Chapter 7

Discrete Dynamic Patterns

Consider, first, the spelling checker. It consists of a large dictionary of words in a particular language, and it is useful for compensating for the deficiencies of a progressive education, since if you have written a page of text the chances of a spelling error are quite high, and we all wish to conceal our intellectual inadequacies. It also has use when trying to do OCR, since if we are uncertain as to whether a given character is an 's' or an '5', we can look at the surrounding letters and use the fact, for example, that 'Profes5or' is not a word but 'Professor' is. When we do this we are using syntactic information to resolve an uncertainty. We are *smoothing* the word, filtering the string of characters, somewhat as we might use a moving average filter to use the neighbouring values in a time series to modify a particularly egregious value. It is plain to everyone who has tried to decipher bad handwriting that such methods are applied a good deal in human information processing.

We shall be concerned in this chapter with long strings of symbols, short strings of symbols, and what are known as *formal languages*, which is to say sets of strings of symbols. We do not insist that the language arise from people speaking or writing, although these do give particular cases, but we examine the subject in vastly more generality. We shall also be concerned with *stochastic languages*, briefly mentioned in the last chapter, where there is, for each string in the language, some number saying how often it comes up.

The reflective reader will, of course, be wondering why we devote a whole chapter to such matters. Having focussed to a large extent of the recognition of characters in discussing applications, the reader *might* think that this is the important connection with Pattern Recognition, since strings of characters form words, strings of words form sentences and sentences are (or aren't) grammatical. Anyway, there is presumably *some* connection between formal language and natural language, and characters come into natural language. Thus the reader seeking for connections between things, that is to say the reader who is thinking about what he reads, might hazard a guess that we are going to be concerned with using a spelling checker to improve the chances of a correct reading of some characters. He might also guess that there is a connection with Speech Recognition, since our first move there, when discussing the standard approach, was to reduce the continuous trajectories in $\mathbb{R}^n$ to trajectories in a discrete space, that is to say, symbol strings. So maybe we are going to enlarge our perspectives on Speech Recognition on the side.

It is true that one aim of this chapter is to show how to extract information about which strings of symbols occur and which don't in a page of text so as to improve recognition of characters, but this is the tip of a rather humongous iceberg. The problem in the case of strings of symbols is fairly easy to understand, but the ideas go across to cases which are on the face of things very different. For example, I shall explain in the next chapter how to clean up an image by using knowledge drawn from other images. This is a clever trick when human beings do it, and is known to psychologists as *Transfer of Learning*, and making a program do it is even cleverer. It is essentially a rather considerable generalisation of using a dictionary to resolve uncertainty or correct a preliminary decision at a lower level, as in 'Profes5or'. The generalisation is so extreme as to require a good deal of preliminary technicality however, and this is one purpose of the present chapter.

We observe that our low level objects, characters in the above example, are combined together to form higher level objects, words in the example. You would not be the first person to notice that it doesn't stop there; words are aggregated into phrases, phrases into sentences, sentences into paragraphs, and so on. There are hierarchies out there. The idea of a 'high level object' is rather an interesting idea to try to make precise. Cast your mind back to when you learnt to drive a car. As a rank tiro, you may have had to think quite hard when the instructor gave the order 'turn left'. It requires slowing down, i.e. taking your right foot up a bit, signalling left, looking in the mirror to ensure that the driver of the car behind knows you are slowing, disengaging the gears by depressing the clutch (push your left foot down), and moving the gear lever from where it is to somewhere else (where, for God's sake?). In addition you have to keep an eye open to ensure that the turn is not prohibited by some road signs and that you won't run over any little old ladies. Oh, and you'd better remember to turn the steering wheel as well. Seen from this point of view, turning left is a pretty complicated business. Actually, the muscle contractions required to depress the clutch and reduce the pressure on the accelerator are also pretty complicated and you took years to learn to do them; most of the time between birth and three years of age in fact. And now you plan a drive of a hundred miles without, I hope, a trace of panic. It is clear that there is a chunking of low level actions consisting of muscle fibres firing, into larger things like lifting a foot, chunking of actions like lifting feet, pushing legs, turning the wheel and so on into 'turning left', and a further chunking of turns and stops and starts in order to get you from A to B. Although why anybody should want to get to B is far from clear to anybody who, like myself, has never visited A.

Now the intuitive notion of chunking low level things into higher level things, and so on, and the corresponding decomposition of some things into sub-things occurs all over the place. If you are obliged to write a computer program for example, then you are asked to write a string of symbols which defines a map. The program will have some input, possibly several different sorts, and some output. You therefore are faced with the problem of constructing this map using the more primitive maps given in the language such as '+', '*', '&', 'if' and so on. You also have standard ways to glue these primitive maps together. It is good programming practice to write your program as a set of procedures, and a procedure is a sub-map. A well written program is likely to have elements of a hierarchical structure to it, in other words it is a chunk of chunks of... of the given primitives. This is more obvious in the so called applicative or functional languages (the French word for map is *application*), but it holds in all of them. So again, chunking occurs, and the rather woolly idea of 'high level' objects and lower level objects makes sense. And as observed, chunking occurs in natural language. In the case of a printed book, we may consider at the lowest level the pixels of ink on paper, then the edges which are defined by the pixels, then the strokes made up of edges, then the characters built up from strokes, then the words built up from characters, the phrases built up from words, the sentences from phrases, the paragraphs from sentences, the chapters from paragraphs, and finally the book from chapters. And you could, in good conscience, insert some intermediate levels if you wished.

The intuitive notions are quite compelling and are even well known to psychologists, but the Mathematician is impelled to ask how to define the terms properly. Talking about things with no very clear meaning does very well in the Soggy Sciences, but it doesn't cut the mustard in real science or in engineering, and I have no wish to be mistaken for a psychologist as the term is currently widely understood, anymore than anyone in a Chemistry department might wish to be taken for an alchemist. Nevertheless, it has to be admitted that what we are studying was identified by psychologists first as an important aspect of cognitive processing in human beings. Our job will be to make the ideas precise. This will involve us in defining syntactic structures of a continuous sort, and it would be a good idea to get the standard ideas clarified in our minds first. The really interesting parts of this chapter therefore will be the sections that have to do with being able to define clearly what is meant by a chunk, for these have extensive applications to all sorts of pattern recognition. If you want to describe the shape of a set of points in $\mathbb{R}^n$ it may be convenient to find a description of parts of the set and the relationship between the parts, and here it pays to know what you mean by the terms.

It is important to clarify ideas about structured objects with substructures, and the case of a sequential or temporal relation between the substructures is the easiest to deal with, so it makes sense to tackle this case first. This chapter is therefore a bridge between classical pattern recognition and the modern syntactic approach.

In the hope, then, that your interest has been aroused enough to carry you through the forest of definitions which are coming up, we proceed to investigate formal languages and streams of symbols.

7.1 Alphabets, Languages and Grammars

7.1.1 Definitions and Examples

Definition 7.1 *An* alphabet *is a set of objects called* symbols*.*

Alphabets are usually finite, but I shall not so limit myself. Think of the ASCII character set to make your ideas concrete, but watch out for more bizarre examples coming up.

Definition 7.2 *A* string *on an alphabet is a finite sequence of symbols from the alphabet.*

Definition 7.3 *A* language *over an alphabet A is a set of strings on A.*

Languages may be finite, but these aren't the interesting ones. It is usual to define A^* to be the set of all possible strings over A, which makes A^* a language. This allows us to define a *language over A* as a subset of A^*.

Example 7.1 *If A is the set of ASCII characters, the set of correctly spelled English words is a language. The set of syntactically correct PASCAL programs is another. The set of English sentences is a third.*

Example 7.2 *If A is the set of words in the Oxford English Dictionary, then the set of English sentences is a language over A. It is a different language from the same set of sentences over the ASCII character set.*

Definition 7.4 *A* recognition grammar for a language $\mathcal{L}$ over A *is an algorithm which when given a string on A will output either a 1 or a 0, and will output a 1 whenever the string is in* $\mathcal{L}$ *and a 0 when it isn't.*

I shall not define an *algorithm* here, you can think of a computer program of some sort if you want to make it reasonably definite.

The definition I give is not quite standard, because the usual next move in the literature is to go to some kinds of restrictions on the algorithms which might be employed and I do not wish to consider the usual class of restrictions. I shall discuss this further below.

Definition 7.5 *A* generative grammar *for a language* $\mathcal{L}$ *over A is an algorithm which has no input but produces strings on A such that only strings in* $\mathcal{L}$ *are produced and every string is produced eventually.*

If the alphabet is finite, A* is countable, so generative grammars may exist. Indeed it is hard at first glance to see how they might not, since how can I tell you what $\mathcal{L}$ *is*, except by specifying a generative grammar? You may wish to brood upon this point.

Definition 7.6 *A* stochastic language over A *is a language over A together with a real number in [0,1]* $p(w)$ *assigned to each string* w*, so that the limit of the sums of these numbers over the entire language exists and is 1.*

Example 7.3 *Let A consist of the set a,b and $\mathcal{L}$ be the set*

$$\{aba, abba, ...ab^n a, ..\}$$

for every positive integer n. Let the number $p(ab^n a)$ assigned to the string $ab^n a$ be $\frac{1}{2^n}$.

Example 7.4 *Let A be the ASCII character set, $\mathcal{L}$ the set of English words in the Oxford English dictionary, and for $w \in \mathcal{L}, p(w)$ is defined to be the relative frequency of occurence of the word in the present book.*

Definition 7.7 *A* stochastic recognition grammar *for a stochastic language $\mathcal{L}$ over A is an algorithm which when given a string w on A returns $p(w)$.*

Definition 7.8 *A stochastic generative grammar for a stochastic language $\mathcal{L}$ over A is an algorithm with no input which produces strings on A so that the relative frequency of the string w in a sample of size N strings produced by the algorithm, tends to $p(w)$ as N gets larger.*

Example 7.5 *Suppose we are given the finite state diagram of $fig.$7.1 and a pseudo-random number generator. The algorithm traces a path through the diagram from $<$ to $>$ in order to generate a string, and whenever it makes a jump from one state to another, or back to itself via a self loop, it produces the symbol attached to the transition arc. As soon as it gets to $>$ it starts over again to get a new string. The algorithm decides whether to go by one arc or another out of a state by running its random number generator and deciding in accord with the probabilities marked on the arcs out of a state. Thus if there is only one arc out of a node or state it moves along the single arc, while if there are two with equal probabilities* 0.5 *as in the diagram, it does the pseudo-random number equivalent of tossing a coin. It is easy to persuade oneself that it produces only strings $ab^n a$ for positive integer n, and that the frequencies of these strings will tend to $p(ab^n a) = \frac{1}{2^n}$, given only moderate optimism about the pseudo-random number generator's capacity to reliably simulate a 0.5 probability choice.*

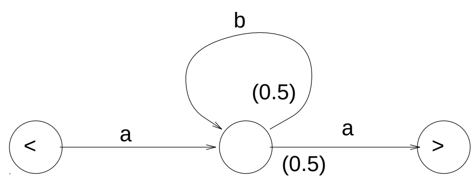

Figure 7.1: A Markov Process for generating a stochastic language.

Definition 7.9 *A finite network of nodes and arcs, with probabilities attached and symbols attached to the arcs, such as that illustrated in the last example, is known as a* first order markov process *for generating the stochastic language. If we take the probabilities away and forget about frequency counts, the diagram is known as a* finite state grammar *or* regular automaton *for the language.*

Clearly, some languages can have finite state generative grammars for them and others may not have such grammars. The set of *finite state languages*, otherwise known in some quarters as the *regular languages* are those for which a finite grammar can be found. The set is widely regarded as an important subset of all possible languages, and deciding whether a language is regular or not is a problem of some interest. It is trivial that all finite languages are finite state, but the converse is obviously false.

The distinction between generative grammars and recognition grammars can be thought to be important or regarded as an inadequacy of the language and a sign of a poorly constructed framework for talking about things. The distinction is nugatory for stochastic grammars for languages over a finite alphabet. If I have a recognition grammar for such a language, then I merely need to produce all the strings on the alphabet in some order and then test to see if the string is in the language. If it is, I put it in the box $\mathcal{L}$ and I have generated a string of the language. This clearly works for the stochastic and non-stochastic cases. Conversely, if I have a generative grammar and you give me a string w and want to know if it is in $\mathcal{L}$, I need to generate the strings until I can match the given string or fail to match it. But a failure to match it in any finite number of moves is inconclusive for the non-stochastic case. Whereas in the stochastic case I can obtain estimates for $p(w)$ by generating sequences of strings and counting occurrences of w.

Of course, one might doubt whether this is an algorithm for assigning a probability or relative frequency of occurrence to a string; it only yields estimates based on finite samples. A platonist philosopher or even a logician might argue that only production of the whole infinite set would yield the procedure, and this cannot be accomplished by a real program running on a real machine. Much depends here on what one accepts as an algorithm, and also on what one regards as an algorithm producing a real number measuring a relative frequency of occurrence. I apply a commonsense test of meaning here. For my purposes, a program which assures me that the probability of an event is 0.3333 is (a) feasible to write and (b) may be useful to run. Likewise, being told that the probability of occurrence of a string is less than 0.01 is helpful, whereas being told that I haven't got a stochastic recognition grammar unless presentation of a string of symbols yields an infinite decimal string, specifying the probability, would be bad for my blood pressure. My choices are either to specify what I mean very carefully indeed, as I should have to if I were writing a program for a computer or a text for Pure Mathematicians, or rely on the maturity and common sense of the reader to work out what I intend for practical purposes, which is quicker but a little risky. I choose to take a chance. I may regret this later.

By the *trace* of an algorithm I mean the sequence of operations it performs on a particular input.

Definition 7.10 *The trace of a recognition grammar for a particular string is called a* parse *of the string.*

7.1.2 ReWrite Grammars

Noam Chomsky is a conspiracy theorist who used to be a theoretical linguist; the Chomsky Hierarchy refers to language types rather than degrees of authority. Chomsky's work in formal language theory has been described by some as 'seminal' [1].

At the bottom of the ladder sit the finite state languages. If we want to we can think of them as being the languages we get from hopping about a finite state diagram, or we can cast them into ReWrite form as in the following example:

The language $\{ab^na : n \in \mathbb{N}\}$ can be represented by the finite state diagram pf fig.7.1; it can be expressed rather more elliptically as $\{aa, aba, abba, abbba, abbbba, \cdots\}$. Now we use three alphabets to describe it, and ReWrite rules. The first alphabet has only one symbol in it, S (which some say stands for 'sentence', and others for 'start'), and you are allowed to ReWrite it to the string ABA on a quite different alphabet containing two symbols, A, B. We write this in the form:

$$S \longrightarrow ABA$$

The symbols A and B can be rewritten into the last alphabet, the symbols of the alphabet of the language for which we are trying to obtain a grammar, the alphabet $\{a, b\}$. The allowed ways of doing this, together with the above ReWrite for completeness sake, are:

$$S \longrightarrow ABA$$

[1] Others have been even ruder.

$$\begin{aligned} A &\longrightarrow a \\ B &\longrightarrow b \\ B &\longrightarrow bB \end{aligned}$$

It is easy to see that if you simply follow the rules which are to start with an S and rewrite the strings you get by rewriting the symbols in them and stop only when there is nothing more to do, then you must generate one of the strings of the given language. Also, every string of the language can be obtained in this way.

It is also easy to see that you could make this into a stochastic grammar for the stochastic language generated with the probabilities given by fig.7.1 by putting probabilities on which rule you choose when there is a choice of rewriting a particular symbol.

With only a little thought, you can convince yourself that any finite state diagram can be turned into a ReWrite grammar. The idea is simply to label the nodes of the diagram with letters from a new alphabet, and write

$$S \longrightarrow xY$$

whenever you can get from the start of the diagram to the node Y by emmitting the symbol x, and then to continue similarly for all the other nodes. We may conveniently use a blank for the symbol for the terminating node or nodes if we have such things.

Conversely, if the ReWrite rules all look like

$$X \longrightarrow xY$$

where X and Y may be the same, and Y may empty, when it represents the terminating node of the diagram, then it is easy to build a finite state diagram of nodes and arrows which produces the same output. Clearly, some compression of this description is allowable; we didn't really need the A symbol in the example at all, and x may be a string of terminal symbols, that is, symbols of the alphabet of the language we actually care about.

In general we may define a ReWrite grammar as follows:

Definition 7.11 *A* ReWrite Grammar *for a language $\mathcal{L}$ is a finite sequence of alphabets of distinct symbols where the last alphabet is the alphabet of the language $\mathcal{L}$, together with a finite set of* productions *or* rules *of the form*

$$P \longrightarrow Q$$

where P and Q are strings of symbols from any of the alphabets. A derivation *of a string ℓ of $\mathcal{L}$ is a sequence of productions and a sequence of strings, such that the first element of the sequence of strings is a single symbol from the first alphabet, the last string is precisely ℓ, and the move from one string to its succesor is sanctioned by the corresponding rewrite rule: the k^{th} string contains a substring P where the $(k+1)^{th}$ string contains a substring Q, and the left hand side of the k^{th} production also contains P where the right hand side contains Q, and where the left hand side of the production is a substring of the k^{th} string.*

The alphabets other than the last are called the intermediate alphabets *and any symbol from them is called an* intermediate symbol. *The last alphabet is called the alphabet of* terminal symbols*, the first alphabet the alphabet of* initial symbols. *Then we require that every string of the language can be produced by a derivation, and no strings not in the language can be so derived.*

This is a little untidy: it suffices to have only one initial symbol, by introducing an eaven earlier alphabet if necessary, together with the obvious productions to get from my new initial symbol to your old initial symbols. Similarly, we can get by with amalgamating all the intermediate alphabets into just one. Why bother to make the distinction? Ah, well, there may be reasons coming along shortly.

One of the first restrictions which we could make on the ReWrite Grammars defined above is to stipulate that you can only go *down* the sequence of alphabets, that is to say, when we have a production

$$P \longrightarrow Q$$

Q is not allowed to contain symbols from earlier alphabets than P. If you reflect on natural language, where you can ReWrite the S symbol as, perhaps, $NP * VP * NP$ where NP is a new symbol (standing for Noun Phrase) and the sequence of ReWrites which can go on to generate a sequence of ASCII characters constituting a sentence in the English language, you will perhaps find this stipulation attractive. It gives some kind of hierarchical structure to the system.

The most extreme restriction of this kind we might make is that the left hand side of each production is allowed to consist of a single symbol at any level, and the right hand side of a string of symbols at the level one lower. This gives only finite languages such as English. It doesn't do a whole lot for English either. [2] Even a superficial familiarity with the structure of English or any other natural language leaves one feeling that grammars such as this don't get us too far.

An alternative restriction which might be imposed would be to allow the right hand side of a production to contain a string of symbols from the next alphabet lower down followed by an optional symbol from the same level as the left hand side, and to stipulate still only single symbols on the left. This gives us, by the above argument, the finite state languages. This is a weaker restriction, so we get a larger class of languages which can be obtained by this class of ReWrite rules.

To weaken things even further, suppose that you ensure that the left

[2] Linguists have argued that English is not *really* finite, since you can self embed parenthetic clauses as deep as you like. Thus we have such constructs as the old lady who swallowed a cow in order to catch the long list of other things she swallowed in order to catch the spider which she swallowed to catch the fly she started off with. Similarly for houses that Jack built, and those sentences of German which end with a barrage of verbs. I have found while talking, something I do a lot, to students, and others, that the deepest (and I mean deepest!) level of recursion to which it is practicable (in the sense of keeping the attention of the student, or whoever) to go, is about three. I also find sentences of more than three million words tiresome. Sentences of more than thirty billion words would find no human being living long enough to read them or remember what the hell the beginning was about. It seems safe to conclude that English is actually finite, although the simplification of making it infinite *might* be worthwhile. But to do so in order to accommodate nursery rhymes seems excessive.

hand side is a single symbol, from some level alphabet, and the right hand side is some mix of symbols from the same level or the next lower level alphabet. I shall call such a grammar a *Context Free* (ReWrite) grammar. If a language may be obtained from such a grammar but not from a finitie state grammar, it is called a *context free language*.

As an example, consider the language $\mathcal{L}_{cf}$ defined by

$$\mathcal{L}_{cf} = \{a^n b a^n : n \in \mathbb{N}\}$$

Now it is not too hard to see that this is *not* a finite state language, that is to say there is no finite diagram such that running round it will generate precisely this set of strings.

To convince yourself of this, observe that if you run around a finite state diagram which has no loops in it, then there is some longest string which can be obtained, and all other strings are the same length or shorter. So there can only be a finite number of them. It follows that if the language has an infinite number of strings and is finite state, then the finite state diagram which generates the language must have at least one loop. Extending the argument a little, given such a diagram, there is a longest path (at least one) which does not pass around a loop, and any longer path must go around some loop at least once. So if you have an infinite language which is regular, there is some string in it with the property that generating the string involved going at least once around a loop, so that some of the string represents going around the loop, the bit before (if any) represents getting to the start of the loop, and the part after was obtained by going home after completing the loop. Now if you went around once, you could have gone around twice, three times or any other number of times. Or to say it in algebra, there is for any finite state language which is infinite, a string ℓ in that language having a substring p in it, so that $\ell = apb$ for other strings a, b (possibly empty) and such that $ap^n b$ is also always in the language, for any $n \in \mathbb{Z}^+$. This result is called the *pumping lemma*, posibly because you get a heart attack if you run around the same loop often enough.

Now apply it to the language $\mathcal{L}_{cf}$. There is, if the language is finite state, some (possibly longish) string in it and a substring ℓ representing an arbitrarily repeatable loop. If this substring ℓ contains the b, then there would have to be strings which contain lots of bs in the language, and there aren't. So it must consist entirely of as on the left of the b, or some string of as on the right of the b. Either way this leads to strings $a^n b a^m$ for $n \neq m$, which do not in fact occur. So the language is not finite state.

It can however easily be obtained by the following ReWrite system

$$\begin{array}{lcl} S & \longrightarrow & B \\ B & \longrightarrow & b \\ B & \longrightarrow & aBa \end{array}$$

among others. This is evidently context-free.

Context Free languages can also be described by graphical means; the diagrams between pp 116 and 118 in the *PASCAL User Manual and Report* by Kathleen Jensen and Niklaus Wirth, Springer Verlag, 1975 follow a Backus-Naur formalism (Rewrite Grammar) description of the syntax of the programming language PASCAL. It is easy to see that we could put all the labelled boxes together into one big diagram. They do not, however, constitute an automaton or finite state diagram (or finite state machine) because a subdiagram consisting of a box called *factor* can contain inside itself a sequence consisting of a term 'not' followed by the box *factor*. This allows for recursion. The PASCAL diagrams have symbols associated with nodes rather than arcs, but it is easy to persuade oneself that this is a minor difference of convention, and that if I have a finite state diagram where the symbols are emitted at nodes, you can easily construct one in which symbols are emitted along arcs so that both systems produce the same strings, and *vice versa*. Similarly with the diagrams for PASCAL, which determine what is called a *Push Down Stack Automaton*. To represent such a machine we may need to have diagrams containing little boxes with a suitable symbolic label so that a path can take one into and thence out of each such box, and to also have a separate diagram telling you about what is inside the box. What may be inside the box is a path from input to output which passes through other such boxes or even the same box again. It is not too hard to prove that such automata can generate and recognise any Context Free Language and that all languages generated and recognised (or *accepted*) by such automata are context free. This allows for an extension of the pumping lemma for CF languages; the proofs of all these claims are left to the reader to work out, or he can cheat by examining the standard books on the subject. Working out the proper definitions of the terms is probably the only bit that is much fun, and it would be wrong of me to take this pleasure from him.

The next level of generality in the Chomsky Hierarchy was to simply ensure that the right hand side of any production had to contain at least as many terminal symbols as the left hand side. Such grammars are called *Context Sensitive*, as are the languages which can only be obtained from such grammars. (It is easy to construct a context sensitive grammar for a finite state language, merely unnecessary).

And finally, the Unrestricted ReWrite Systems allow anything whatever. It is a fact that any language which can be generated by a Turing Machine can be generated by some ReWrite system, and *vice versa*, and Church's Thesis is the claim that anything that is effectively computable can be computed by a Turing Machine. So if you are willing to buy Church's Thesis, any algorithm whatever for generating strings can be realised by a ReWrite system if you feel so inclined. This explains why I use the term 'algorithm' in the definitions at the beginning of the chapter. [3]

Computer Scientists are familiar with the Backus-Naur Formalism, much used for specifying the structure of Computer Languages, as a slightly disguised version of the Rewrite form of a grammar; the Metamathematicians got to it first. The principal applications of formal language theory of this sort are (a) in designing Compilers, where there is

[3] Of course there are people who will want every term defined short of circularity, but algorithms for most of us are like elephants. We can't give satisfactory definitions of them , but we can recognise them when we see one.

virtually no application of a body of theory but some value in the discipline of thought forced by the terminology, and (b) keeping a certain sort of Pure Mathematician off the streets.[4]

7.1.3 Grammatical Inference

It is a very nice thing to have a grammar for a language, and represents a nice compact representation of the language. It is a *model* for the language in a sense which you should recognise as congenial to a follower of Rissanen. The question is, how are such things to be obtained?

Definition 7.12 *The* Grammatical Inference Problem *is, given a language $\mathcal{L}$ over an alphabet A, to obtain a grammar for it. This holds for both stochastic and non-stochastic languages; for a stochastic language one seeks a stochastic grammar.*

In real life, one is actually given a sample of strings produced by some process and the sample is finite. The real life problem is to model the process economically. Thus the next example is a sensible problem which can be made absurd with no trouble at all:

Example 7.6 *You are given what are claimed to be 15 strings produced by a stochastic process of an unknown sort. The strings are $aba(8)$, $abba(4)$, $abbba(2)$ and $abbbba(1)$, where the numbers in parentheses give the counts of the string. You are required to model the process and use your model to obtain an estimate of the probability of seeing $abbbbbba$.*

Without some extra constraints such as a class of models from which to pick, the problem has no unique solution. For example, you might model the data by saying these are the only strings there are. This is

[4]This is important. The damage those guys could do if let loose in business or politics doesn't bear thinking about.

just the SureThing or Predestination model again. Your estimate of the probability of getting *abbbbbba* is that it is zero. This is very incautious of you, since if it occurs your model will get an infinite amount of 'surprise', the shock of which might disable you. On the other hand, you might assign a probability of $\frac{1}{2^n}$ for getting $ab^n a$, and of zero for getting any other string. This would also be incautious, since if the next string were *abab* you would get infinite surprise. On the other hand, my instincts and intuitions tell me that I would sooner chance getting *abab* than *abbbbbba*, which looks to me much more like *abbbba*, which I have seen.

This is telling us that my own personal grammatical inference system has more similarities to a model with $p(ab^n a) = \frac{1}{2^n}$ for n a positive integer than it does to the finite model with $p(aba) = \frac{8}{15}$, $p(abba) = \frac{4}{15}$, $p(abbba) = \frac{2}{15}$, and $p(abba) = \frac{1}{15}$. And I *do* have some sort of personal grammatical inference system, because I do have different expectations about getting other strings from the same source as produced the 15 samples. The question of what kind of expectations I in fact have, and how I obtain them from contemplating the strings would appear to be a study of my personal psychology; not, on the face of things, a very interesting object of thought. If it should turn out that your expectations are very similar to mine and may have been obtained in the same way, then the subject immediately becomes more interesting. And if it should turn out that the kinds of expectations about what can occur in future for sequences of events of a kind such as might interest a cockroach are also obtained by a similar process, then that process becomes very interesting indeed. Now even a cockroach experiences a sequence of things happening to it, and has an interest in predicting the next one or two. Cockroaches, in other words, have to perform grammatical inference too, just like you and me. It seems possible then that we might all use similar methods, which means that in examining these methods we should be saying something about how brains use data to obtain expectations of what might happen next. This hypothesis might seem wildly implausible, but it is (a) economical and (b) susceptible to some kind of analysis and experimentation. It is therefore defensible as a first attempt at making sense of the phenomenon of learning as carried out by brains.

Definition 7.13 *A* local (stochastic) grammar of order k *for a language* $\mathcal{L}$ *is a set of* k*grams, i.e. strings of length* k*, all of which are substrings of strings in* $\mathcal{L}$*, and such that every substring of length* k *of a string of* $\mathcal{L}$ *is in the set of* k*grams. If the language is stochastic, each* k*gram is accompanied by its relative frequency in the language among those* k*grams having the first* $k-1$ *symbols the same. For strings of length less than* k*, the whole string is stored, with its relative frequency in the stochastic case.*

The rule for generating strings from the grammar is to choose left segments agreeing with the preceding $k-1$ *symbols and to multiply the corresponding relative frequencies conditional on the preceding* $k-1$*gram. Recognising strings is done in the same way.*

Example 7.7 *Let* $\mathcal{L}$ *be the stochastic language* $\{ab^n a(\frac{1}{2^n}) : n \in \mathbb{Z}^+\}$ *and take* k *to be 3. I write* $<$ *for the start of a string and* $>$ *for its end. Then I get* 3*grams:*

$< ab,\ aba,\ ba >$ *from the string* aba *which occurs half the time in any sufficiently large sample from* $\mathcal{L}$*. I also get:*

$< ab,\ abb,\ bba,\ ba >$ *from the string* $abba$ *which occurs in about one quarter of all sufficiently large samples from* $\mathcal{L}$*. Given a sufficently large sample or alternatively by using a little arithmetic, we conclude that the local stochastic grammar of order 3 for the language will be:*

$$< ab : (1.0),\ aba : (0.5),\ abb : (0.5),\ bba : (0.5),\ bbb : (0.5),\ ba >: (1.0)$$

Now this is a generative grammar and a recognition grammar with or without the frequencies. If we wish to obtain the language with the frequencies, we start off at the begin symbol and have no choice about what comes next, it has to be ab, so we write down $< ab$*. Now with ab as the given* 2*gram we observe that we have an 0.5 chance of getting an a following and an 0.5 chance of getting a b, So half the time we go to* $< aba$ *and the other half we go to* $< abb$*. If we take the first case, we see that we have no choice but to terminate with* $< aba >$ *as our* *string, which therefore happens half the time. In the second case, we have* $< abb$ *and there is a probability of 0.5 that we follow with an a, in which case we terminate with one quarter of our cases being* $< abba >$*, and an 0.5 probability of following with another b to get* $< abbb$ *so far.*

It is clear that this will generate the same language as the finite state diagram of fig.7.1, so we have an alternative class of grammars to the finite state grammars. If we ignore the probabilities, we get the non-stochastic language back. Finite samples will consistently tend to underestimate bbb against bba and abb against aba, but the discrepancies get small fast with sample size.

To recognise a string or to compute its probability is done as follows. Suppose we are given the string abbba. We can go immediately to $<$ *ab and match the first three symbols of the string and write down a probability of 1.0. next we match abb and assign a probability of 0.5. Next we match bbb with probability 0.5,then bba with probability 0.5, and finally we match ba* $>$ *with probability 1.0. Multiplying the five numbers together gives 0.125 as the probability of the string. For the non-stochastic case we simply note that matches to* ab^na *are all possible and no others are.*

The local grammar, stochastic or not, comes with an inference procedure which is very simple and easy to compute. It assigns a small but non zero probability to getting *abbbbbba* on the basis of the sample of fifteen strings discussed above, but a zero probability to *abab*, for $k > 1$, which accords tolerably well with my personal prejudices in the matter. For k = 1 we obtain simply the relative frequencies of the symbols, and any string is possible provided that all symbols in the alphabet are included in the sample. It is possible to accomplish grammatical inference for finite state machines with rather more work, and for more general categories of language with more difficulty. The language consisting of $\{a^nba^n \ : n \in \mathbb{Z}^+\}$ recall, cannot be described by any finite state machine, although it is easy for a push-down stack machine. The distinctions are dealt with lucidly by Eilenberg, see the bibliography.[5]

[5]It may be remarked that many books on languages and automata are written by mathematical illiterates for mathematical illiterates, which makes them hard

It is instructive to infer a local grammar of order k for different k from the language $a^nba^n; n \in \mathbb{Z}^+$, or some finite sample of it. The stochastic language $a^nba^n, (\frac{1}{2^n})$ likewise. For $k \approx 20$ and around a million samples, you are unlikely to be able to tell the difference between the language generated by the grammar and the stochastic language. This has some implications for doing practical inference from finite samples. Note that the language generated by the local grammar of order k from the whole language as a 'sample' is in general a superset of the original language. When it isn't, we say the language is *local of order k*, in this case they are the same. Formally:

Definition 7.14 *A (stochastic) language is* local of order k *iff it is identical to the language obtained from the local grammar of order k.*

It is easy to see that the language $\{a^nba^n : n \in \mathbb{N}\}$ is not local of order k for any k.

It is clear that the whole business of modelling languages, or more realistically language samples, can be done using a variety of possible classes of model, and that the class of local grammars of order k for different k at least makes the job easy if not always altogether convincing. The disenchantment with local grammars of order k may be reduced slightly by going to a local grammar of order $k + 1$.

Definition 7.15 *The* Shannon filtration *for a language $\mathcal{L}$ is the sequence of local grammars of order k for all positive integers k.*

We can take the Shannon Filtration of a natural language sample, at least up to the point where k is as big as the sample, when it gets pretty silly. It gets pretty silly long before that as a rule; also impractical to store. For instance we may do what IBM's Yorktown Heights

to read, and Eilenberg is a notable exception. His book is hard to read for quite different reasons: he assumes you are intelligent.

Speech Group does and store trigrams of words extracted from English text, together with the counts of the last word in the trigram for all possible bigrams. Shannon used almost exclusively the filtration I have named after him for getting better and better approximations to English. Convergence is not very fast and there are better filtrations. Finding a good one has importance in the study of natural language and such applications as speech recognition and OCR.

7.1.4 Inference of ReWrite grammars

If I specify a language, the question arises, can I infer a grammar for it? For the regular or finite state languages, in the stochastic and non-stochastic cases, the answer is yes. It follows that if I have a finite sample of a (stochastic) language, I can always obtain a grammar, although this may not be the one I want. It is worth a little detour into regular languages in order to examine this matter.

The treatment I shall give extracts the juice from the papers of King Sun Fu on the topic, and makes it much plainer what is going on. [6]

The approach is one of compression. It is easiest to illustrate the method with an example, then to contemplate the abstract issues and finally to formulate the rules. This is how algorithms and theorems are devised.

Suppose we are given the language $\mathcal{L} = \{ab^na : n \in \mathbb{N}\}$, or if you prefer $\{aa, aba, abba, abbba, abbbba, \cdots\}$. We write the lot out as a sort of infinite finite state machine which just happens to be disconnected: any subset of ab^na for $n \leq N$ will be a perfectly good finite state diagram. I use the symbol $<$ for the begin states and $>$ for the end states. This gives us $fig.$7.2.

The next step is to start gluing together the nodes and arcs, starting

[6] After doing it my way, I discovered that Eilenberg had got there before me, and he explains how somebody else had got there before him. Such is Life.

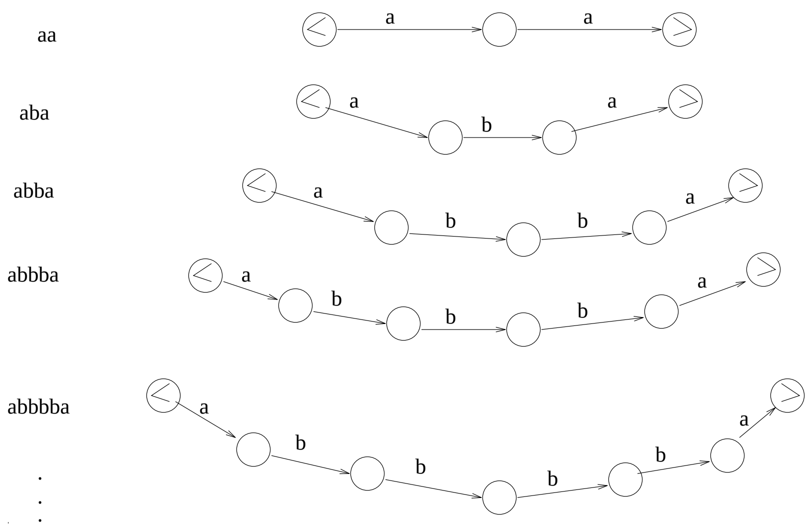

Figure 7.2: A finite state language and an infinite state grammar for it.

with the begin and end symbols, and then proceeding from the left. The next stage is shown in $Fig.7.3$.

This gives us the possibility of gluing the first nodes you come to from the start together, and identifying all the arcs also. This is safe because all the arcs have the same symbol, a, on them. This gives the diagram of $fig.7.4$, where we see that we can proceed to identify from the left to get the 'tower' of b symbols.

Finally, in $fig.7.5$ we amalgamate the tower into a self loop, and all the final arcs also get amalgamated. This yields the necessary finite state diagram for the language.

All that remains is to carefully formulate the gluing rules which make the transition from the original diagram to the final diagram. It seems reasonable that for finite state languages there *are* such rules; what we need is a procedure which tests to see if the language generated by the transformed network is altered by the gluing operation. This looks to

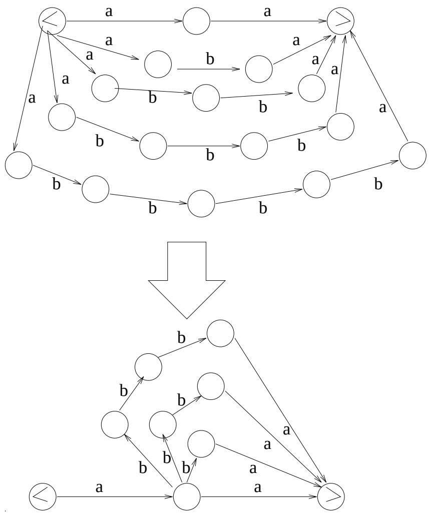

Figure 7.3: Gluing bits together.

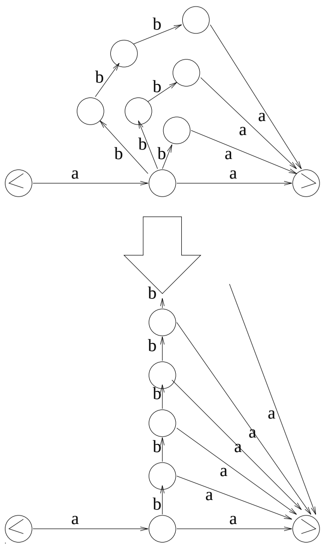

Figure 7.4: Gluing more bits together.

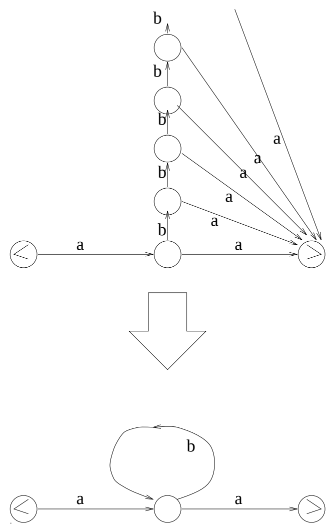

Figure 7.5: The final gluing.

be a finitary business.

The details may be found in Eilenberg's excellent *Automata, Languages and Machines*, Vol.A. Chapter III section 5, although the terminology requires some translation. Note the reference to Beckman's IBM Research Report.

The ideas outlined can be turned into operations on ReWrite grammars, where the methods rapidly become virtually unintelligible. It is much easier to visualise them in terms of finite state diagrams, or *automata* as Eilenberg calls them. In the ReWrite grammar form, we have just outlined the ideas for inference of finite state grammars which King Sun Fu described in the papers cited in the bibliography at the end of this chapter.

For the stochastic case, there are some interesting issues: you will recall our inference of a local grammar for the stochastic language $\{(ab^n a, \frac{1}{2^n})n \in \mathbb{Z}^+\}$. The same process can be applied to a finite sample. It can also be applied to the right segments of a stochastic lan-

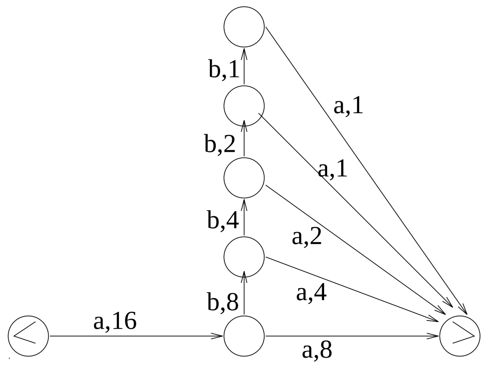

Figure 7.6: The Stochastic case.

guage obtained by doing identifications from the left, as above. All that is necessary is to keep counts of the number of strings and to add them when we do the amalgamation. Then if we had the sample: $\{(aa,8),(aba,4),(abba,2),(abbba,1),(abbbba,1)\}$, where the numbers give the string occurrences, we may do the same process of identifications, to give the diagram of $fig.$7.6 where the counts on the arcs have been obtained by summing:

Now the diagram has merely summarised the data given, and does not produce anything essentially new. Nevertheless, it is tempting to collapse the column, even though it has finite height. We might justify this by observing that from the top of it, the view backward looks, if we reduce the counts to fractions, exactly the same as the view from the topmost but one-provided we do not look too far back. If we look at most two nodes back, the top of the tower sees behind it a 0.5 chance of coming in from its predecessor by emitting a b, and an equal chance of going home by emitting an a. The predecessor sees exactly the same, as does its predecessor. This suggests that amalgamating the top of the tower with the node preceding it will be harmless, and we might just as well identify the arcs leaving, because they go to the same place with the same symbol. This gives a self loop at the top of the tower. Of course, it is only locally harmless. It actually changes the language generated; it increases it to an infinite set. And having done this, we now discover that the view forward from every node of what is left of the tower is the same as for every other such node, so we might as well amalgamate them all.

We see then that we can compress the diagram just a little more than is altogether good for it, and if we do, we get a model out which expects other things than those we actually observed. We could do even more compression and get out even larger languages: in the limiting case we blithely identify every node, and we expect any non empty string on the alphabet $\{a, b\}$. If there are p as and q bs, the probability is $(\frac{32}{47})^p(\frac{15}{47})^q$. The reader is encouraged to satisfy himself that the result is a well defined stochastic language and that all the probabilities sum, in the limit, to 1. This might suggest a neat way to work out some of those formulae involving multinomial expansions which are so tiresome to remember.

This can be done with any language sample on any alphabet, the business of amalgamating nodes and summing counts can be performed with more or less enthusiasm and corresponding disregard for the actual data. This gives an alternative to seeking hidden markov model intialisations by going into seedy bars if you prefer, for some reason, to avoid them. [7] It is quite entertaining to experiment with some wild language samples (such as you might get out of a Vector Quantiser for speech) and see if you can find interesting and plausible rules for amalgamating nodes.

It is possible, as was indicated earlier, to modify the finite state diagrams or automata for the case of Context free grammars; the idea here is to allow boxes to contain copies of themselves as well as being part

[7] Alternatively, you can go into seedy bars with a clear conscience, knowing that you no longer have to worry about meeting people who will expect payment for slipping you a suitable initialisation.

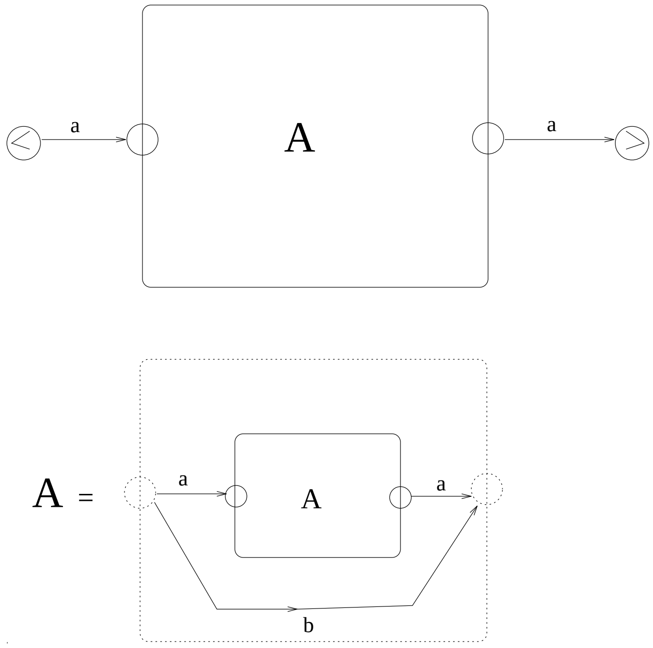

Figure 7.7: A Push-Down Stack Machine.

of a network. The case of the language $\mathcal{L}_{cf}$ discussed above is shown in such a form in $fig.$7.7.

Inference for such things is possible in special cases, but one might reasonably doubt if this is a problem that is likely to occur in practice. If we think of the problems faced by a cockroach trying in his limited way to remember a sequence of muscle contractions, say, and if we suppose that allowable sequences that have successfully got the 'roach where he wanted in the past have had some particular form, then we observe that although some small amount of palindromic structure is conceivable, it seems, *a priori*, somewhat unlikely that it would get carried very far. And we are most likely to be concerned not with the case of a deterministic grammar or language, but a stochastic one, and with a finite sample. The reader might like to try the following experiment: construct two grammars for a small alphabet, generate a dozen strings from each, put the strings from one grammar into a red box and from the other grammar into a blue box. Procure a small child of age three or thereabouts. Read the strings from the red box grammar while ostentatiously pointing to the red box, then read the strings of the other grammar while pointing to the blue box from which they were drawn. Now generate a few more strings, and ask the child to point to the appropriate box. See if the child gets the right answer. If not, is it colour blind? Did you make an injudicious choice of grammars? Try some other strings not generated by either grammar and see if the child is uncertain. The child is doing grammatical inference on the sample provided, and it is of some interest to speculate about the class of grammars it is prone to use. If instead of reading out symbols and making noises, a sequence of brightly coloured blocks is used, does the child still use the same grammatical inference process?

Return child after use.

The grammatical inference problem can be recast to other forms, and other problems, notably a large fraction of those occurring in I.Q. tests, can be viewed as variants on grammatical inference. Short sequences of distinguishable events often occur in nature, and classifying them is something which may be rather desirable for an organism wishing to reproduce itself before being eaten or starving to death. It would be rather nice to know to what extent the central nervous system of animals uses consistent methods for obtaining such classifications, or equivalently, whether some particular class of stochastic grammars is preferred. I shall return to these somewhat speculative matters later when discussing a new class of neural models.

7.2 Streams, predictors and smoothers

A language as we have defined it is a set (usually infinite in principle) of strings. It may be noted that natural language fits (uneasily) into this definition in several different ways; it may be taken with alphabet the printed characters, in which case we can take the words as the strings, or the sentences, or even the paragraphs. If we take the words, then the strings of one language become the symbols of the next, where the strings may be sentences. Or paragraphs, or chapters, or books, or for that matter libraries... So natural languages are actually hierarchies of languages as we have defined them, which suggests that we may not be defining them very intelligently.

What we are confronted with on opening a book is, after we have mastered the convention of scanning from left to right across a line, down a page and up to the next page or turning over, a stream of characters, or a very long string indeed.[8] Since we don't read it all at once, and don't know how long it is precisely, it makes sense to idealise it as we do a signal received from a television transmitter, as an infinitely long sequence of things. In this case as an infinitely long sequence of symbols. This is how text files are thought of as stored on disk in a computer, although they are not in fact, happily, infinitely long.[9] Then a white space is a *separator* symbol and a full stop, '.', is another, while linefeed carriage return, twice, is another. This allows us to chop the stream up into strings which then provides us with a language, in fact several neatly nested.

Definition 7.16 *A* stream *or* time series of symbols *on an alphabet A is a map from $\mathbb{Z}^+$ into A.*

A local grammar of order k makes just as much sense for a stream as it does for a stochastic language. In fact if we are given a sequence of strings from a language, we can turn them into a stream by either putting begin $<$ and end $>$ symbols around them and just concatenating, or perhaps replacing $><$ by a white space in this arrangement to save on symbols.

[8] If you are used to reading chinese you may feel differently about this, but then I expect you are having a lot of trouble reading the present work.

[9] The reason for allowing infinitely long sequences is not that anybody expects to meet one and be able to say with satisfaction, 'wow, that was infinitely long', it is more of a reluctance to specify the longest case one expects to get. In other words, it is done to simplify things; the infinitude of the natural numbers is a case in point. There may be some grounds for reasonable doubt as to whether this in fact works: sometimes one merely defers the difficulties to a later stage.

Definition 7.17 *A* stream sample *on an alphabet A is a map from the set [1..N] to A, where [1..N] is the set of all positive integers less than N+1.*

In real life, we are given stream samples. N may be several million or more. A may be $\{0,1\}$ in computer applications. A local grammar becomes a *predictor* of what the next symbol may be as it moves along (up? down?) the stream.

Definition 7.18 *A* predictor *of order k for a stream over A, is a map from the set of $k$grams in a local grammar for the stream to the set of pdfs over the alphabet A.*

In other words, if I give you k symbols which have been found to occur consecutively, a predictor of order k will accept this as input and return a probability, conditional on the input, for such a $k$gram being followed by any given symbol of A.

Example 7.8 *Go through the stream or stream sample and obtain a local grammar of order 2 by listing all trigrams. Now collect together all trigrams which agree in the first two places, and count for each symbol in the alphabet the number of times it appears in the last place. For example, the preceding two sentences would have the trigrams*

```
_an
_al
_ag
_a_
_ap
```

which start with a whitespace (replaced here by an underscore) and have 'a' as the second symbol. The first and second each occur twice. Thus the probability estimate of getting ' an' is $\frac{2}{7}$.

Definition 7.19 *The predictor which stores $k+1$grams by storing all those with the first k symbols the same and retaining a count of the different possibilities for the last symbol is called a* local predictor *of* order *k.*

It is easy to see how such a predictor could have its uses in an OCR program. If you were to collect trigrams from the text you were sure about, you could compare with known statistics for English and decide if it was English, or perhaps some related language such as bureaucrats gobbledegook. Then once you had a plausible identification of the language, you could clean up 'Profes5or' by using two sets of data, the relative probabilities of 'es5', 'ess', 'esa', and so on from the probabilistic model for the character, and the relative probabilities from the predictor in the absence of any actual information about the character except its context. A little thought will show how to express the net probability of the character being a '5' or an 's'. Such a system will easily improve the performance of an OCR system. Indeed it can be used to build a font independent OCR device as follows:

Suppose that the text can be segmented into characters with high reliability and that the object language is known. I scan the text and start with the first character. I assign it at random to one of the letters of the alphabet. I move on to the next character. If it is the same as one I have seen before in this data, I assign it to the same letter as last time, if it has not been seen before, I assign it to a new letter so far unused, at random. I continue in this way until all characters have been assigned letters. This is simple clustering.

I have now got the text transcribed to ASCII characters, but the text has been coded by a 'Caesar' code, that is a 1-1 substitution scheme. Now I use the known statistics of the language to break the code, a matter of a few minutes work by hand, and much faster on a PC.

There are a few nuances which the reader is urged to explore. Alternatively, if he wishes merely to use a cheap font-independent system, he can buy one from my company *Machine Intelligence* of Western Australia.

Given a book of English text to scan, with the text stored as letters not characters, the little green *thing* from outer space would be struck at once by the fact that one character is much more common than any other. I am referring not to the letter 'e', but to the white space separating words. These characters, used by us as separators, may be identified as separators without prior knowledge. This is just as well; in speech there are no acoustic cues telling the auditor when a word has ended- there are no white spaces. It would be easy enough to put them back in were they left out of printed text however. How this may be done will be described next.

7.3 Chunking by Entropy

Suppose we run a local predictor of order k over a stream sample and consider at each point in the stream the probability distribution or more generally the *pdf* over the alphabet at each stage. One of the important things we can say about the predictor is what it's entropy is at each point of the stream. Another thing we can do is to compute the amount of 'surprise' which the actual symbol produces at each point of the stream. We can also compute the mean entropy of the stream sample relative to the predictor by summing the last number over the entire stream sample and dividing by the number of symbols, and we can also compute the mean entropy of the predictor. If the predictor is a good model for the stream, these will be about the same. If one predictor is better than another, we can tell by comparing the mean information (surprise) supplied by the stream to each predictor, and picking the smaller number. God, who is omniscient, is never surprised by what happens, which must make for unspeakable boredom.

It was observed by Shannon that as we run along English text, the entropy of a loal predictor of order k for $k > 1$ decreases inside a word until it comes to the white space. If I give you *by_c* as a $(k-1)$gram, the number of possible next symbols is large and the probability distribution is broad. If you get *y_co* as the $(k-1)$gram, the number of next symbols is somewhat less and their probabilities somewhat higher, so the entropy is less. At *_com* there are very few next possible symbols in English, and m, p, are much more likely than the others. At *comp* the next symbol is a vowel and the entropy is very low. At *ring* the next symbol is either white space, comma, s or e to high probability. And at *ing_* it is virtually impossible to say what the next symbol will be and the entropy has shot up again. So even using 5grams to obtain a local predictor, it is easy to find separators without prior knowledge of which symbols have been selected for that purpose. Alternatively, if someone were to go through text and remove all the white spaces, it would be possible to go through and do a reasonably good job of putting them back in again. Or to put it in a third form, we can use a predictor to go through text and chunk it into words.

Definition 7.20 *Using a predictor to separate a stream of symbols into consecutive strings as described above is called* entropic chunking*, and the strings so found are known as* chunks*.*

Once chunks have been found by entropic chunking, the stream of symbols may be reconstructed to become a stream of chunks.

Definition 7.21 *The process of mapping a stream of symbols into a stream of chunks is called* UpWriting*; we say that the stream of chunks is at a* higher level *than the original stream of symbols. Recovering the stream of symbols from the stream of chunks is called* DownWriting*. UpWriting entails compiling a dictionary of chunks and assigning a new symbol to each chunk. DownWriting entails looking up the dictionary and replacing each chunk with the corresponding string of symbols.*

It must be conceded that the practicalities may be considerable; finding maxima in the entropy may give not words of English but stems and inflections. Thus the s at the end of a word used to denote plurality may be a separate chunk, and similarly with *ed* to denote tense. Inflected languages simply haven't evolved to the consistency of having a white space used to denote a chunk end. [10]

Experiments make it convincing that the method of entropic chunking by a local predictor works sufficiently well to be applied to machine language programs, *inter alia*, and that they discover the obvious structure of operators and addresses.

As well as chunking by entropy, we can chunk by the amount of information supplied by a character, or the *surprise* as I have called it earlier. This will, by definition, give the same average result as chunking by entropy if our predictor has been obtained by counting occurrences in the text itself, but it can give different results on different texts.

The process of UpWriting yields compression; it is easy to see that knowing the language in which something is written gives a large amount of redundancy to be extracted and that this dictionary method will accomplish something of that extraction. It is also easy to see that there may be a language, or more exactly a stream, for which chunking doesn't work. Let it be noted that there do exist streams for which it does work, and that natural language texts are among them.

The process of UpWriting a stream into a higher level stream may be iterated. In the case of natural language streams, this yields a progressive decomposition into larger and larger chunks, as has already been described. In each stage there is some compression of the original stream, and each stage can be implemented in the same way by a local predictor of fairly low order. It is easy to persuade oneself that one attains greater compression than by having a high order predictor at least for some kinds of stream. Let us call such streams *hierarchical* streams. Formally:

Definition 7.22 *A stream is said to be* hierarchical *if it allows com-*

[10] Grammarians used to believe that high levels of inflection showed a highly evolved language, probably because Latin was regarded as morally superior to English, German or French. Then it was discovered that Chinese used to be inflected a few thousand years ago but the chinese very sensibly gave it up.

pression by a sequence of UpWrites by low order predictors.

Since it is apparent that much human information processing involves a hierarchy of structures, hierarchical streams are natural things to study in that they are relatively simple cases of the decomposition process that, presumably, extends to learning to drive a car. The existence of clichés tells us that English is at least partly hierarchical to two levels, but plainly there is more going on than that.

7.4 Stochastic Equivalence

Consider the natural language English with symbols at the word level. Take a stream, any stream, and consider such sentences as '*The cat sat on the mat*'. Regarding this as a $k$gram for $k = 6$ suppose we collect all other such 6grams which agree with the given one except in the second symbol. Thus '*The dog sat on the mat*' will be in the list. We have something very like the local predictor of order 6 except that we take account of context on both sides. We will get a similar object out, a probability distribution over the alphabet (in this case dictionary) of symbols. Of course, I didn't have to pick the second place as the 'variable', it could have been any location.

Definition 7.23 *A* context *of order k for a stream $\mathcal{S}$ over an alphabet A is a string of the stream together with a designated place between* 1 *and k which is replaced with a placeholder symbol. The symbols of the alphabet together with their relative frequencies which match the place holder when all the other symbols of the context are matched define the* context distribution *for the stream and the context.*

Suppose I have two context distributions arising from different contexts. Then it might happen that they give rise to distributions which are pretty much the same; as distributions, they may not differ by more than I would expect from simply sampling variation. There are various measures of the difference between distributions, from χ^2 to Kullback-Leibler via a straight euclidean metric, and I shall not go into any of them here. It suffices to observe that it makes sense to decide that some context distributions are not distinguishable from others. It is clear that this will happen in English; a cat and a dog are both things that can sit on mats, but they can do quite a lot of other things in common too. So the context distribution for 'The -?- sat on the mat' is probably not discriminable from that for 'She played with the young -?-' for most stream samples. If we have a distribution arising from a context or a local predictor (a particular sort of context with the place holder at the right hand end) which is not well represented because the context is a rare one, this could be useful as we could get a better estimate of the true distribution by pooling with a distribution obtained from another context. This is fraught with risk, of course, but there is a risk involved in having too few data to make a reliable estimate. Life is pretty fraught all round.

At the letter level in English text, there are some very distinctive distributions which may be found by examining different contexts: separators and punctuation symbols for example, and digits also form a cluster in the space of probability distributions over the alphabet of symbols. (Which looks exactly like a piece of $\mathbb{R}^n$, where n is the size of the alphabet). Vowels form a distinguishable cluster in this space, with a consistent ratio of counts in a variety of different contexts. And some reflection suggests that this feature of natural language is not confined to English, nor is it confined to the letter or word level.[11]

Any such regularity as this can be exploited in a grammar (regarded as a data compression device for specifying streams) or language model. Suppose I were to assign labels to the distributions. If there were too many of them and no grounds for pooling them, then I could have as many distributions as contexts, which is many more symbols than my alphabet originally contained. If, on the other hand there are very few, it might be desirable to do a new sort of UpWrite to the names of the distributions. The DownWrite from this would require making

[11] Past a certain point it is called 'plagiarism'.

some choice of which symbol from the distribution to select. Instead of storing the symbol names and assigning them equal length, we would do well to send the distribution and a derived coding for the symbols exploiting the fact that commonly occurring ones should have short names. This could be a useful feature of any device which had to remember lots of symbol strings. A stream or language might have the property that it would be more economical to code clusters and places within the clusters than just to send the stream, it would depend on the extent to which clustering in the space of context distributions occurred.

Example 7.9 *The stream is obtained using 16 symbols a through to p. There are auxilliary symbols A,B, C,D which are 'bags'. A is the bag containing a, b, c and d, and so on down to D which contains m, n, o, p. The bags are selected cyclically, A,B,C,D,A,B,C,D.. until 1024 have been listed altogether. Each time a bag is selected, one of the symbols inside the bag is picked with equal probability one quarter. So something like aeimbfjnc.. comes outof the process. Now it takes 4 bits to specify one of the 16 symbols, and there are 1024 of them, so a straight listing of the stream sample would occupy* $4K$ *bits. But this does not exploit the fact that the bag structure is cyclic. If we have four bags, A,B,C,D then we can code each symbol by coding the bag it comes from with 2 bits, and the symbol in the bag by another 2 bits. Now the cyclic bag structure allows us to specify the bag 2-bits for each bag in much less than* $2K$ *bits: we need to code the information that the cycle ABCD recurs* 256 *times. A thousand bits or so would seem more than adequate for this. So we save at least* $1K$ *bits by using the structure of the stream.*

Definition 7.24 *A stream is said to have a* stochastic equivalence *structure when it is the case that the context distributions for some order k of context form clusters in the space of probability distributions over the alphabet. If two symbols a and b occur with similar probabilities in a significant number of distributions which are all close in the space of all distributions over the alphabet, we may say that a and b are* stochastically equivalent.

It is clear, just by stopping and thinking for a while, that English at the letter level has clusters for digits, for punctuation and for vowels. It is easy to confirm, by computation on English text, that for $k = 4$ these clusters actually exist. It is rather more expensive to compute the distributions for English at the word level, particularly for large k, but it is possible for $k = 3$, and again one's intuitions are borne out: synonyms and also antonyms are discovered sitting in the same distribution; class name exemplars seem to be implemented as elements of such distributions. For example, the class of domestic animals (including children) constitutes a distribution which occurs in many texts in many different contexts. Buses, taxis, cars, boats and aeroplanes figure as prominent elements of a distribution which again recurs in many stream samples and many contexts. This may not be altogether surprising, but it is hardly a necessary feature of a stream.

The above definition lacks the precision which one might reasonably expect in book written by a mathematician, even one who has gone downhill into engineering, and it is a matter of some irritation that it is not easy to give a satisfactory definition of what it means for a set of points to be clustered. (A probability distribution over an alphabet of N symbols *is*, of course, a point in $\mathbb{R}^N$.) There are many different definitions, see any of the books on clustering, and althought they usually agree on obvious cases, they may difer on the marginal ones. For present purposes, I suggest the reader thinks of a modelling by a mixture of gaussians and uses Rissanen's stochastic complexity criterion to decide on the optimal number of gaussians. If the optimal number is zero or one, then we can say that there is no clustering. If the number is small compared with the number of data points so that many distributions find themselves packed closely to other distributions, then we obtain a working definition of stochastic equivalence. I do not propose this as any more than an *ad hoc* heuristic at this point.

If there is a clustering structure for the context distributions, then we can take all the $k$gram predictors which have 'close' distributions in the last place, and deem them to be 'close' also. After all, they ought to be stored together in any sensible prediction scheme precisely because they predict the same sequent. So we can cluster the $k-1$grams. This gives us the possibility of a new type of UpWrite, a stochastic equivalence class of $k$grams gets written to a new symbol.

Definition 7.25 *An UpWrite may be of two types: the* chunking UpWrite *assigns a symbol in the UpWrite stream to a* substring *in the base stream. The* clustering UpWrite *assigns a symbol in the UpWrite stream to a* stochastic equivalence class *in the base level stream.*

7.5 Quasi-Linguistic Streams

Definition 7.26 *A stream is said to be* quasi-linguistic *if it is hierarchical and at least some of the (chunked) UpWritten streams admit a stochastic equivalence structure. In other words, we allow ourselves to do both kinds of UpWrite, into chunks of consecutive symbols or into stochastic equivalence classes of symbols. If any sequence of either UpWrite yields data compression, then the stream is* quasi-linguistic

Although the definitions are not as tight as is desirable, they are adequate for there to be reasonable grounds for consensus on whether quasi-linguistic streams exist and whether a particular stream is an example of one. Now it is easy to convince oneself by a certain amount of simple programming that samples of English text of less than a million characters give reasonable grounds for believing such things exist. There are many other possible structures which may be found in Natural Language stream samples, but whereas grammarians have been usually concerned with finding them by eye and articulating them in the form of rules, I am concerned with two other issues: first the question of inference, of how one extracts the structure from the sample and what algorithms make this feasible. Second the investigation of more powerful methods of specifying structure than by rule systems. It should be plain that it is certainly a structural feature of natural language if it exhibits a stochastic equivalence structure, but it is not one which is naturally articulated by giving a set of rules.

If such a structure is found in a stream, then it can be used for smoothing probability estimates of predictors.

Suppose for example we are using a trigrammar at the word level to constrain the acoustic part of a word recognition system, as is done at IBM's Thomas J Watson Research Center in Yorktown Heights. It is common to encounter a new word never seen before (proper names, of ships, for example) and also common to encounter two words never seen before in that order. Suppose we have a string $ABCDEF$? and want to predict the ? from the preceding words, EF, but have never seen EF together, although we have seen E and F. We may first try replacing E by something stochastically equivalent to it in the context CDE. There are, presumably, a number of other symbols which might have occurred in place of E after CD. We identify the probability distribution as belonging to a cluster of other such distributions. We replace the given distribution by the union of all those in the local cluster. This gives us a list of other symbols which are stochastically equivalent to E. We weight them by their relative likelihoods, and list them in the form $CDX_1, CDX_2, \cdots$. Now we take the F and replace it by all symbols Y_i which are stochastically equivalent in the context DX_j. Again we weight by their probabilities, and include the DX_jF case as of particularly high weight. We take the union of the resulting distributions for the weighted X_jY_i as the bigram predictor. Then we try to recognise this distribution, i.e. we choose the closest cluster of known distributions. This is our final distribution estimate for the successor of $ABCDEF$. Similar methods apply whenever E or F is a new word which has never been seen before.

If one reflects on the way in which we use context to determine meaning of a new word in a page of text, it may be seen that this method holds out some promise. Semantic meaning *is* inferred from other known meanings and the syntax of language. If we take the narrowest of stochastic equivalence sets, we may replace some of the symbols in a context with a stochastic equivalence class, and generate a larger stochastic equivalence class by the process described above. These classes become very large quite quickly, and rapidly tend towards things like the grammatical categories 'Noun', 'Adjective', 'Transitive Verb', and so on.

Quasi-linguistic stream samples are of interest largely because they constitute abstractions of natural languages, and moreover the abstractions have been constructed so as to facilitate the inference of structures. This 'Newtonian' approach to language analysis differs from the traditional 'scholastic' approach, which you may find described from the writings of Chomsky on. The question arises, to what extent do streams of symbols occur, other than in natural language, which exhibit this structure?

A real number is a symbol, and so a time series of real numbers may be considered in this manner. There is of course a metric structure on the symbols which is usually presumed to be known. The interested reader is invited to investigate extensions to syntactic methods to real valued time series.

7.6 Graphs and Diagram Grammars

King Sun Fu, who was an engineer working in Pattern Recognition and not an oriental potentate, was one of the first to note that significant pattern recognition is syntactic and his works may be consulted for examples, some rather artificial, of cases where syntactic methods can be applied. Perhaps the classic case is an attempt to segment handwritten characters into stroke primitives, and then to extract the syntax of the sequence of primitives. The primitives were extracted by eye, and may not have been the best choice.

The difficulties engendered by the segmentation problem have stimulated attempts to go to higher dimensions than one and to apply to more complicated entities than symbol strings, the same kind of syntactic ideas. A *directed graph* is a set of nodes with arcs going between some of them, an arrow showing the direction of the arc. A graph is a directed graph with arcs always going both ways, or alternatively with the arrows left off altogether. It is easy to see that you can glue some such objects together to get others, and that there are thus some generalisations of concatenation for strings. It is plain that some sort of segmentation and decomposition of graphs (directed or not) can be done and that rules for allowing some subgraphs to be rewritten to others, just as in the standard rewrite grammars, are possible. What is not apparent is whether the things arise in nature with such decompositions built in. It is not plain how you make a predictor or a context, and without these you cannot easily get chunking or stochastic equivalences. The opportunities for Pure Mathematicians to occupy their time with little profit to the rest of us would seem to be excellent, but people have thought that before and been horribly wrong.

Huffman, Clowes and later Waltz have analysed the way in which trihedral edges in line drawings can be constrained when the drawings depict solid objects such as cuboids. This is a nice example of syntax arising naturally in the world, but the simplifications needed to make the subject tractable reduce the inference problem to triviality.

McLaughlin and Alder have shown how some structure in a silhouette of a hand, namely that it is built up out of fingers and a palm, can be extracted automatically and used to distinguish between left and right hand palm prints. The method has also been used to classify aeroplane silhouettes and to learn the rotation, scale and translation transformations. This will be discussed at greater length in the next chapter.

7.7 Exercises

1. Get one of the children's reading primers giving details of the adventures of Janet and John or Dick and Dora, or Peter and Jane or some similar pair. Put the sentences into the computer. Alternatively find such a source on the net if you can, and then tell me what it is. [12]

[12] As Herod said to the three wise men.

Note that teaching these sentences to a computer program is not at all the same as teaching them to a child because (a) the child has already acquired a spoken language and (b) the child can look at the pictures that come with the text and interpret them. The program can't. Nevertheless, constructing a program which can learn some of the structure is worth a try.

First write a program which takes the sentences as strings at the letter level. Construct a *trigrammar*, a list of consecutive triples of letters, including whitespace and punctuation. Use it to build a prediction model for the data, and determine how to chunk letters into words. And other things. Cope with the problem of new bigrams by the 'fallback' method, ignore the first letter. If the last letter of the bigram has never been seen before, fallback all the way to letter frequency. Compute the entropy of the trigrammar model of the text. Compare with the mean information supplied to the model by a piece of text it hasn't seen before.

Repeat for the word level. It may be convenient to have punctuation regarded as words. A vocabulary of not more than about 100 words is advisable. Again, compute the entropy of the model of the data. If you obtain estimates by trying it out on new data, It will be necessary in both cases to cope with the situation where a new symbol is encountered: it is suggested that you reserve a $\frac{1}{n}$ probability for anything you haven't seen before, given that you've seen n different things so far.

Classify the words into types by eye. Names of children, for example, form a subcategory. UpWrite the original text at the word level into 'bags', each bag containing the terms of a category such as **childname**. Now model the UpWritten stream. Determine the entropy of the model of the UpWritten stream. Determine the entropy of the model for the original stream consisting of the bag stream and the DownWrite.

2. Make up a bag grammar by choosing some particular sentence structure such as

   Article $<$ Adjective $<$ Noun $<$ Verb $<$ Article $<$ Noun $<$.

Now generate a list of possible words in each category, not more than twenty words in each. Finally generate sentences from any choice of a *pdf* for each 'bag'.

The problem is to try to recover the bag structure from the sample. Try it as follows: Take a trigrammar and for any words ab list the sequents of ab and make them into a probability distribution over the alphabet. Now in the space of all such distributions, look for clusters. If you find at least one cluster, label it C and to assign it the group of *pdf*s in the cluster. Now perform a *cluster UpWrite* on the original stream by replacing whatever follows ab by C precisely when C is the cluster containing a distribution obtained from collecting sequents to ab.

Replace the old by the new stream, and do it again. How much data is required before the categories of 'noun', 'verb', emerge? What is the entropy of the model of the stream which first predicts bags then predicts elements in a bag by assuming the probabilities for choice of bag element are independent and the mean *pdf* for the cluster? How does this compare with a trigram word model?

3. Automate the process of picking bags such as **childname** or **animalname** in the case of data taken from the children's primers. This could be described as 'real data', almost. Do you get any compression using these methods? Do you get better results than an $n$grammar for various n?

7.8 Bibliography

1. S. Eilenberg, *Automata, Languages and Machines,* Vols A,B. Academic Press 1974.

2. S. Ginsberg, *Algebraic and automata-theoretic properties of formal languages* American Elsevier Pub. Co., 1975.

3. J. Hopcroft and J. Ullman *Introduction to Automata Theory, Languages and Computation* Addison Wesley, 1979.

4. H. Ehrig, H.-J. Kreowski, G. Rozenberg (eds.) *Graph Grammars and Their Application to Computer Science* : 4th International Workshop, Bremen, Germany, March 5-9, 1990.

5. P.S.P. Wang (Ed.) *Array grammars, patterns and recognizers* World Scientific, 1989.

6. Frederick J. Damerau, *Markov models and linguistic theory : an experimental study of a model for English* Mouton, 1971.

7. G. Herdan, *The advanced theory of language as choice and chance* Springer-Verlag, 1966.

8. The *IBM Language Modelling Kit* is a set of papers produced by the Continuous Speech Recognition Group, IBM Thomas J Watson Research Center, Yorktown Heights, NY 10598, U.S.A. Some have been cited earlier.