Markov Chains and Hidden Markov Models 14

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.

This observation leads us to a powerful and useful model of sequences: the next item is produced by a probability that depends on a short set of previous items. This model has extremely useful applications when one wants to understand speech, sound or language.

14.1 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

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

This means that the transition probabilities do not change with time. 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.

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.

D. Forsyth, *Probability and Statistics for Computer Science*,
https://doi.org/10.1007/978-3-319-64410-3_14

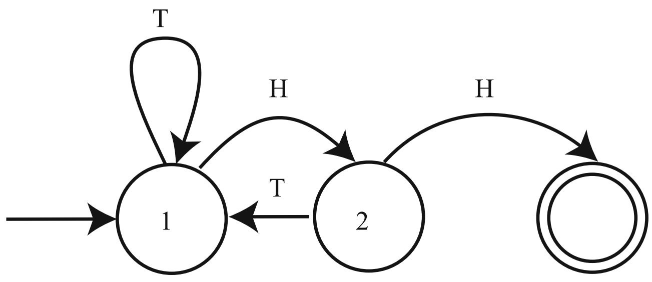

Fig. 14.1 A directed graph representing the coin flip example. By convention, the end state is a *double circle*, and the start state has an incoming arrow. I've labelled the arrows with the event that leads to the transition, but haven't bothered to put in the probabilities, because each is 0.5

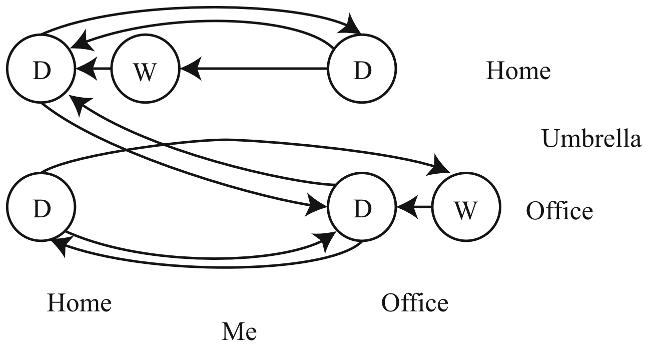

Fig. 14.2 A directed graph representing the umbrella example. Notice you can't arrive at the office wet with the umbrella at home (you'd have taken it), and so on. Labelling the edges with probabilities is left to the reader

Worked example 14.1 (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 Figure 14.1 shows a simple 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 $P(4 \text{ flips}) = 2/8 = 1/4$. We might want to answer significantly more interesting questions. For example, what is the probability that we must flip the coin more than ten times? It is often possible to answer these questions by analysis, but we will use simulations.

Worked example 14.2 (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 14.2 gives this chain. A more interesting question is with what probability I arrive at my destination wet? Again, we will solve this with simulation.

Notice an important difference between Examples 14.1 and 14.2. In the coin flip case, 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.

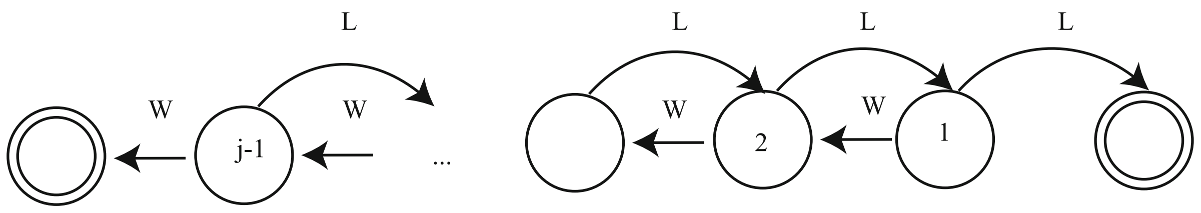

Fig. 14.3 A directed graph representing the gambler's ruin example. I have labelled each state with the amount of money the gambler has at that state. There are two end states, where the gambler has zero (is ruined), or has j and decides to leave the table. The problem we discuss is to compute the probability 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 14.3 (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 14.3 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 $s - 1$.

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.

Useful Facts 14.1 (Markov Chains)
A Markov chain is a sequence of random variables X_n with the property that,

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

14.1.1 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.

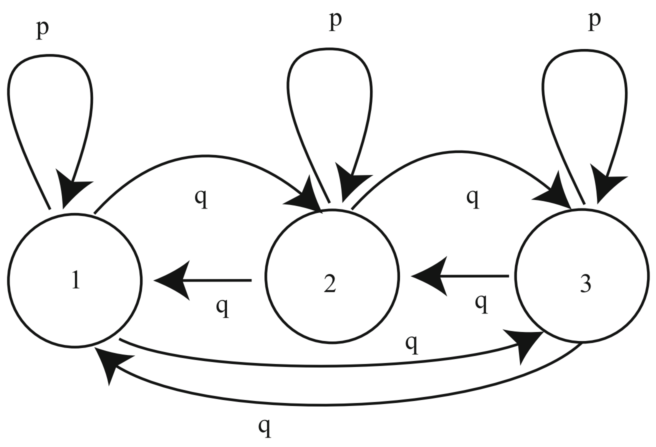

Fig. 14.4 A virus can exist in one of 3 strains. At the end of each year, the virus mutates. With probability α, it chooses uniformly and at random from one of the 2 other strains, and turns into that; with probability $1-\alpha$, it stays in the strain it is in. For this figure, we have transition probabilities $p = (1-\alpha)$ and $q = (\alpha/2)$

Worked example 14.4 (Viruses) Write out the transition probability matrix for the virus of Fig. 14.4, 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 π. 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 14.2 (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 π_{n-1}, then the probability distribution of state X_n is given by $\pi_{n-1}^T \mathcal{P}$.

14.1.2 Stationary Distributions

Worked example 14.5 (Viruses) We know that the virus of Fig. 14.4 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 α tends to stay in whatever state it is in, the distribution of states after 2 years is still quite peaked; when α 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 14.5, 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 π,

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

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 14.6). 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.

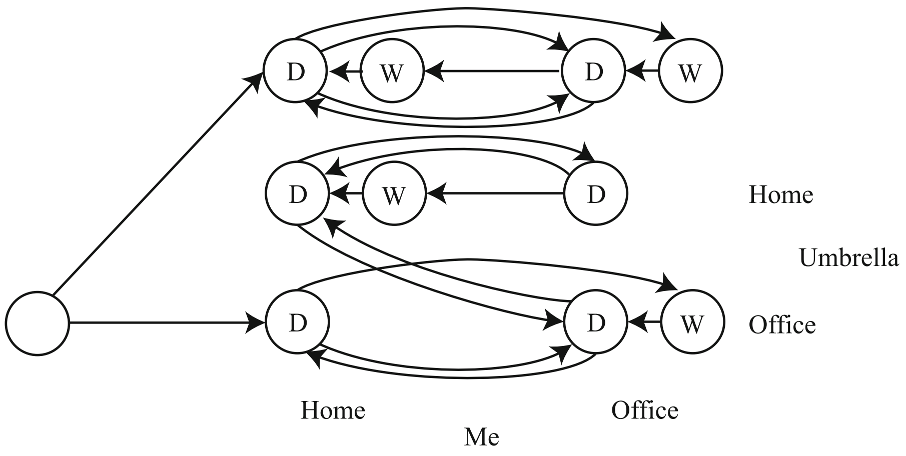

Fig. 14.5 In this umbrella example, there can't be a stationary distribution; what happens depends on the initial, random choice of buying/not buying an umbrella

Worked example 14.6 (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 14.2. If I have not, I just get wet. Illustrate this Markov chain with a state diagram.

Solution Figure 14.5 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 14.3 (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.

14.1.3 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. 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 14.7 (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 14.7). 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 14.8 (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. Rather than solve it, I will follow the grand tradition of looking at other people's work.

There are a variety of places you can find 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/.

The paragraphs that cleverly trained Markov chain language models produce can be hilarious, and are very 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.

14.2 Estimating Properties of Markov Chains

Many problems in probability can be worked out in closed form if one knows enough combinatorial mathematics, or can come up with the right trick. Textbooks are full of these, and we've seen some. Explicit formulas for probabilities are often extremely useful. But it isn't always easy or possible to find a formula for the probability of an event in a model. Markov chains are a particularly rich source of probability problems that might be too much trouble to solve in closed form. An alternative strategy is to build a simulation, run it many times, and count the fraction of outcomes where the event occurs. This is a simulation experiment.

14.2.1 Simulation

Imagine we have a random variable X with probability distribution $P(X)$ that takes values in some domain D. Assume that we can easily produce independent simulations, and that we wish to know $\mathbb{E}[f]$, the expected value of the function f under the distribution $P(X)$.

The weak law of large numbers tells us how to proceed. Define a new random variable $F = f(X)$. This has a probability distribution $P(F)$, which might be difficult to know. We want to estimate $\mathbb{E}[f]$, the expected value of the function f under the distribution $P(X)$. This is the same as $\mathbb{E}[F]$. Now if we have a set of IID samples of X, which we write x_i, then we can form a set of IID samples of F by forming $f(x_i) = f_i$. Write

$$F_N = \frac{\sum_{i=1}^{N} f_i}{N}.$$

This is a random variable, and the weak law of large numbers gives that, for any positive number ϵ

$$\lim_{N\to\infty} P(\{\|F_N - \mathbb{E}[F]\| > \epsilon\}) = 0.$$

You can interpret this as saying that, that for a set of IID random samples x_i, the probability that

$$\frac{\sum_{i=1}^{N} f(x_i)}{N}$$

is very close to $\mathbb{E}[f]$ is high for large N

Worked example 14.9 (Computing an Expectation) Assume the random variable X is uniformly distributed in the range $[0-1]$, and the random variable Y is uniformly distributed in the range $[0-10]$. X and Z are independent. Write $Z = (Y - 5X)^3 - X^2$. What is $\text{var}(\{Z\})$?

Solution With enough work, one could probably work this out in closed form. An easy program will get a good estimate. We have that $\text{var}(\{Z\}) = \mathbb{E}[Z^2] - \mathbb{E}[Z]^2$. My program computed 1000 values of Z (by drawing X and Y from the appropriate random number generator, then evaluating the function). I then computed $\mathbb{E}[Z]$ by averaging those values, and $\mathbb{E}[Z]^2$ by averaging their squares. For a run of my program, I got $\text{var}(\{Z\}) = 2.76 \times 10^4$.

You can compute a probability using a simulation, too, because a probability can be computed by taking an expectation. Recall the property of indicator functions that

$$\mathbb{E}\left[\mathbb{I}_{[\mathcal{E}]}\right] = P(\mathcal{E})$$

(Section 4.3.3). This means that computing the probability of an event $\mathcal{E}$ involves writing a function that is 1 when the event occurs, and 0 otherwise; we then estimate the expected value of that function.

Worked example 14.10 (Computing a Probability for Multiple Coin Flips) You flip a fair coin three times. Use a simulation to estimate the probability that you see three H's.

Solution You really should be able to work this out in closed form. But it's amusing to check with a simulation. I wrote a simple program that obtained a 1000×3 table of uniformly distributed random numbers in the range $[0-1]$. For each number, if it was greater than 0.5 I recorded an H and if it was smaller, I recorded a T. Then I counted the number of rows that had 3 H's (i.e. the expected value of the relevant indicator function). This yielded the estimate 0.127, which compares well to the right answer.

Worked example 14.11 (Computing a Probability) Assume the random variable X is uniformly distributed in the range $[0-1]$, and the random variable Y is uniformly distributed in the range $[0-10]$. Write $Z = (Y - 5X)^3 - X^2$. What is $P(\{Z > 3\})$?

Solution With enough work, one could probably work this out in closed form. An easy program will get a good estimate. My program computed 1000 values of Z (by drawing X and Y from the appropriate random number generator, then evaluating the function) and counted the fraction of Z values that was greater than 3 (which is the relevant indicator function). For a run of my program, I got $P(\{Z > 3\}) \approx 0.619$

For all the examples we will deal with, producing an IID sample of the relevant probability distribution will be straightforward. You should be aware that it can be very hard to produce an IID sample from an arbitrary distribution, particularly if that distribution is over a continuous variable in high dimensions.

14.2.2 Simulation Results as Random Variables

The estimate of a probability or of an expectation that comes out of a simulation experiment is a random variable, because it is a function of random numbers. If you run the simulation again, you'll get a different value, unless you did something silly with the random number generator. Generally, you should expect this random variable to have a normal distribution. You can check this by constructing a histogram over a large number of runs. The mean of this random variable is the parameter you are trying to estimate. It is useful to know that this random variable tends to be normal, because it means the standard deviation of the random variable tells you a lot about the likely values you will observe.

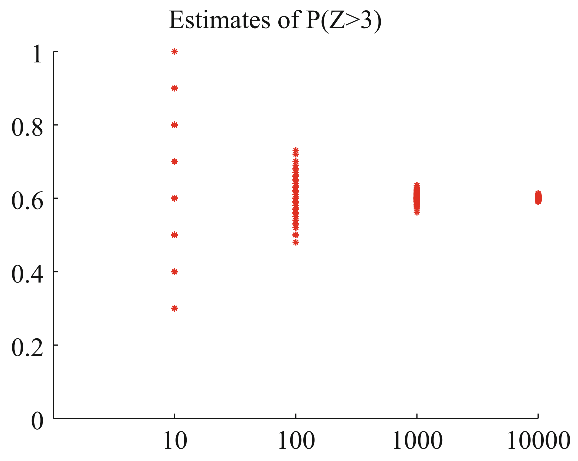

Fig. 14.6 Estimates of the probability from Example 14.11, obtained from different runs of my simulator using different numbers of samples. In each case, I used 100 runs; the number of samples is shown on the horizontal axis. You should notice that the estimate varies pretty widely when there are only 10 samples in each run, but the variance (equivalently, the size of the spread) goes down sharply as the number of samples per run increases to 1000. Because we expect these estimates to be roughly normally distributed, the variance gives a good idea of how accurate the original probability estimate is

Another helpful rule of thumb, which is almost always right, is that the standard deviation of this random variable behaves like

$$\frac{C}{\sqrt{N}}$$

where C is a constant that depends on the problem and can be very hard to evaluate, and N is the number of runs of the simulation. What this means is that if you want to (say) double the accuracy of your estimate of the probability or the expectation, you have to run four times as many simulations. Very accurate estimates are tough to get, because they require immense numbers of simulation runs.

Figure 14.6 shows how the result of a simulation behaves when the number of runs changes. I used the simulation of Example 14.11, and ran multiple experiments for each of a number of different samples (i.e. 100 experiments using 10 samples; 100 using 100 samples; and so on).

Worked example 14.12 (Getting 14's with 20-Sided Dice) You throw 3 fair 20-sided dice. Estimate the probability that the sum of the faces is 14 using a simulation. Use $N = [1e1, 1e2, 1e3, 1e4, 1e5, 1e6]$. Which estimate is likely to be more accurate, and why?

Solution You need a fairly fast computer, or this will take a long time. I ran ten versions of each experiment for $N = [1e1, 1e2, 1e3, 1e4, 1e5, 1e6]$, yielding ten probability estimates for each N. These were different for each version of the experiment, because the simulations are random. I got means of $[0, 0.0030, 0.0096, 0.0100, 0.0096, 0.0098]$, and standard deviations of [0 0.0067 0.0033 0.0009 0.0002 0.0001]. This suggests the true value is around 0.0098, and the estimate from $N = 1e6$ is best. The reason that the estimate with $N = 1e1$ is 0 is that the probability is very small, so you don't usually observe this case at all in only ten trials.

Small probabilities can be rather hard to estimate, as we would expect. In the case of Example 14.11, let us estimate $P(\{Z > 950\})$. A few moments with a computer will show that this probability is of the order of 1e-3 to 1e-4. I obtained a million different simulated values of Z from my program, and saw 310 where $Z > 950$. This means that to know this probability to, say, three digits of numerical accuracy might involve a daunting number of samples. Notice that this does not contradict the rule of thumb that the standard deviation of the random variable defined by a simulation estimate behaves like $\frac{C}{\sqrt{N}}$; it's just that in this case, C is very large indeed.

Useful Facts 14.4 (The Properties of Simulations)
You should remember that

- The weak law of large numbers means you can estimate expectations and probabilities with a simulation.
- The result of a simulation is usually a normal random variable.
- The expected value of this random variable is usually the true value of the expectation or probability you are trying to simulate.
- The standard deviation of this random variable is usually $\frac{C}{\sqrt{N}}$, where N is the number of examples in the simulation and C is a number usually too hard to estimate.

Worked example 14.13 (Comparing Simulation with Computation) You throw three fair six-sided dice. You wish to know the probability the sum is three. Compare the true value of this probability with estimates from six runs of a simulation using $N = 10{,}000$. What conclusions do you draw?

Solution I ran six simulations with $N = 10{,}000$, and got [0.0038, 0.0038, 0.0053, 0.0041, 0.0056, 0.0049]. The mean is 0.00458, and the standard deviation is 0.0007, which suggests the estimate isn't that great, but the right answer should be in the range $[0.00388, 0.00528]$ with probability about 0.68. The true value is $1/216 \approx 0.00463$. The estimate is tolerable, but not super accurate.

14.2.3 Simulating Markov Chains

We will always assume that we know the states and transition probabilities of the Markov chain. Properties that might be of interest in this case include: the probability of hitting an absorbing state; the expected time to go from one state to another; the expected time to hit an absorbing state; and which states have high probability under the stationary distribution.

Worked example 14.14 (Coin Flips with End Conditions) I flip a coin repeatedly until I encounter a sequence HTHT, at which point I stop. What is the probability that I flip the coin nine times?

Solution You might well be able to construct a closed form solution to this if you follow the details of example 14.13 and do quite a lot of extra work. A simulation is really straightforward to write; notice you can save time by not continuing to simulate coin flips once you've flipped past nine times. I got 0.0411 as the mean probability over 10 runs of a simulation of 1000 experiments each, with a standard deviation of 0.0056.

Worked example 14.15 (A Queue) A bus is supposed to arrive at a bus stop every hour for 10 h each day. The number of people who arrive to queue at the bus stop each hour has a Poisson distribution, with intensity 4. If the bus stops, everyone gets on the bus and the number of people in the queue becomes zero. However, with probability 0.1 the bus driver decides not to stop, in which case people decide to wait. If the queue is ever longer than 15, the waiting passengers will riot (and then immediately get dragged off by the police, so the queue length goes down to zero). What is the expected time between riots?

Solution I'm not sure whether one could come up with a closed form solution to this problem. A simulation is completely straightforward to write. I get a mean time of 441 h between riots, with a standard deviation of 391. It's interesting to play around with the parameters of this problem; a less conscientious bus driver, or a higher intensity arrival distribution, lead to much more regular riots.

Worked example 14.16 (Inventory) A store needs to control its stock of an item. It can order stocks on Friday evenings, which will be delivered on Monday mornings. The store is old-fashioned, and open only on weekdays. On each weekday, a random number of customers comes in to buy the item. This number has a Poisson distribution, with intensity 4. If the item is present, the customer buys it, and the store makes 100; otherwise, the customer leaves. Each evening at closing, the store loses 10 for each unsold item on its shelves. The store's supplier insists that it order a fixed number k of items (i.e. the store must order k items each week). The store opens on a Monday with 20 items on the shelf. What k should the store use to maximise profits?

Solution I'm not sure whether one could come up with a closed form solution to this problem, either. A simulation is completely straightforward to write. To choose k, you run the simulation with different k values to see what happens. I computed accumulated profits over 100 weeks for different k values, then ran the simulation five times to see which k was predicted. Results were 21, 19, 23, 20, 21. I'd choose 21 based on this information.

For Example 14.16, you should plot accumulated profits. If k is small, the store doesn't lose money by storing items, but it doesn't sell as much stuff as it could; if k is large, then it can fill any order but it loses money by having stock on the shelves. A little thought will convince you that k should be near 20, because that is the expected number of customers each week, so $k = 20$ means the store can expect to sell all its new stock. It may not be exactly 20, because it must depend a little on the balance between the profit in selling an item and the cost of storing it. For example, if the cost of storing items is very small compared to the profit, an very large k might be a good choice. If the cost of storage is sufficiently high, it might be better to never have anything on the shelves; this point explains the absence of small stores selling PC's.

Quite substantial examples are possible. The game "snakes and ladders" involves random walk on a Markov chain. If you don't know this game, look it up; it's sometimes called "chutes and ladders", and there is an excellent Wikipedia page. The state is given by where each players' token is on the board, so on a 10×10 board one player involves 100 states, two players 100^2 states, and so on. The set of states is finite, though big. Transitions are random, because each player throws dice. The snakes (resp. ladders) represent extra edges in the directed graph. Absorbing states occur when a player hits the top square. It is straightforward to compute the expected number of turns for a given number of players by simulation, for example. For one commercial version, the Wikipedia page gives the crucial numbers: for two players, the expected number of moves to a win is 47.76, and the first player wins with probability 0.509. Notice you might need to think a bit about how to write the program if there were, say, eight players on a 12×12 board—you would likely avoid storing the entire state space.

14.3 Example: Ranking the Web by Simulating a Markov Chain

Perhaps the most valuable technical question of the last 30 years has been: Which web pages are interesting? Some idea of the importance of this question is that it was only really asked about 20 years ago, and at least one gigantic technology company has been spawned by a partial answer. This answer, due to Larry Page and Sergey Brin, and widely known as PageRank, revolves around simulating the stationary distribution of a Markov chain.

You can think of the world wide web as a directed graph. Each page is a state. Directed edges from page to page represent links. Count only the first link from a page to another page. Some pages are linked, others are not. We want to know how important each page is.

One way to think about importance is to think about what a random web surfer would do. The surfer can either (a) choose one of the outgoing links on a page at random, and follow it or (b) type in the URL of a new page, and go to that instead. This is a random walk on a directed graph. We expect that this random surfer should see a lot of pages that have lots of incoming links from other pages that have lots of incoming links that (and so on). These pages are important, because lots of pages have linked to them.

For the moment, ignore the surfer's option to type in a URL. Write $r(i)$ for the importance of the i'th page. We model importance as leaking from page to page across outgoing links (the same way the surfer jumps). Page i receives importance down each incoming link. The amount of importance is proportional to the amount of importance at the other end of the link, and inversely proportional to the number of links leaving that page. So a page with only one outgoing link transfers all its importance down that link; and the way for a page to receive a lot of importance is for it to have a lot of important pages link to it alone. We write

$$r(j) = \sum_{i \to j} \frac{r(i)}{|i|}$$

where $|i|$ means the total number of links pointing *out* of page i. We can stack the $r(j)$ values into a *row* vector $\mathbf{r}$, and construct a matrix $\mathcal{P}$, where

$$p_{ij} = \begin{cases} \frac{1}{|i|} & \text{if } i \text{ points to } j \\ 0 & \text{otherwise} \end{cases}$$

With this notation, the importance vector has the property

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

and should look a bit like the stationary distribution of a random walk to you, except that $\mathcal{P}$ isn't stochastic—there may be some rows where the row sum of $\mathcal{P}$ is zero, because there are *no* outgoing links from that page. We can fix this easily by replacing each row that sums to zero with $(1/n)\mathbf{1}$, where n is the total number of pages. Call the resulting matrix $\mathcal{G}$ (it's quite often called the **raw Google matrix**).

The web has pages with no outgoing links (which we've dealt with), pages with no incoming links, and even pages with no links at all. A random walk could get trapped by moving to a page with no outgoing links. Allowing the surfer to randomly enter a URL sorts out all of these problems, because it inserts an edge of small weight from every node to every other node. Now the random walk cannot get trapped.

There are a variety of possible choices for the weight of these inserted edges. The original choice was to make each inserted edge have the same weight. Write $\mathbf{1}$ for the n dimensional column vector containing a 1 in each component, and let $0 < \alpha < 1$. We can write the matrix of transition probabilities as

$$\mathcal{G}(\alpha) = \alpha \frac{(\mathbf{1}\mathbf{1}^T)}{n} + (1 - \alpha)\mathcal{G}$$

where $\mathcal{G}$ is the original Google matrix. An alternative choice is to choose a weight for each web page. This weight could come from: a query; advertising revenues; thaumaturgy; blind prejudice; page visit statistics; other sources; or a mixture of all (Google keeps quiet about the details). Write this weight vector $\mathbf{v}$, and require that $\mathbf{1}^T\mathbf{v} = 1$ (i.e. the coefficients sum to one). Then we could have

$$\mathcal{G}(\alpha, \mathbf{v}) = \alpha \frac{(\mathbf{1}\mathbf{v}^T)}{n} + (1 - \alpha)\mathcal{G}.$$

Now the importance vector $\mathbf{r}$ is the (unique, though I won't prove this) *row* vector $\mathbf{r}$ such that

$$\mathbf{r} = \mathbf{r}\mathcal{G}(\alpha, \mathbf{v}).$$

How do we compute this vector? One natural algorithm is to estimate $\mathbf{r}$ with a random walk, because $\mathbf{r}$ is the stationary distribution of a Markov chain. If we simulate this walk for many steps, the probability that the simulation is in state j should be $r(j)$, at least approximately.

This simulation is easy to build. Imagine our random walking bug sits on a web page. At each time step, it transitions to a new page by either (a) picking from all existing pages at random, using $\mathbf{v}$ as a probability distribution on the pages (which it does with probability α); or (b) chooses one of the outgoing links uniformly and at random, and follows it (which it does with probability $1 - \alpha$). The stationary distribution of this random walk is $\mathbf{r}$. Another fact that I shall not prove is that, when α is sufficiently large, this random walk very quickly "forgets" it's initial distribution. As a result, you can estimate the importance of web pages by starting this random walk in a random location; letting it run for a bit; then stopping it, and collecting the page you stopped on. The pages you see like this are independent, identically distributed samples from $\mathbf{r}$; so the ones you see more often are more important, and the ones you see less often are less important.

14.4 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 occasionally rhyming with “loon”. 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.

14.4.1 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 $\boldsymbol{\pi}$. 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, $(\mathcal{P}, \mathcal{Q}, \boldsymbol{\pi})$. 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 Sect. 14.1.3. 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 $\boldsymbol{\pi}$, 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.

14.4.2 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 to recover a sequence X_i that maximises

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

which is

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

and this is

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

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 $\log P(X_1, X_2, \ldots, X_N, Y_1, Y_2, \ldots, Y_N|\mathcal{P}, \mathcal{Q}, \boldsymbol{\pi})$ in a very useful way, because the X_u form a Markov chain. We want to maximise

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

but this is

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

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 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 Fig. 14.7)

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 want to find the best path from each node in the first column to each 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 Fig. 14.8). 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 Fig. 14.8). 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

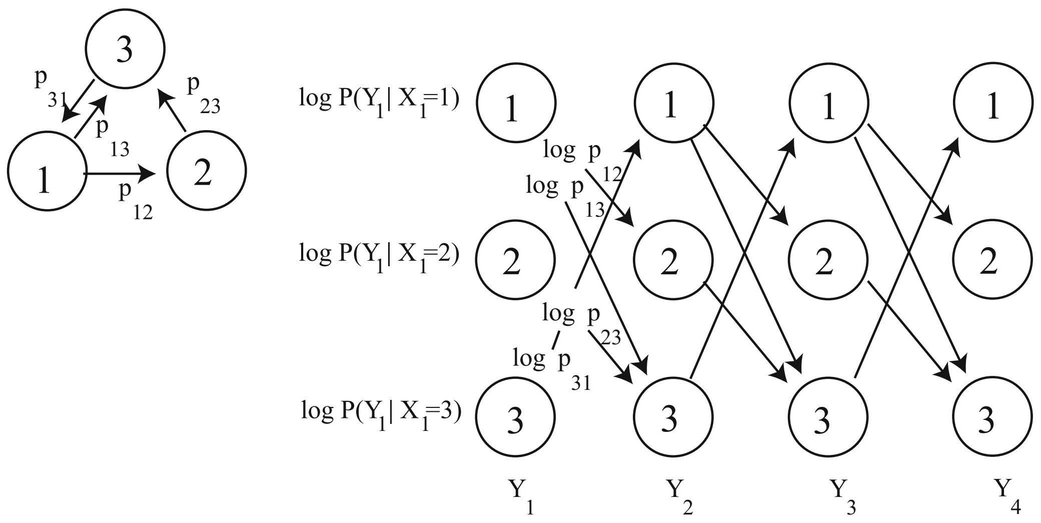

Fig. 14.7 At the *top left*, a simple state transition model. Each outgoing edge has some probability, though the topology of the model forces two of these probabilities to be 1. Below, the trellis corresponding to that model. Each path through the trellis corresponds to a legal sequence of states, for 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

node. This is much more easily done than described. All this works just as well for the fourth-last column, etc. (panel III in Fig. 14.8) 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 Fig. 14.8). Finally, to find the value of the best path leaving the first column, we compute the maximum value over all nodes in the first column.

We can also get the path with the maximum 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 Fig. 14.8.

14.4.3 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

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

and

$$B_{N-1}(j) = \underset{u}{\text{argmax}} \left[\log P(X_N = u|X_{N-1} = j) \right.$$
$$\left. + \log P(Y_N|X_N = u)\right].$$

You should check this against step I of Fig. 14.8

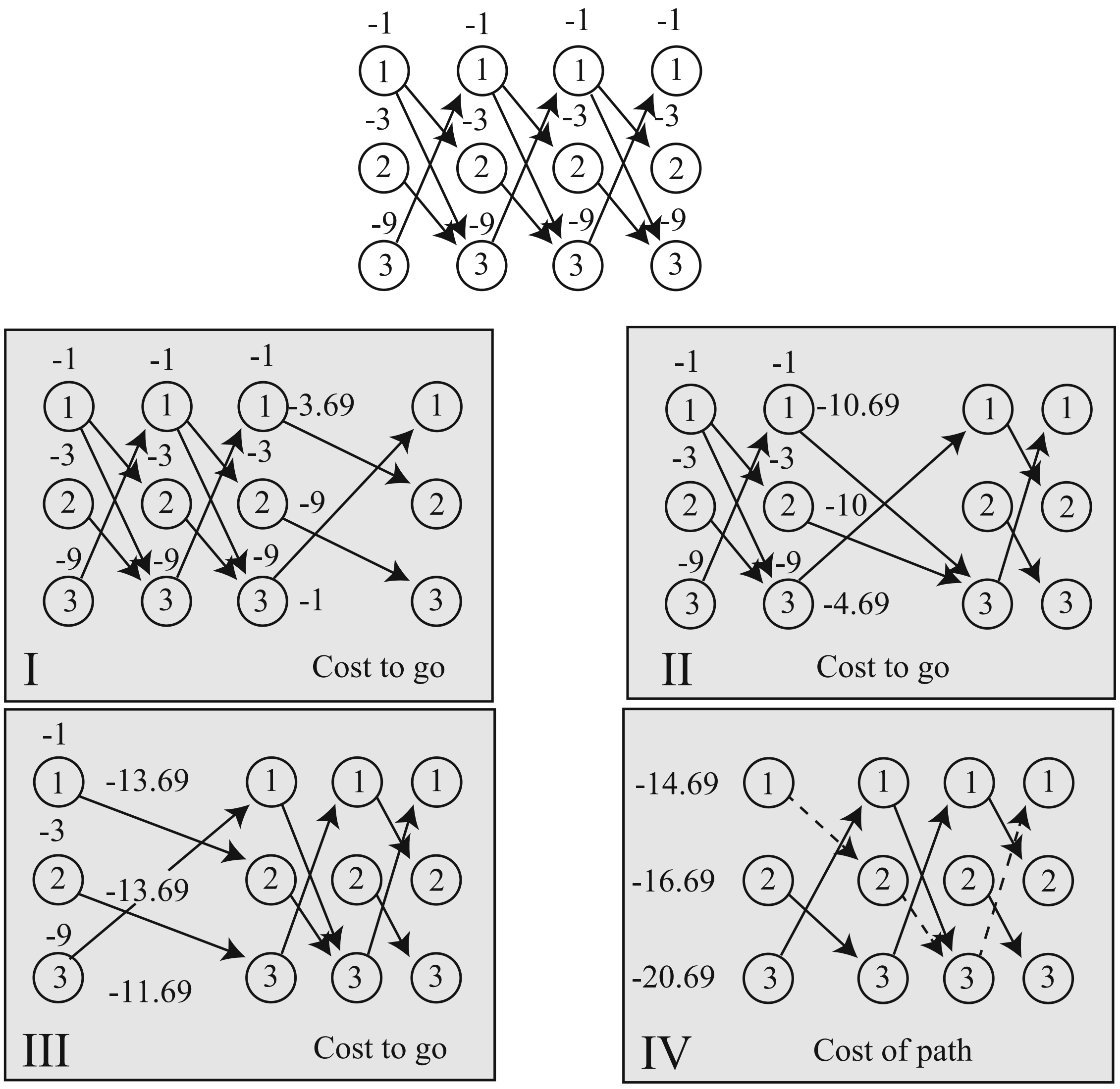

Fig. 14.8 An example of finding the best path through a trellis. The probabilities of leaving a node are uniform (and remember, $\ln 2 \approx -0.69$). Details in the text

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

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

and

$$B_w(j) = \underset{u}{\text{argmax}} \, [\log P(X_{w+1} = u | X_w = j) + \log P(Y_{w+1} | X_{w+1} = u) + C_{w+1}(u)] \,.$$

Check this against steps II and III in Fig. 14.8.

14.4.4 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 (Tables 14.1, 14.2 and 14.3 show some uni-, bi- and trigram probabilities).

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

Table 14.1 The most common single letters (unigrams) that I counted from a draft of this chapter, with their probabilities

*	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

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)

Table 14.2 The most common bigrams that I counted from a draft of this chapter, with their probabilities

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)

The '*' stands for a space. For each of the ten 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

Table 14.3 The most frequent ten trigrams in a draft of this chapter, with their probabilities

*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

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 (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

> the trellis 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").

Remember this: *A hidden Markov model can be used to model many sequences. Observations are noisy versions of underlying hidden states, and the states form a Markov chain. One can infer the hidden states from the observations with dynamic programming. This approach applies very widely, and is extremely useful in practice.*

14.5 You Should

14.5.1 Remember These Definitions

14.5.2 Remember These Terms

Markov chain	331
transition probabilities	331
biased random walk	331
absorbing state	332
recurrent	332
stochastic matrices	333
irreducible	335
stationary distribution	335
unigrams	337
unigram models	337
bigrams	337
bigram models	337
trigrams	337

14.5.3 Remember These Facts

14.5.4 Be Able to

- Estimate various probabilities and expectations for a Markov chain by simulation.
- Evaluate the results of multiple runs of a simple simulation.
- Set up a simple HMM and use it to solve problems.

Problems

14.1 Multiple die rolls: You roll a fair die until you see a five, then a six; 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 five ways not to get a five, 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 $P(N) = (5/6)P(N-1) + (25/36)P(N-2)$.

14.2 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 Σ_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:
  a. $TT\Sigma_{N-2}$
  b. $HH\Sigma_{N-3}$
  c. $THH\Sigma_{N-2}$
  d. $HTT\Sigma_{N-3}$
(c) Now use this argument to show that $P(N) = (1/2)P(N-2) + (1/4)P(N-3)$.

14.3 For the umbrella example of Worked example 14.2, 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

14.4 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). The gambler starts by choosing a die. Choosing a die is by flipping a coin; if the coin comes up heads, the gambler chooses the fair die, otherwise, the unfair die. The gambler rolls the chosen die repeatedly until a six comes up. When a six appears, the gambler chooses again (by flipping a coin, etc), and continues.

(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. What is the probability the emitted symbol is 1?
(b) Use your simulation to produce 10 sequences of 100 symbols. 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?
(c) Does inference accuracy improve when you use sequences of 1000 symbols?

14.5 **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 ϵ and split this between all events with zero count. Your models should differ only by the size of ϵ.
(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 ϵ. Keep in mind that the corrected text could be worse than the corrupted text.