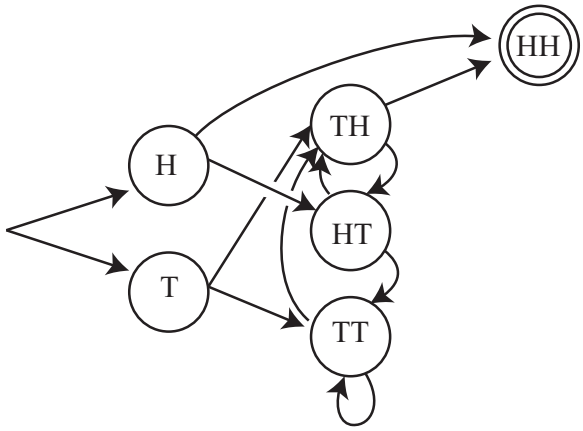


FIGURE 13.3: *A directed graph representing the coin flip example, using the pairs of random variables described in worked example 13.3. A sequence "HTHTHH" (where the last two H's are the last two flips) would be generated by transitioning to H, then to HT, then to TH, then to HT, then to TH, then to HH. By convention, the end state is a double circle. Each edge has probability 1/2.*

**Worked example 13.3** *Multiple Coin Flips*

You choose to flip a fair coin until you see two heads in a row, and then stop. Represent the resulting sequence of coin flips with a Markov chain. What is the probability that you flip the coin four times?

**Solution:** You could think of the chain as being a sequence of independent coin flips. This is a Markov chain, but it isn't very interesting, and it doesn't get us anywhere. A better way to think about this problem is to have the $X$'s be *pairs* of coin flips. The rule for changing state is that you flip a coin, then append the result to the state and drop the first item. Then you need a special state for stopping, and some machinery to get started. Figure 13.3 shows a drawing of the directed graph that represents the chain. The last three flips must have been $THH$ (otherwise you'd go on too long, or end too early). But, because the second flip must be a $T$, the first could be either $H$ or $T$. This means there are two sequences that work: $HTHH$ and $TTHH$. So $P(4 \text{ flips}) = 2/16 = 1/8$. We might want to answer significantly more interesting questions. For example, what is the probability that we must flip the coin more than 10 times? It is often possible to answer these questions by analysis, but we will use simulations.

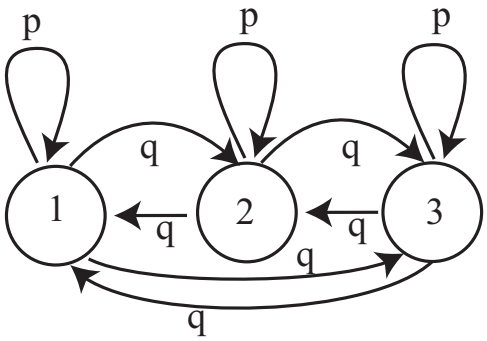


FIGURE 13.4: *A virus can exist in one of 3 strains. At the end of each year, the virus mutates. With probability $\alpha$, it chooses uniformly and at random from one of the 2 other strains, and turns into that; with probability $1-\alpha$, it stays in the strain it is in. For this figure, we have transition probabilities $p = (1-\alpha)$ and $q = (\alpha/2)$.*

**Useful Facts: 13.1** *Markov chains*

A Markov chain is a sequence of random variables $X_n$ with the property that:

$$P(X_n = j|\text{values of all previous states}) = P(X_n = j|X_{n-1}).$$

### 13.1.1 Transition Probability Matrices

Define the matrix $\mathcal{P}$ with $p_{ij} = P(X_n = j|X_{n-1} = i)$. Notice that this matrix has the properties that $p_{ij} \geq 0$ and

$$\sum_j p_{ij} = 1$$

because at the end of each time step the model must be in some state. Equivalently, the sum of transition probabilities for outgoing arrows is one. Non-negative matrices with this property are **stochastic matrices**. By the way, you should look very carefully at the $i$'s and $j$'s here — Markov chains are usually written in terms of *row* vectors, and this choice makes sense in that context.

**Worked example 13.4** *Viruses*

Write out the transition probability matrix for the virus of Figure 13.4, assuming that $\alpha = 0.2$.

**Solution:** We have $P(X_n = 1|X_{n-1} = 1) = (1 - \alpha) = 0.8$, and $P(X_n = 2|X_{n-1} = 1) = \alpha/2 = P(X_n = 3|X_{n-1} = 1)$; so we get

$$\begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}$$

Now imagine we do not know the initial state of the chain, but instead have a probability distribution. This gives $P(X_0 = i)$ for each state $i$. It is usual to take these $k$ probabilities and place them in a $k$-dimensional *row vector*, which is usually written $\pi$. From this information, we can compute the probability distribution over the states at time 1 by

$$\begin{aligned} P(X_1 = j) &= \sum_i P(X_1 = j, X_0 = i) \\ &= \sum_i P(X_1 = j|X_0 = i)P(X_0 = i) \\ &= \sum_i p_{ij}\pi_i. \end{aligned}$$

If we write $\mathbf{p}^{(n)}$ for the row vector representing the probability distribution of the state at step $n$, we can write this expression as

$$\mathbf{p}^{(1)} = \pi\mathcal{P}.$$

Now notice that

$$\begin{aligned} P(X_2 = j) &= \sum_i P(X_2 = j, X_1 = i) \\ &= \sum_i P(X_2 = j|X_1 = i)P(X_1 = i) \\ &= \sum_i p_{ij} \left( \sum_{ki} p_{ki}\pi_k \right). \end{aligned}$$

so that

$$\mathbf{p}^{(n)} = \pi\mathcal{P}^n.$$

This expression is useful for simulation, and also allows us to deduce a variety of interesting properties of Markov chains.

**Useful Facts: 13.2** *Transition probability matrices*

A finite state Markov chain can be represented with a matrix $\mathcal{P}$ of transition probabilities, where the $i$, $j$'th element $p_{ij} = P(X_n = j | X_{n-1} = i)$. This matrix is a stochastic matrix. If the probability distribution of state $X_{n-1}$ is represented by $\pi_{n-1}$, then the probability distribution of state $X_n$ is given by $\pi_{n-1}^T \mathcal{P}$.

### 13.1.2 Stationary Distributions

**Worked example 13.5** *Viruses*

We know that the virus of Figure 13.4 started in strain 1. After two state transitions, what is the distribution of states when $\alpha = 0.2$? when $\alpha = 0.9$? What happens after 20 state transitions? If the virus starts in strain 2, what happens after 20 state transitions?

**Solution:** If the virus started in strain 1, then $\pi = [1, 0, 0]$. We must compute $\pi(\mathcal{P}(\alpha))^2$. This yields $[0.66, 0.17, 0.17]$ for the case $\alpha = 0.2$ and $[0.4150, 0.2925, 0.2925]$ for the case $\alpha = 0.9$. Notice that, because the virus with small $\alpha$ tends to stay in whatever state it is in, the distribution of states after two years is still quite peaked; when $\alpha$ is large, the distribution of states is quite uniform. After 20 transitions, we have $[0.3339, 0.3331, 0.3331]$ for the case $\alpha = 0.2$ and $[0.3333, 0.3333, 0.3333]$ for the case $\alpha = 0.9$; you will get similar numbers even if the virus starts in strain 2. After 20 transitions, the virus has largely "forgotten" what the initial state was.

In example 13.5, the distribution of virus strains after a long interval appears not to depend much on the initial strain. This property is true of many Markov chains. Assume that our chain has a finite number of states. Assume that any state can be reached from any other state, by some sequence of transitions. Such chains are called **irreducible**. Notice this means there is no absorbing state, and the chain cannot get "stuck" in a state or a collection of states. Then there is a unique vector $\mathbf{s}$, usually referred to as the **stationary distribution**, such that for *any* initial state distribution $\pi$,

$$\lim_{n \to \infty} \pi \mathcal{P}^{(n)} = \mathbf{s}.$$

Equivalently, if the chain has run through many steps, it no longer matters what the initial distribution is. The probability distribution over states will be $\mathbf{s}$.

The stationary distribution can often be found using the following property. Assume the distribution over states is $\mathbf{s}$, and the chain goes through one step. Then

the new distribution over states must be $\mathbf{s}$ too. This means that

$$\mathbf{s}\mathcal{P} = \mathbf{s}$$

so that $\mathbf{s}$ is an eigenvector of $\mathcal{P}^T$, with eigenvalue 1. It turns out that, for an irreducible chain, there is exactly one such eigenvector.

The stationary distribution is a useful idea in applications. It allows us to answer quite natural questions, without conditioning on the initial state of the chain. For example, in the umbrella case, we might wish to know the probability I arrive home wet. This could depend on where the chain starts (example 13.6). If you look at the figure, the Markov chain is irreducible, so there is a stationary distribution and (as long as I've been going back and forth long enough for the chain to "forget" where it started), the probability it is in a particular state doesn't depend on where it started. So the most sensible interpretation of this probability is the probability of a particular state in the stationary distribution.

**Worked example 13.6** *Umbrellas, but without a stationary distribution*

This is a different version of the umbrella problem, but with a crucial difference. When I move to town, I decide randomly to buy an umbrella with probability 0.5. I then go from office to home and back. If I have bought an umbrella, I behave as in example 13.1. If I have not, I just get wet. Illustrate this Markov chain with a state diagram.

**Solution:** Figure 13.5 does this. Notice this chain *isn't* irreducible. The state of the chain in the far future depends on where it started (i.e. did I buy an umbrella or not).

**Useful Facts: 13.3** *Many Markov chains have stationary distributions*

If a Markov chain has a finite set of states, and if it is possible to get from any state to any other state, then the chain will have a stationary distribution. A sample state of the chain taken after it has been running for a long time will be a sample from that stationary distribution. Once the chain has run for long enough, it will visit states with a frequency corresponding to that stationary distribution, though it may take many state transitions to move from state to state.

### 13.1.3 Example: Markov Chain Models of Text

Imagine we wish to model English text. The very simplest model would be to estimate individual letter frequencies (most likely, by counting letters in a large body of example text). We might count spaces and punctuation marks as letters.

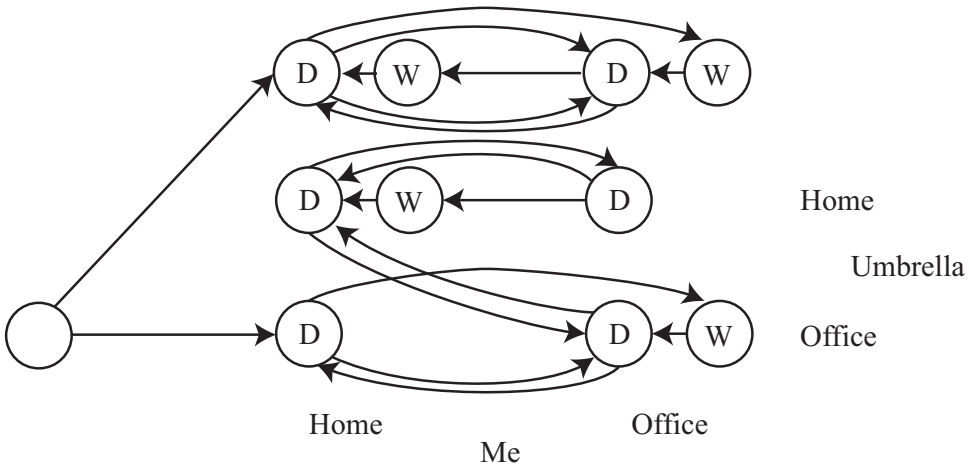


FIGURE 13.5: *In this umbrella example, there can't be a stationary distribution; what happens depends on the initial, random choice of buying/not buying an umbrella.*

We regard the frequencies as probabilities, then model a sequence by repeatedly drawing a letter from that probability model. You could even punctuate with this model by regarding punctuation signs as letters, too. We expect this model will produce sequences that are poor models of English text – there will be very long strings of "a"s, for example. This is clearly a (rather dull) Markov chain. It is sometimes referred to as a 0-th order chain or a 0-th order model, because each letter depends on the 0 letters behind it.

A slightly more sophisticated model would be to work with pairs of letters. Again, we would estimate the frequency of pairs by counting letter pairs in a body of text. We could then draw a first letter from the letter frequency table. Assume this is an "a". We would then draw the second letter by drawing a sample from the conditional probability of encountering each letter after "a", which we could compute from the table of pair frequencies. Assume this is an "n". We get the third letter by drawing a sample from the conditional probability of encountering each letter after "n", which we could compute from the table of pair frequencies, and so on. This is a first order chain (because each letter depends on the one letter behind it).

Second and higher order chains (or models) follow the general recipe, but the probability of a letter depends on more of the letters behind it. You may be concerned that conditioning a letter on the two (or $k$) previous letters means we don't have a Markov chain, because I said that the $n$'th state depends on only the $n-1$'th state. The cure for this concern is to use states that represent two (or $k$) letters, and adjust transition probabilities so that the states are consistent. So for a second order chain, the string "abcde" is a sequence of four states, "ab", "bc", "cd", and "de".

**Worked example 13.7** *Modelling short words*

Obtain a text resource, and use a trigram letter model to produce four letter words. What fraction of bigrams (resp. trigrams) do not occur in this resource? What fraction of the words you produce are actual words?

**Solution:** I used the text of a draft of this chapter. I ignored punctuation marks, and forced capital letters to lower case letters. I found 0.44 of the bigrams and 0.90 of the trigrams were not present. I built two models. In one, I just used counts to form the probability distributions (so there were many zero probabilities). In the other, I split a probability of 0.1 between all the cases that had not been observed. A list of 20 word samples from the first model is: "ngen", "ingu", "erms", "isso", "also", "plef", "trit", "issi", "stio", "esti", "coll", "tsma", "arko", "llso", "bles", "uati", "namp", "call", "riat", "eplu"; two of these are real English words (three if you count "coll", which I don't; too obscure), so perhaps 10% of the samples are real words. A list of 20 word samples from the second model is: "hate", "ther", "sout", "vect", "nces", "ffer", "msua", "ergu", "blef", "hest", "assu", "fhsp", "ults", "lend", "lsoc", "fysj", "uscr", "ithi", "prow", "lith"; four of these are real English words (you might need to look up "lith", but I refuse to count "hest" as being too archaic), so perhaps 20% of the samples are real words. In each case, the samples are too small to take the fraction estimates all that seriously.

Letter models can be good enough for (say) evaluating communication devices, but they're not great at producing words (example 13.7). More effective language models are obtained by working with words. The recipe is as above, but now we use words in place of letters. It turns out that this recipe applies to such domains as protein sequencing, dna sequencing and music synthesis as well, but now we use amino acids (resp. base pairs; notes) in place of letters. Generally, one decides what the basic item is (letter, word, amino acid, base pair, note, etc.). Then individual items are called **unigrams** and 0'th order models are **unigram models**; pairs are **bigrams** and first order models are **bigram models**; triples are **trigrams**, second order models **trigram models**; and for any other $n$, groups of $n$ in sequence are **n-grams** and $n-1$'th order models are **n-gram models**.

**Worked example 13.8** *Modelling text with n-grams of words*

Build a text model that uses bigrams (resp. trigrams, resp. n-grams) of words, and look at the paragraphs that your model produces.

**Solution:** This is actually a fairly arduous assignment, because it is hard to get good bigram frequencies without working with enormous text resources. Google publishes n-gram models for English words with the year in which the n-gram occurred and information about how many different books it occurred in. So, for example, the word "circumvallate" appeared 335 times in 1978, in 91 distinct books – some books clearly felt the need to use this term more than once. This information can be found starting at http://storage.googleapis.com/books/ngrams/books/datasetsv2.html. The raw dataset is huge, as you would expect. There are numerous n-gram language models on the web. Jeff Attwood has a brief discussion of some models at https://blog.codinghorror.com/markov-and-you/; Sophie Chou has some examples, and pointers to code snippets and text resources, at http://blog.sophiechou.com/2013/how-to-model-markov-chains/. Fletcher Heisler, Michael Herman, and Jeremy Johnson are authors of RealPython, a training course in Python, and give a nice worked example of a Markov chain language generator at https://realpython.com/blog/python/lyricize-a-flask-app-to-create-lyrics-using-markov-chains/. Markov chain language models are effective tools for satire. Garkov is Josh Millard's tool for generating comics featuring a well-known cat (at http://joshmillard.com/garkov/). There's a nice Markov chain for reviewing wines by Tony Fischetti at http://www.onthelambda.com/2014/02/20/how-to-fake-a-sophisticated-knowledge-of-wine-with-markov-chains/

It is usually straightforward to build a unigram model, because it is usually easy to get enough data to estimate the frequencies of the unigrams. There are many more bigrams than unigrams, many more trigrams than bigrams, and so on. This means that estimating frequencies can get tricky. In particular, you might need to collect an immense amount of data to see every possible n-gram several times. Without seeing every possible n-gram several times, you will need to deal with estimating the probability of encountering rare n-grams *that you haven't seen*. Assigning these n-grams a probability of zero is unwise, because that implies that they *never* occur, as opposed to occur seldom.

There are a variety of schemes for **smoothing** data (essentially, estimating the probability of rare items that have not been seen). The simplest one is to assign some very small fixed probability to every n-gram that has a zero count. It turns out that this is not a particularly good approach, because, for even quite small $n$, the fraction of n-grams that have zero count can be very large. In turn, you can find that most of the probability in your model is assigned to n-grams you have never seen. An improved version of this model assigns a fixed probability to unseen n-grams, then divides that probability up between all of the n-grams that

have never been seen before. This approach has its own characteristic problems. It ignores evidence that some of the unseen n-grams are more common than others. Some of the unseen n-grams have (n-1) leading terms that are (n-1)-grams that we *have* observed. These (n-1)-grams likely differ in frequency, suggesting that n-grams involving them should differ in frequency, too. More sophisticated schemes are beyond our scope, however.

## 13.2 HIDDEN MARKOV MODELS AND DYNAMIC PROGRAMMING

Imagine we wish to build a program that can transcribe speech sounds into text. Each small chunk of text can lead to one, or some, sounds, and some randomness is involved. For example, some people pronounce the word "fishing" rather like "fission". As another example, the word "scone" is sometimes pronounced rhyming with "stone", sometimes rhyming with "gone", and very occasionally rhyming with "toon" (really!). A Markov chain supplies a model of all possible text sequences, and allows us to compute the probability of any particular sequence. We will use a Markov chain to model text sequences, but what we observe is sound. We must have a model of how sound is produced by text. With that model and the Markov chain, we want to produce text that (a) is a likely sequence of words and (b) is likely to have produced the sounds we hear.

Many applications contain the main elements of this example. We might wish to transcribe music from sound. We might wish to understand American sign language from video. We might wish to produce a written description of how someone moves from video observations. We might wish to break a substitution cipher. In each case, what we want to recover is a sequence that can be modelled with a Markov chain, but we don't see the states of the chain. Instead, we see noisy measurements that *depend* on the state of the chain, and we want to recover a state sequence that is (a) likely under the Markov chain model and (b) likely to have produced the measurements we observe.

### 13.2.1 Hidden Markov Models

Assume we have a finite state, time homogenous Markov chain, with $S$ states. This chain will start at time 1, and the probability distribution $P(X_1 = i)$ is given by the vector $\pi$. At time $u$, it will take the state $X_u$, and its transition probability matrix is $p_{ij} = P(X_{u+1} = j|X_u = i)$. We do not observe the state of the chain. Instead, we observe some $Y_u$. We will assume that $Y_u$ is also discrete, and there are a total of $O$ possible states for $Y_u$ for any $u$. We can write a probability distribution for these observations $P(Y_u|X_u = i) = q_i(Y_u)$. This distribution is the **emission distribution** of the model. For simplicity, we will assume that the emission distribution does not change with time.

We can arrange the emission distribution into a matrix $\mathcal{Q}$. A **hidden Markov model** consists of the transition probability distribution for the states, the relationship between the state and the probability distribution on $Y_u$, and the initial distribution on states, that is, $(\mathcal{P}, \mathcal{Q}, \pi)$. These models are often dictated by an application. An alternative is to build a model that best fits a collection of observed data, but doing so requires technical machinery we cannot expound here.

I will sketch how one might build a model for transcribing speech, but you

should keep in mind this is just a sketch of a very rich area. We can obtain the probability of a word following some set of words using n-gram resources, as in section 13.1.3. We then build a model of each word in terms of small chunks of word that are likely to correspond to common small chunks of sound. We will call these chunks of sound **phonemes**. We can look up the different sets of phonemes that correspond to a word using a pronunciation dictionary. We can combine these two resources into a model of how likely it is one will pass from one phoneme inside a word to another, which might either be inside this word or inside another word. We now have $\mathcal{P}$. We will not spend much time on $\pi$, and might even model it as a uniform distribution. We can use a variety of strategies to build $\mathcal{Q}$. One is to build discrete features of a sound signal, then count how many times a particular set of features is produced when a particular phoneme is played.

### 13.2.2 Picturing Inference with a Trellis

Assume that we have a sequence of $N$ measurements $Y_i$ that we believe to be the output of a known hidden Markov model. We wish to recover the "best" corresponding sequence of $X_i$. Doing so is **inference**. We will choose the sequence that maximizes the posterior probability of $X_1, \ldots, X_N$, conditioned on the observations and the model, which is

$$P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N, \mathcal{P}, \mathcal{Q}, \pi).$$

This is **maximum a posteriori** inference (or **MAP** inference).

It is equivalent to to recover a sequence $X_i$ that minimises

$$-\log P(X_1, X_2, \ldots, X_N | Y_1, Y_2, \ldots, Y_N, \mathcal{P}, \mathcal{Q}, \pi).$$

This is more convenient, because (a) the log turns products into sums, which will be convenient and (b) minimizing the negative log probability gives us a formulation that is consistent with algorithms in chapter 14. The negative log probability factors as

$$-\log\left(\frac{P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)}{P(Y_1, Y_2, \ldots, Y_N)}\right)$$

and this is

$$-\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi) + \log P(Y_1, Y_2, \ldots, Y_N).$$

Notice that $P(Y_1, Y_2, \ldots, Y_N)$ doesn't depend on the sequence of $X_u$ we choose, and so the second term can be ignored. What is important here is that we can decompose $-\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)$ in a very useful way, because the $X_u$ form a Markov chain. We want to minimize

$$-\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N | \mathcal{P}, \mathcal{Q}, \pi)$$

but this is

$$-\begin{bmatrix} \log P(X_1) + \log P(Y_1|X_1) + \\ \log P(X_2|X_1) + \log P(Y_2|X_2) + \\ \ldots \\ \log P(X_N|X_{n-1}) + \log P(Y_N|X_N). \end{bmatrix}$$

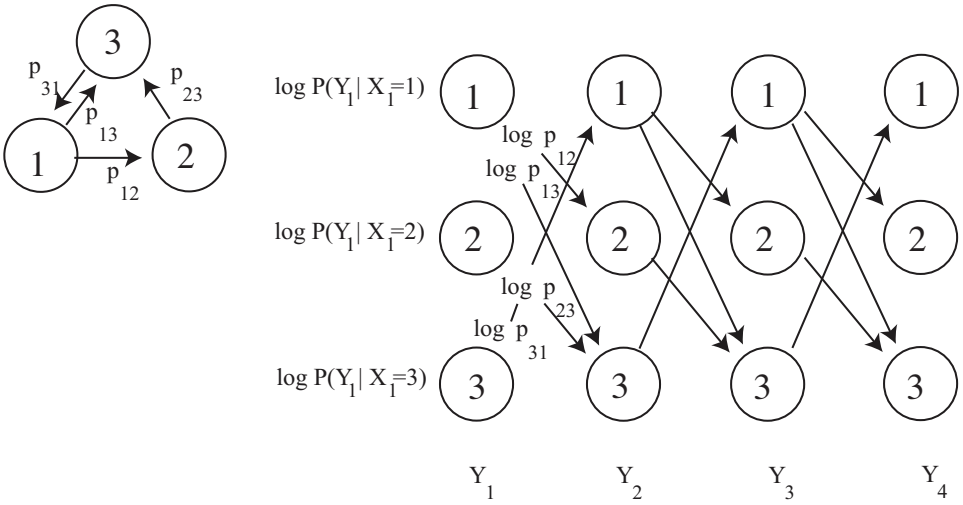


FIGURE 13.6: *At the* **top left**, *a simple state transition model. Each outgoing edge has some probability, though the topology of the model forces two of these probabilities to be* 1. *Below, the trellis corresponding to that model. Each path through the trellis corresponds to a legal sequence of states, for a sequence of three measurements. We weight the arcs with the log of the transition probabilities, and the nodes with the log of the emission probabilities. I have shown some weights*

Notice that this cost function has an important structure. It is a sum of terms. There are terms that depend on a single $X_i$ (unary terms) and terms that depend on two (binary terms). Any state $X_i$ appears in at most two binary terms.

We can illustrate this cost function in a structure called a **trellis**. This is a weighted, directed graph consisting of $N$ copies of the state space, which we arrange in columns. There is a column corresponding to each measurement. We add a directed arrow from any state in the $u$'th column to any state in the $u+1$'th column if the transition probability between the states isn't 0. This represents the fact that there is a possible transition between these states. We then label the trellis with weights. We weight the node representing the case that state $X_u = j$ in the column corresponding to $Y_u$ with $-\log P(Y_u|X_u = j)$. We weight the arc from the node representing $X_u = i$ to that representing $X_{u+1} = j$ with $-\log P(X_{u+1} = j|X_u = i)$.

The trellis has two crucial properties. Each directed path through the trellis from the start column to the end column represents a legal sequence of states. Now for some directed path from the start column to the end column, sum all the weights for the nodes and edges along this path. This sum is the negative log of the joint probability of that sequence of states with the measurements. You can verify each of these statements easily by reference to a simple example (try Figure 13.6)

There is an efficient algorithm for finding the path through a trellis which maximises the sum of terms. The algorithm is usually called **dynamic programming** or the **Viterbi algorithm**. I will describe this algorithm both in narrative, and as a recursion. We could proceed by finding, for each node in the first column,

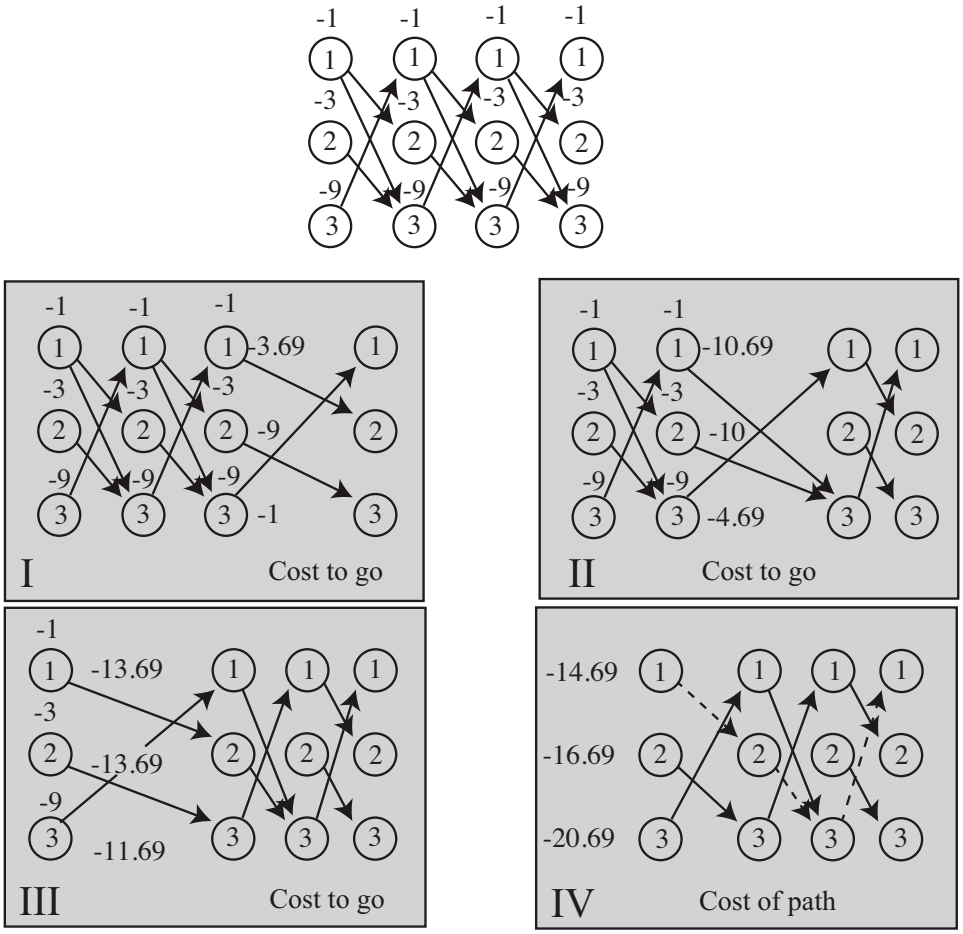


FIGURE 13.7: *An example of finding the best path through a trellis. The probabilities of leaving a node are uniform (and remember,* $\ln 2 \approx -0.69$*). Details in the text.*

the best path from that node to any node in the last. There are $S$ such paths, one for each node in the first column. Once we have these paths, we can choose the one with highest log joint probability. Now consider one of these paths. It passes through the $i$'th node in the $u$'th column. The path segment from this node to the end column must, itself, be the best path from this node to the end. If it wasn't, we could improve the original path by substituting the best. This is the key insight that gives us an algorithm.

Start at the *final* column of the tellis. We can evaluate the best path from each node in the final column to the final column, because that path is just the node, and the value of that path is the node weight. Now consider a two-state path, which will start at the second last column of the trellis (look at panel I in Figure 13.7). We can easily obtain the value of the best path leaving each node in this column. Consider a node: we know the weight of each arc leaving the node

and the weight of the node at the far end of the arc, so we can choose the path segment with the largest value of the sum; this arc is the best we can do leaving that node. This sum is the best value obtainable on leaving that node—which is often known as the **cost to go function**.

Now, because we know the best value obtainable on leaving each node in the second-last column, we can figure out the best value obtainable on leaving each node in the third-last column (panel II in Figure 13.7). At each node in the third-last column, we have a choice of arcs. Each of these reaches a node *from which we know the value of the best path*. So we can choose the best path leaving a node in the third-last column by finding the path that has the best value of: the arc weight leaving the node; the weight of the node in the second-last column the arc arrives at; and the value of the path leaving that node. This is much more easily done than described. All this works just as well for the third-last column, etc. (panel III in Figure 13.7) so we have a recursion. To find the value of the best path with $X_1 = i$, we go to the corresponding node in the first column, then add the value of the node to the value of the best path leaving that node (panel IV in Figure 13.7). Finally, to find the value of the best path leaving the first column, we compute the minimum value over all nodes in the first column.

We can also get the path with the minimum likelihood value. When we compute the value of a node, we erase all but the best arc leaving that node. Once we reach the first column, we simply follow the path from the node with the best value. This path is illustrated by dashed edges in Figure 13.7 (panel IV).

### 13.2.3 Dynamic Programming for HMM's: Formalities

We will formalize the recursion of the previous section with two ideas. First, we define $C_w(j)$ to be the cost of the best path segment to the end of the trellis *leaving* the node representing $X_w = j$. Second, we define $B_w(j)$ to be the node in column $w + 1$ that lies on the best path *leaving* the node representing $X_w = j$. So $C_w(j)$ tells you the cost of the best path, and $B_w(j)$ tells you what node is next on the best path.

Now it is straightforward to find the cost of the best path leaving each node in the second last column, and also the path. In symbols, we have

$$C_{N-1}(j) = \min_u \left[ -\log P(X_N = u | X_{N-1} = j) - \log P(Y_N | X_N = u) \right]$$

and

$$B_{N-1}(j) = \operatorname*{argmin}_u \left[ -\log P(X_N = u | X_{N-1} = j) - \log P(Y_N | X_N = u) \right].$$

You should check this against step I of Figure 13.7

Once we have the best path leaving each node in the $w + 1$'th column and its cost, it's straightforward to find the best path leaving the $w$'th column and its cost. In symbols, we have

$$C_w(j) = \min_u \left[ -\log P(X_{w+1} = u | X_w = j) - \log P(Y_{w+1} | X_{w+1} = u) - C_{w+1}(u) \right]$$

and

$$B_w(j) = \operatorname*{argmin}_u \left[ -\log P(X_{w+1} = u | X_w = j) - \log P(Y_{w+1} | X_{w+1} = u) - C_{w+1}(u) \right].$$

Check this against steps II and III in Figure 13.7.

Now finding the best path is easy. We run the recursion until we have $C_1(j)$ for each $j$. This gives the cost of the best path leaving the $j$'th node in column 1. We choose the node with the best cost, say $\hat{j}$. The next node on the best path is $B_1(\hat{j})$; and the path is $B_1(\hat{j}), B_2(B_1(\hat{j})), \ldots$.

### 13.2.4 Example: Simple Communication Errors

Hidden Markov models can be used to correct text errors. We will simplify somewhat, and assume we have text that has no punctuation marks, and no capital letters. This means there are a total of 27 symbols (26 lower case letters, and a space). We send this text down some communication channel. This could be a telephone line, a fax line, a file saving procedure or anything else. This channel makes errors independently at each character. For each location, with probability $1-p$ the output character at that location is the same as the input character. With probability $p$, the channel chooses randomly between the character one ahead or one behind in the character set, and produces that instead. You can think of this as a simple model for a mechanical error in one of those now ancient printers where a character strikes a ribbon to make a mark on the paper. We must reconstruct the transmission from the observations.

| * | e | t | i | a | o | s | n | r | h |
|---|---|---|---|---|---|---|---|---|---|
| 1.9e-1 | 9.7e-2 | 7.9e-2 | 6.6e-2 | 6.5e-2 | 5.8e-2 | 5.5e-2 | 5.2e-2 | 4.8e-2 | 3.7e-2 |

TABLE 13.1: *The most common single letters (unigrams) that I counted from a draft of this chapter, with their probabilities. The "*" stands for a space. Spaces are common in this text, because I have tended to use short words (from the probability of the "*", average word length is between five and six letters).*

I built a unigram model, a bigram model, and a trigram model. I stripped the text of this chapter of punctuation marks and mapped the capital letters to lower case letters. I used an HMM package (in my case, for Matlab; but there's a good one for R as well) to perform inference. The main programming here is housekeeping to make sure the transition and emission models are correct. About 40% of the bigrams and 86% of the trigrams did not appear in the text. I smoothed the bigram and trigram probabilities by dividing the probability 0.01 evenly between all unobserved bigrams (resp. trigrams). The most common unigrams, bigrams and trigrams appear in the tables. As an example sequence, I used

> "the trellis has two crucial properties each directed path through the trellis from the start column to the end column represents a legal sequence of states now for some directed path from the start column to the end column sum all the weights for the nodes and edges along this path this sum is the log of the joint probability of that sequence of states

| Lead char | | | | | |
|---|---|---|---|---|---|
| * | *t (2.7e-2) | *a (1.7e-2) | *i (1.5e-2) | *s (1.4e-2) | *o (1.1e-2) |
| e | e* (3.8e-2) | er (9.2e-3) | es (8.6e-3) | en (7.7e-3) | el (4.9e-3) |
| t | th (2.2e-2) | t* (1.6e-2) | ti (9.6e-3) | te (9.3e-3) | to (5.3e-3) |
| i | in (1.4e-2) | is (9.1e-3) | it (8.7e-3) | io (5.6e-3) | im (3.4e-3) |
| a | at (1.2e-2) | an (9.0e-3) | ar (7.5e-3) | a* (6.4e-3) | al (5.8e-3) |
| o | on (9.4e-3) | or (6.7e-3) | of (6.3e-3) | o* (6.1e-3) | ou (4.9e-3) |
| s | s* (2.6e-2) | st (9.4e-3) | se (5.9e-3) | si (3.8e-3) | su (2.2e-3) |
| n | n* (1.9e-2) | nd (6.7e-3) | ng (5.0e-3) | ns (3.6e-3) | nt (3.6e-3) |
| r | re (1.1e-2) | r* (7.4e-3) | ra (5.6e-3) | ro (5.3e-3) | ri (4.3e-3) |
| h | he (1.4e-2) | ha (7.8e-3) | h* (5.3e-3) | hi (5.1e-3) | ho (2.1e-3) |

TABLE 13.2: *The most common bigrams that I counted from a draft of this chapter, with their probabilities. The '*' stands for a space. For each of the 10 most common letters, I have shown the five most common bigrams with that letter in the lead. This gives a broad view of the bigrams, and emphasizes the relationship between unigram and bigram frequencies. Notice that the first letter of a word has a slightly different frequency than letters (top row; bigrams starting with a space are first letters). About* 40% *of the possible bigrams do not appear in the text.*

| *th | the | he* | is* | *of | of* | on* | es* | *a* | ion |
|---|---|---|---|---|---|---|---|---|---|
| 1.7e-2 | 1.2e-2 | 9.8e-3 | 6.2e-3 | 5.6e-3 | 5.4e-3 | 4.9e-3 | 4.9e-3 | 4.9e-3 | 4.9e-3 |
| tio | e*t | in* | *st | *in | at* | ng* | ing | *to | *an |
| 4.6e-3 | 4.5e-3 | 4.2e-3 | 4.1e-3 | 4.1e-3 | 4.0e-3 | 3.9e-3 | 3.9e-3 | 3.8e-3 | 3.7e-3 |

TABLE 13.3: *The most frequent 10 trigrams in a draft of this chapter, with their probabilities. Again, '*' stands for space. You can see how common 'the' and '*a*' are; 'he*' is common because '*the*' is common. About* 80% *of possible trigrams do not appear in the text.*

> with the measurements you can verify each of these statements easily by reference to a simple example"

(which is text you could find in a draft of this chapter). There are 456 characters in this sequence.

When I ran this through the noise process with $p = 0.0333$, I got

> "theztrellis has two crucial properties each directed path through the tqdllit from the start column to the end coluln represents a legal sequencezof states now for some directed path from the start column to thf end column sum aml the veights for the nodes and edges along this path this sum is the log of the joint probability oe that sequence of states wish the measurements youzcan verify each of these statements easily by reference to a simple examqle"

which is mangled but not too badly (13 of the characters are changed, so 443 locations are the same).

The unigram model produces

> "the trellis has two crucial properties each directed path through the tqdllit from the start column to the end coluln represents a legal sequence of states now for some directed path from the start column to thf end column sum aml the veights for the nodes and edges along this path this sum is the log of the joint probability oe that sequence of states wish the measurements you can verify each of these statements easily by reference to a simple examqle"

which fixes three errors. The unigram model only changes an observed character when the probability of encountering that character on its own is less than the probability it was produced by noise. This occurs only for "z", which is unlikely on its own and is more likely to have been a space. The bigram model produces

> "she trellis has two crucial properties each directed path through the trellit from the start column to the end coluln represents a legal sequence of states now for some directed path from the start column to the end column sum aml the veights for the nodes and edges along this path this sum is the log of the joint probability oe that sequence of states wish the measurements you can verify each of these statements easily by reference to a simple example"

This is the same as the correct text in 449 locations, so somewhat better than the noisy text. The trigram model produces

> "the trellis has two crucial properties each directed path through the trellit from the start column to the end column represents a legal sequence of states now for some directed path from the start column to the end column sum all the weights for the nodes and edges along this path this sum is the log of the joint probability of that sequence of states with the measurements you can verify each of these statements easily by reference to a simple example"

which corrects all but one of the errors (look for "trellit").

## 13.3 LEARNING AN HMM

There are two very distinct cases for learning an HMM. In the first case, the hidden states have known and important semantics. For example, the hidden states could be words, or letters. In this case, we want any model we learn to respect the semantics of the hidden states. For example, the model should recover the right words to go with ink or sound or whatever. This case is straightforward (section 13.3.1).

In the second case, we want to model sequences, but the hidden states are just a modelling device. One example is motion capture data. Various devices can be used to measure the position of human joints in space while a person is moving around. These devices report position as a function of time. This kind of data is extremely useful in making computer generated imagery (CGI) for films and computer games. We might observe some motion capture data, and try to make more (HMM's actually do this quite poorly). As another example, we might observe stock price data, and try to make more. As yet another example, we might

observe encrypted text and want to make more encrypted text (for example, to confuse the people encrypting the text). This case isn't like the examples I've used to describe or justify HMM's, but it occurs fairly often. In this case, EM is an effective learning algorithm (section 13.3.2).

### 13.3.1 When the States Have Meaning

There are two interesting versions of this case. In one, we see example sequences of $X_i$ with corresponding $Y_i$. Since everything is discrete, building models of $P(X_{i+1}|X_i)$ and of $P(Y|X)$ is straightforward – one simply counts. This assumes that there is enough example data. When there is not – usually signalled by zero counts for some cases – one must use smoothing methods that are beyond our scope. If one has this kind of data, it is possible to build other kinds of sequence model; I describe these models in the following chapter.

In the second case, we see example sequences of $X_i$ but do not see the $Y_i$ corresponding to those sequences. A standard example is a substitution cipher for English text. Here it is easy to get a lot of data for $P(X_{i+1}|X_i)$ (one looks up text resources in English), but we have no data for $P(Y|X)$ because we do not know what $X$ corresponds to observed $Y$'s. Learning in this case is a straightforward variant of the EM algorithm for learning when there is no $X$ data (below; for the variant, see the exercises).

### 13.3.2 Learning an HMM with EM

We have a dataset $\mathbf{Y}$ for which we believe a hidden Markov model is an appropriate model. This dataset consists of $R$ sequences of visible states. The $u$'th sequence has $N(u)$ elements. We will assume that the observed values lie in a discrete space (i.e. there are $O$ possible values that the $Y$'s can take, and no others). We wish to choose a model that best represents a set of data. Assume, for the moment, that we knew each hidden state corresponding to each visible state. Write $Y_t^{(u)}$ is the observed value for the $t$'th observed state in the $u$'th sequence; write $X_t^{(u)}$ for the random variable representing the hidden value for the $t$'th observed state in the $u$'th sequence; write $s_k$ for the hidden state values (where $k$ is in the range $1 \dots S$); and write $y_k$ for the possible values for $Y$ (where $k$ is in the range $1 \dots O$).

The hidden Markov model is given by three sets of parameters, $\pi$, $\mathcal{P}$ and $\mathcal{Q}$. We will assume that these parameters are not affected by where we are in the sequence (i.e. the model is homogeneous). First, $\pi$ is an $S$ dimensional vector. The $i$'th element, $\pi_i$ of this vector gives the probability that the model starts in state $s_i$, i.e. $\pi_i = P(X_1 = s_i|\theta)$. Second, $\mathcal{P}$ is an $S \times S$ dimensional table. The $i$, $j$'th element of this table gives $P(X_{t+1} = s_j|X_t = s_i)$. Finally, $\mathcal{Q}$ is an $O \times S$ dimensional table. We will write $q_j(y_i) = P(Y_t = y_i|X_t = s_j)$ for the $i$, $j$'th element of this table. Note I will write $\theta$ to represent all of these parameters together.

Now assume that we *know* the values of $X_t^{(u)}$ for all $t$, $u$, (i.e. for each $Y_t^{(u)}$ we know that $X_t^{(u)} = s_i$). Then estimating the parameters is straightforward. We can estimate each by counting. For example, we estimate $\pi_i$ by counting the number of sequences where $X_1 = s_i$, then dividing by the total number of sequences. We

will encapsulate this knowledge in a function $\delta_t^{(u)}(i)$, where

$$\delta_t^{(u)}(i) = \begin{cases} 1 & \text{if } X_t^{(u)} = s_i \\ 0 & \text{otherwise} \end{cases} .$$

If we know $\delta_t^{(u)}(i)$, we have

$$\begin{aligned} \pi_i &= \frac{\text{number of times in } s_i \text{ at time 1}}{\text{number of sequences}} \\ &= \frac{\sum_{u=1}^{R} \delta_1^{(u)}(i)}{R} \\ \mathcal{P}_{ij} &= \frac{\text{number of transitions from } s_j \text{ to } s_i}{\text{total number of transitions}} \\ &= \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)-1} \delta_t^{(u)}(j)\delta_{t+1}^{(u)}(i)}{\sum_{u=1}^{R} [N(u)-1]} \\ q_j(y_i) &= \frac{\text{number of times in } s_j \text{ and observe } Y = y_i}{\text{number of times in } s_j} \\ &= \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \delta_t^{(u)}(j)\delta(Y_t^{(u)}, y_i)}{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \delta_t^{(u)}(j)} \end{aligned}$$

where $\delta(u, v)$ is one if its arguments are equal and zero otherwise.

The problem (of course) is that we *don't know* $\delta_t^{(u)}(i)$. But we have been here before (section 105 and section 105). The situation follows the recipe for EM: we have missing variables (the $X_t^{(u)}$; or, equivalently, the $\delta_t^{(u)}(i)$) where the log-likelihood can be written out cleanly in terms of the missing variables. We assume we know an estimate of the parameters $\hat{\theta}^{(n)}$. We construct

$$Q(\theta; \hat{\theta}^{(n)}) = \mathbb{E}_{P(\delta|Y,\hat{\theta}^{(n)})}[\log P(\delta, Y|\theta)]$$

(the E-step). Then we compute

$$\hat{\theta^{(n+1)}} = \underset{\theta}{\text{argmin}} \; Q(\theta; \hat{\theta}^{(n)})$$

(the M-step). As usual, the problem is the E-step. I will not derive this in detail (enthusiasts can easily reconstruct the derivation from what follows together with chapter 9). The essential point is that we need to recover

$$\xi_t^{(u)}(i) = \mathbb{E}_{P(\delta|Y,\hat{\theta}^{(n)})}\left[\delta_t^{(u)}(i)\right] = P(X_t^{(u)} = s_i|Y, \hat{\theta}^{(n)}).$$

For the moment, assume we know these. Then we have

$$\begin{aligned}
\hat{\pi_i}^{(n+1)} &= \text{expected frequency of being in } s_i \text{ at time 1} \\
&= \frac{\sum_{u=1}^{R} \xi_1^{(u)}(i)}{R} \\
\hat{\mathcal{P}}_{ij}^{(n+1)} &= \frac{\text{expected number of transitions from } s_j \text{ to } s_i}{\text{expected number of transitions from state } s_j} \\
&= \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j) \xi_{t+1}^{(u)}(i)}{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)} \\
\hat{q}_j^{(n+1)}(k) &= \frac{\text{expected number of times in } s_j \text{ and observing } Y = y_k}{\text{expected number of times in state } s_j} \\
&= \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j) \delta(Y_t^{(u)}, y_k)}{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)}
\end{aligned}$$

where $\delta(u, v)$ is one if its arguments are equal and zero otherwise.

To evaluate $\xi_t^{(u)}(i)$, we need two intermediate variables: a **forward variable** and a **backward variable**. The forward variable is

$$\alpha_t^{(u)}(j) = P(Y_1^{(u)}, \ldots, Y_t^{(u)}, X_t^{(u)} = s_j | \hat{\theta}^{(n)}).$$

The backward variable is

$$\beta_t^{(u)}(j) = P(\{Y_{t+1}^{(u)}, Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)}\} | X_t^{(u)} = s_j | \hat{\theta}^{(n)}).$$

Now assume that we know the values of these variables, we have that

$$\begin{aligned}
\xi_t^{(u)}(i) &= P(X_t^{(u)} = s_i | \hat{\theta}^{(n)}, \mathbf{Y}^{(u)}) \\
&= \frac{P(\mathbf{Y}^{(u)}, X_t^{(u)} = s_i | \hat{\theta}^{(n)})}{P(\mathbf{Y}^{(u)} | \hat{\theta}^{(n)})} \\
&= \frac{\alpha_t^{(u)}(i) \beta_t^{(u)}(i)}{\sum_{i=1}^{S} \alpha_t^{(u)}(i) \beta_t^{(u)}(i)}
\end{aligned}$$

Both the forward and backward variables can be evaluated by induction. We get $\alpha_t^{(u)}(j)$ by observing that:

$$\begin{aligned}
\alpha_1^{(u)}(j) &= P(Y_1^{(u)}, X_1^{(u)} = s_j | \hat{\theta}^{(n)}) \\
&= \pi_j^{(n)} q_j^{(n)}(Y_1).
\end{aligned}$$

Now for all other $t$'s, we have

$$\begin{aligned}
\alpha_{t+1}^{(u)}(j) &= P(Y_1^{(u)}, \ldots, Y_{t+1}^{(u)}, X_{t+1}^{(u)} = s_j | \hat{\theta}^{(n)}) \\
&= \sum_{l=1}^{S} P(Y_1^{(u)}, \ldots, Y_t^{(u)}, Y_{t+1}^{(u)}, X_t^{(u)} = s_l, X_{t+1}^{(u)} = s_j | \hat{\theta}^{(n)}) \\
&= \left( \sum_{l=1}^{S} \left[ \begin{array}{c} P(Y_1^{(u)}, \ldots, Y_t^{(u)}, X_t^{(u)} = s_l | \hat{\theta}^{(n)}) \times \\ P(X_{t+1}^{(u)} = s_j | X_t^{(u)} = s_l, \hat{\theta}^{(n)}) \end{array} \right] \right) P(Y_{t+1}^{(u)} | X_{t+1}^{(u)} = s_j, \hat{\theta}^{(n)}) \\
&= \left[ \sum_{l=1}^{S} \alpha_t^{(u)}(l) p_{lj}^{(n)} \right] q_j^{(n)}(Y_{t+1})
\end{aligned}$$

We get $\beta_t^{(u)}(j)$ by observing that:

$$\begin{aligned}
\beta_{N(u)}^{(u)}(j) &= P(\text{no further output} | X_{N(u)}^{(u)} = s_j, \hat{\theta}^{(n)}) \\
&= 1.
\end{aligned}$$

Now for all other $t$ we have

$$\begin{aligned}
\beta_t^{(u)}(j) &= P(Y_{t+1}^{(u)}, Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)} | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \\
&= \sum_{l=1}^{S} \left[ P(Y_{t+1}^{(u)}, Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)}, X_{t+1}^{(u)} = s_l | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \right] \\
&= \sum_{l=1}^{S} \left[ \begin{array}{c} P(Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)} | X_{t+1}^{(u)} = s_j, \hat{\theta}^{(n)}) \\ \times P(Y_{t+1}^{(u)}, X_{t+1}^{(u)} = s_l | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \end{array} \right] \\
&= P(Y_{t+2}^{(u)}, \ldots, Y_{N(u)}^{(u)} | X_{t+1}^{(u)} = s_j, \hat{\theta}^{(n)}) \left( \sum_{l=1}^{S} \left[ \begin{array}{c} P(X_{t+1}^{(u)} = s_l | X_t^{(u)} = s_j, \hat{\theta}^{(n)}) \\ \times P(Y_{t+1}^{(u)} | X_{t+1}^{(u)} = s_l \hat{\theta}^{(n)}) \end{array} \right] \right) \\
&= \beta_{t+1}(j) \left( \sum_{l=1}^{S} \left[ q_l^{(n)}(Y_{t+1}^{(u)}) p_{lj}^{(n)} \right] \right)
\end{aligned}$$

As a result, we have a simple fitting algorithm, collected in Algorithm 13.1.

**Procedure: 13.1** *Fitting Hidden Markov Models with EM*

We fit a model to a data sequence $\mathbf{Y}$ is achieved by a version of EM. We seek the values of parameters $\theta = (\mathcal{P}, \mathcal{Q}, \pi)_i$. We assume we have an estimatel $\hat{\theta}^{(n)}$, and then compute the coefficients of a new model; this iteration is guaranteed to converge to a local maximum of $P(\mathbf{Y}|\hat{\theta})$.

Until $\hat{\theta}^{(n+1)}$ is the same as $\hat{\theta}^{(n)}$
compute the forward variables $\alpha$ and $\beta$
using the procedures of algorithms 13.2 and 13.3

compute $\xi_t^{(u)}(i) = \frac{\alpha_t^{(u)}(i)\beta_t^{(u)}(i)}{\sum_{i=1}^{S} \alpha_t^{(u)}(i)\beta_t^{(u)}(i)}$

compute the updated parameters using the procedures of procedure 13.4
end

**Procedure: 13.2** *Computing the Forward Variable for Fitting an HMM*

$$\begin{aligned} \alpha_1^{(u)}(j) &= \pi_j^{(n)} q_j^{(n)}(Y_1) \\ \alpha_{t+1}^{(u)}(j) &= \left[\sum_{l=1}^{S} \alpha_t^{(u)}(l) p_{lj}^{(n)}\right] q_j^{(n)}(Y_{t+1}) \end{aligned}$$

**Procedure: 13.3** *Computing the Backward Variable for Fitting an HMM*

$$\begin{aligned} \beta_{N(u)}^{(u)}(j) &= 1 \\ \beta_t^{(u)}(j) &= \beta_{t+1}(j) \left(\sum_{l=1}^{S} \left[q_l^{(n)}(Y_{t+1}^{(u)}) p_{lj}^{(n)}\right]\right) \end{aligned}$$

**Procedure: 13.4** *Updating Parameters for Fitting an HMM*

$$\hat{\pi_i}^{(n+1)} = \frac{\sum_{u=1}^{R} \xi_1^{(u)}(i)}{R}$$

$$\hat{\mathcal{P}}_{ij}^{(n+1)} = \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)\xi_{t+1}^{(u)}(i)}{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)}$$

$$\hat{q}_j(k)^{(n+1)} = \frac{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)\delta(Y_t^{(u)}, y_k)}{\sum_{u=1}^{R} \sum_{t=1}^{N(u)} \xi_t^{(u)}(j)}$$

where $\delta(u, v)$ is one if its arguments are equal and zero otherwise.

## 13.4 YOU SHOULD

### 13.4.1 remember these terms:

13.4.2 remember these facts:



13.4.3 be able to:

- Set up a simple HMM and use it to solve problems.
- Learn a simple HMM from data using EM

## PROBLEMS

**13.1. Multiple die rolls:** You roll a fair die until you see a 5, then a 6; after that, you stop. Write $P(N)$ for the probability that you roll the die $N$ times.
- **(a)** What is $P(1)$?
- **(b)** Show that $P(2) = (1/36)$.
- **(c)** Draw a directed graph encoding all the sequences of die rolls that you could encounter. Don't write the events on the edges; instead, write their probabilities. There are 5 ways not to get a 5, but only one probability, so this simplifies the drawing.
- **(d)** Show that $P(3) = (1/36)$.
- **(e)** Now use your directed graph to argue that $P(N) = (5/6)P(N-1) + (25/36)P(N-2)$.

**13.2. More complicated multiple coin flips:** You flip a fair coin until you see either $HTH$ or $THT$, and then you stop. We will compute a recurrence relation for $P(N)$.
- **(a)** Draw a directed graph for this chain.
- **(b)** Think of the directed graph as a finite state machine. Write $\Sigma_N$ for some string of length $N$ accepted by this finite state machine. Use this finite state machine to argue that $Sigma_N$ has one of four forms:
  1. $TT\Sigma_{N-2}$
  2. $HH\Sigma_{N-3}$
  3. $THH\Sigma_{N-2}$
  4. $HTT\Sigma_{N-3}$
- **(c)** Now use this argument to show that $P(N) = (1/2)P(N-2)+(1/4)P(N-3)$.

**13.3.** For the umbrella example of worked example 13.1, assume that with probability 0.7 it rains in the evening, and 0.2 it rains in the morning. I am conventional, and go to work in the morning, and leave in the evening.
- **(a)** Write out a transition probability matrix.
- **(b)** What is the stationary distribution? (you should use a simple computer program for this).
- **(c)** What fraction of evenings do I arrive at home wet?
- **(d)** What fraction of days do I arrive at my destination dry?

## PROGRAMMING EXERCISES

**13.4.** A dishonest gambler has two dice and a coin. The coin and one die are both fair. The other die is unfair. It has $P(n) = [0.5, 0.1, 0.1, 0.1, 0.1, 0.1]$ (where $n$ is the number displayed on the top of the die). At the start, the gambler chooses a die uniformly and at random. At each subsequent step, the gambler chooses a die by flipping a weighted coin. If the coin comes up heads (probability $p$), the gambler changes the die, otherwise, the gambler keeps the same die. The gambler rolls the chosen die.

- **(a)** Model this process with a hidden Markov model. The emitted symbols should be $1, \ldots, 6$. Doing so requires only two hidden states (which die is in hand). Simulate a long sequence of rolls using this model for the case $p = 0.01$ and $p = 0.5$. What difference do you see?
- **(b)** Use your simulation to produce 10 sequences of 100 symbols for the case $p = 0.1$. Record the hidden state sequence for each of these. Now recover the hidden state using dynamic programming (you should likely use a software package for this; there are many good ones for R and Matlab). What fraction of the hidden states is correctly identified by your inference procedure?

**13.5.** A dishonest gambler has two dice and a coin. The coin and one die are both fair. The other die is unfair. It has $P(n) = [0.5, 0.1, 0.1, 0.1, 0.1, 0.1]$ (where $n$ is the number displayed on the top of the die). At the start, the gambler chooses a die uniformly and at random. At each subsequent step, the gambler chooses a die by flipping a weighted coin. If the coin comes up heads (probability $p$), the gambler changes the die, otherwise, the gambler keeps the same die. The gambler rolls the chosen die.

- **(a)** Model this process with a hidden Markov model. The emitted symbols should be $1, \ldots, 6$. Doing so requires only two hidden states (which die is in hand). Produce one sequence of 1000 symbols for the case $p = 0.2$ with your simulator.
- **(b)** Use the sequence of symbols and EM to learn a hidden Markov model with two states.
- **(c)** It turns out to be difficult with the tools at our disposal to compare your learned model with the true model. Can you do this by inferring a sequence of hidden states using each model, then comparing the inferred sequences? Explain. *Hint:* No - but the reason requires a moment's thought.
- **(d)** Simulate a sequence of 1000 states using the learned model and also using the true model. For each sequence compute the fraction of 1's, 2's, etc. observed in the sequence. Does this give you any guide as to how good the learned model is? *Hint:* can you use the chi-squared test to tell if any differences you see are due to chance?

**13.6. Warning: this exercise is fairly elaborate, though straightforward.** We will correct text errors using a hidden Markov model.

- **(a)** Obtain the text of a copyright-free book in plain characters. One natural source is Project Gutenberg, at https://www.gutenberg.org. Simplify this text by dropping all punctuation marks except spaces, mapping capital letters to lower case, and mapping groups of many spaces to a single space. The result will have 27 symbols (26 lower case letters and a space). From this text, count unigram, bigram and trigram letter frequencies.
- **(b)** Use your counts to build models of unigram, bigram and trigram letter probabilities. You should build both an unsmoothed model, and at

least one smoothed model. For the smoothed models, choose some small amount of probability $\epsilon$ and split this between all events with zero count. Your models should differ only by the size of $\epsilon$.

**(c)** Construct a corrupted version of the text by passing it through a process that, with probability $p_c$, replaces a character with a randomly chosen character, and otherwise reports the original character.

**(d)** For a reasonably sized block of corrupted text, use an HMM inference package to recover the best estimate of your true text. Be aware that your inference will run more slowly as the block gets bigger, but you won't see anything interesting if the block is (say) too small to contain any errors.

**(e)** For $p_c = 0.01$ and $p_c = 0.1$, estimate the error rate for the corrected text for different values of $\epsilon$. Keep in mind that the corrected text could be worse than the corrupted text.

**13.7. Warning: this exercise is fairly elaborate, though straightforward.** We will break a substitution cipher using a hidden Markov model.

**(a)** Obtain the text of a copyright-free book in plain characters. One natural source is Project Gutenberg, at https://www.gutenberg.org. Simplify this text by dropping all punctuation marks except spaces, mapping capital letters to lower case, and mapping groups of many spaces to a single space. The result will have 27 symbols (26 lower case letters and a space). From this text, count unigram, bigram and trigram letter frequencies.

**(b)** Use your counts to build models of unigram, bigram and trigram letter probabilities. You should build both an unsmoothed model, and at least one smoothed model. For the smoothed models, choose some small amount of probability $\epsilon$ and split this between all events with zero count. Your models should differ only by the size of $\epsilon$.

**(c)** Construct a ciphered version of the text. We will use a substitution cipher, which you can represent as randomly chosen permutation of 27 points. You should represent this permutation as a $27 \times 27$ permutation matrix. Remember a permutation matrix contains only zeros and ones. Each column and each row contains exactly one 1. You can now represent each character with a 27 dimensional one-hot vector. This is a 27 dimensional vector. One component is 1, and the others are 0. For the $i$'th character, the $i$'th component is 1 (so an "a" is represented by a vector with 1 in the first component, etc.). The document becomes a sequence of these vectors. Now you can get a representation of the ciphered document by multiplying the representation of the original document by the permutation matrix.

**(d)** Using at least 10,000 ciphered characters apply EM, rather like section 13.3.2, to estimate an HMM. You should use the unigram model of the previous subexercise as the transition model, and you should *not* re-estimate the transition model. Instead, you should estimate the emission model and prior only. This is straightforward; plug the known transition model, the estimated emission model and the prior into the E-step, and then in the M-step update only the emission model and the prior.

**(e)** For a reasonably sized block of ciphered text, use an HMM inference package and your learned model to recover the best estimate of your true text. Be aware that your inference will run more slowly as the block gets bigger, but you won't see anything interesting if the block is (say) too small to contain any errors.

**(f)** Now perform the last two steps using your bigram and trigram models.

Which model deciphers with the lowest error rate?