Natural Language Grammatical Inference

An Honours Dissertation in Information Technology

Jason L Hutchens

Centre for Intelligent Information
Processing Systems
Dept. of E & E Engineering
The University of Western Australia

March 9, 1995

Abstract

This project is concerned with programming a computer to make predictions about which words are most likely to follow a small segment of English text. At first this may seem a strange problem, but I intend to show that there exist a wide range of applications that would benefit from such a program. Indeed, my motivation for approaching this problem was to provide a way of improving the accuracy of speech recognition systems.

Additionally, I am interested with the problem of Grammatical Inference. In fact, the word prediction problem and the Grammatical Inference problem are intertwined, and it seems that approaching either one will lead to the other. Grammatical Inference entails inferring a grammar for an arbitrary language from a finite set of sample sentences in the language.

It is quite easy to measure the performance of a word prediction system, providing that its prediction is given as a probability distribution. This allows us to compare our predictor with others, such as the trigram predictor developed at the IBM Thomas J. Watson Research Centre by Jelinek *et al.*

A comparison between the IBM predictor and the algorithm presented in this dissertation indicates that the method I have used is able to make broader generalisations about the language under investigation, and produces more accurate predictions about texts generated by artificial grammars.

Contents

List of Figures vi

List of Tables vii

1 Introduction 1

1.1 The Problem 2

1.2 Structure of Dissertation 2

1.3 Acknowledgements 3

2 The Problem 4

2.1 The Shannon Game 4

2.1.1 Performance Measures 5

2.1.2 Playing the Shannon Game 9

2.2 Natural Language Grammatical Inference 11

2.3 Why it is Important 11

3 Background and Concepts 16

3.1 Language Modelling 16

3.2 Grammatical Inference 21

3.3 Word Prediction 24

3.3.1 The IBM Trigram Model 25

4 The Upwrite Predictor **27**

4.1 Statistical Analysis 28

4.2 Entropic Chunking 29

4.3 Clustering 31

4.3.1 The Dog-Rabbit Clustering Algorithm 33

4.3.2 The K-Means Clustering Algorithm 34

4.4 Upwriting 35

4.5 Predicting 37

4.6 An Example 39

5 Testing and Results **45**

5.1 Artificial Grammars 46

5.1.1 Clustering 48

5.1.2 Prediction 51

5.2 Juvenile Text 52

5.2.1 Clustering 53

5.2.2 IBM Predictor 54

5.2.3 Q-Predictor 55

5.3 Sherlock Holmes 56

5.3.1 IBM Predictor 57

5.3.2 Q-Predictor 58

5.4 Analysis of Results 60

6 Applications **62**

6.1 Random Text Generation 62

6.2 Document Proof Reading 65

7 Conclusion **67**

7.1 Limitations and Disadvantages 67

7.2 Recommendations for Future Work 68

Bibliography **69**

A Implementing the Q-Predictor **75**

A.1 Statistical Analysis 76

A.2 Clustering 78

A.3 Upwriting 79

A.4 Predicting 79

B Description of Source Code **81**

B.1 Shannon 81

B.2 Extract_Ngram 84

B.3 Chunk 85

B.4 Cluster 85

B.5 Upwrite 86

B.6 Predict 87

B.7 Convert 87

B.8 Hmm 88

B.9 Gen 88

B.10 Check 89

C User's Manual **91**

C.1 Invocation of Programs in the Shannon Suite 91

C.2 Convert 92

C.3 Hmm1 94

C.4 Hmm2 94

C.5 Ngram 95

C.6 Check 95

List of Figures

2.1 Model of a noisy communications system 5

3.1 A simple finite automaton 18

3.2 Generation of a valid string by a simple hierarchical grammar 19

3.3 A 2^{nd}-order Markov model 20

3.4 Shannon's 2^{nd}-order approximation to English 21

3.5 Structure of a Grammatical Inference Engine 22

3.6 Grammatical Inference versus Chomskian Language Acquisition . . . 23

4.1 Two dimensional data with three distinct clusters 32

4.2 An example of data clustered tith the Dog-Rabbit algorithm 34

4.3 An example of data clustered with the K-Means algorithm 35

4.4 An example of the Upwrite process 36

4.5 Turing's approximation of the probability of a new symbol 39

4.6 The Hidden Markov Model used to generate the training corpus . . . 40

5.1 The Hidden Markov Model used to generate the corpora 47

5.2 Clusters found by the K-means algorithm 49

5.3 Clusters found by the Dog-Rabbit algorithm 50

List of Tables

4.1 Results of the Q-Predictor versus the IBM predictor 43

5.1 Results using the Artificial Corpora 51

5.2 Reduction in Perplexity with the Artificial Corpora 51

5.3 Predictions made by the IBM predictor 54

5.4 Predictions mmde by the Q-Predictor 55

5.5 Results using the Juvenile Corpora 55

5.6 Reduction in perplexity with the Juvenile Corpora 56

5.7 Results using the Sherlock Holmes Corpora 59

5.8 Reduction in perplexity with the Sherlock Holmes Corpora 60

C.1 Option file keywords and corresponding command line switches . . . 93

Chapter 1

Introduction

Language is a unique phenomenon that distinguishes man from other animals. It is our primary method of communication with each other, yet very little is understood about how language is acquired when we are infants. A greater understanding in this area would have the potential to improve man-machine communication.

One major question currently left unanswered in linguistics is as follows. 'How can a human infant acquire language so quickly when it has had such a short life experience?'. This question is often conveniently side-stepped by making the Chomskian assumption that the required structures are hard-wired into the brain at birth [14].

It is noted that human infants learn other complex tasks much more quickly, tasks that often involve pattern recognition. Observing this, Minsky posed the anti-question 'Why do human infants take so long to acquire language, given that they learn other complex, language-like tasks much more quickly?' [44].

It seems possible that the human infant applies its already good pattern recognition abilities to the language acquisition task, and that no language-specific structures of any complexity are present in its brain at birth. As computer programs have been developed to perform pattern recognition, then perhaps such a program could infer part of the structure of a language without having to make any strong prior assumptions about it.

1.1 The Problem

The problem that I am attempting to solve in this dissertation is that of programming a computer to play the *Shannon Game*. To play the Shannon Game, one must predict which words are most likely to follow a given segment of English text. This is exactly what a language model is required to do when used in a speech recognition system. IBM have done a lot of work in this field, and have developed a simple language model, the trigram model, for this task [32, 8, 34].

My aim is to improve on IBM's results by writing a program that is capable of automatically inferring a grammar from a Natural Language corpus, and applying this grammar to the Shannon Game. To play the Shannon Game, a stochastic grammar for an approximation to the target language must be inferred from a text sample, and as the quality of this grammar improves so too does the quality of the predictor that uses the inferred grammar.

I do not intend to restrict myself to the case where the Natural Language under investigation is English, nor do I intend to restrict myself to just a Natural Language. It should be possible for a Grammatical Inference Engine to infer an approximate grammar for any language whatsoever, be it stochastic or non-stochastic, formal or natural, finite or infinite. I am primarily concerned with Natural Languages purely because the intended application of this project is to improve speech recognition systems.

1.2 Structure of Dissertation

The next Chapter of this dissertation provides a description of the problems I am attempting to solve. Chapter 3 gives an overview of previous approaches to these problems.

In Chapter 4, I present the algorithms that I intend to use, developed after an examination of other techniques. Once the algorithms have been implemented as computer programs, they may be compared with other word prediction approaches, and this comparison is performed in Chapter 5. Several applications of the developed programs are then presented in Chapter 6.

Chapter 7 provides a critique of the work I have done thus far, and suggests what may be done in the future to improve upon the programs I have developed.

The implementation of the programs written as part of this project is given in Appendix A, and a brief description of all the relevant source code files follows in Appendix B. Appendix C contains a short user's guide for each of the programs that have been developed. The source code itself is included on the floppy disk attached to the inside cover of this thesis, and is not reproduced in the Appendices because of its length.

1.3 Acknowledgements

I would like to thank my supervisor Mike Alder for his enthusiastic support during the year, and for our many interesting conversations. I am eternally grateful that he found no use for his iron bar.

I acknowledge all the people in CIIPS who have helped me in various ways with my project. In particular I would like to thank Rob, Gek, Danny, Bruce and Paul for their comments and assistance.

I want to thank all of the people on the Internet who provided valuable information about various Natural Language topics. Of particular note are Nick Chater and Hinrich Schütze who happily let me download copies of their papers.

I must finally thank my family for understanding why I spent ten months living like a hermit and programming computers. To tell the truth, I don't think they noticed much difference. I'm just glad that they managed to put up with me during the final days.

Chapter 2

The Problem

The problem that I am trying to solve is that of Natural Language Grammatical Inference. In particular, I intend to apply the inferred grammar to the word prediction problem, which is also known as the Shannon Game. Throughout the remainder of this dissertation, I will refer to an algorithm that is capable of playing the Shannon Game as a *predictor*.

2.1 The Shannon Game

The Shannon Game was proposed by Shannon himself [50], and has been adopted as a technique of presenting the word prediction problem by Jelinek [34] and Alder [2].

Participants in the Shannon Game are presented with a string of symbols in some language. In the case where the language is English, these symbols may be single letters or they may be words. I will take the case where the participants are presented with a string of k words from an English text that contains N words altogether.

The aim of the Shannon Game is for each participant to guess which word is likely to follow their given string. Even better, each participant could suggest several words that could follow the string, along with the probability of each suggested word occurring. We may keep score by counting the number of attempts that the participant requires to correctly guess the next word, or by measuring how surprised the participant is when he or she discovers which word actually occurred.

This is all very well for human participants, who are capable of estimating their surprise on a scale of 1 to 5 [39]. A measure suitable for comparing the performance of computer implementations is required. Computer implementations may give their prediction as a probability distribution over the alphabet, and this distribution is often used by performance measures.

2.1.1 Performance Measures

Information Theory attempts to model a noisy communications channel with the aim of optimising the communication of messages over it while maintaining good performance in the presence of noise. This is achieved by encoding the message in such a way that it occupies minimal space while containing enough redundancy to detect and correct bit errors. Figure 2.1 illustrates the model of a communications system proposed by Shannon [51].

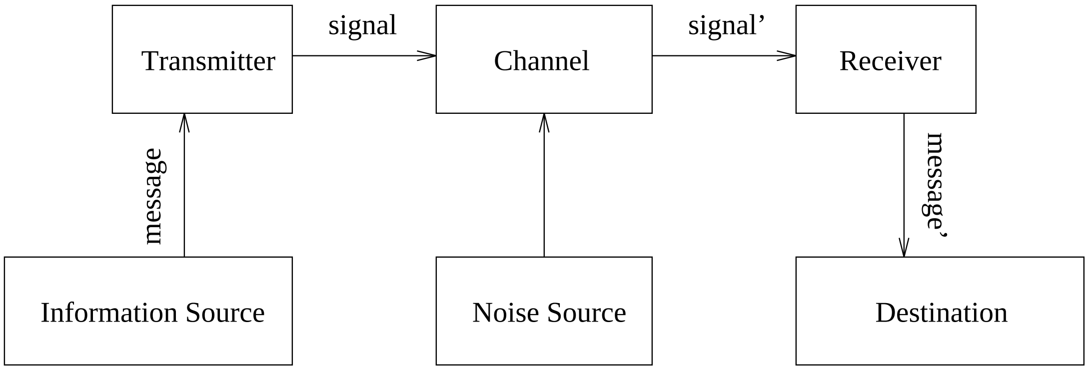

Figure 2.1: Model of a noisy communications system

Several measures were introduced by Shannon to quantify the information content of a received message, and the uncertainty that the receiver has about which symbol it is going to see next. These measures may be adopted for use in the Shannon Game, which also deals with messages of a sort.[1]

[1] English sentences are strings of words, and may be considered to be messages.

Information

The surprise experienced by the predictor when it discovers which word actually followed the string of k words it was given may be measured as the information supplied to the predictor by the text, and this measure is given in Equation 2.1. Note that $p(w|S)$ is the probability assigned by the predictor to w, the word that actually occurred, given that the string S of k words had previously occurred.

$$I = \log_2 \frac{1}{p(w|S)} \tag{2.1}$$

This is equivalent to Equation 2.2, which was derived by applying the basic logarithmic property $\log_m \frac{x}{y} = \log_m x - \log_m y$, and noting that $log_2 1 = 0$. I tend to use this second form, simply because it looks neater when printed within a line of text.

$$I = -\log_2 p(w|S) \tag{2.2}$$

The information supplied by a string of words is equal to the sum of the information supplied by these words separately. For example, $-\log_2 p(xy|S) = -\log_2 p(y|S) - \log_2 p(x|Sy)$. This means that we may measure the information supplied by an entire text by summing the information provided by each of its words, rather than having to calculate the probability of the text itself occurring.

If the predictor was certain that word w would follow the string S, it would assign $p(w|S) = 1$. If this prediction turned out to be correct, then the information supplied to the predictor would be $-\log_2 1 = 0$. That is, the predictor would not experience any surprise at all.

If the predictor decided that word w couldn't possibly follow the string S, it would assign $p(w|S) = 0$. If word w is found to follow the string, then the information supplied to the predictor would be $-\log_2 0 = \infty$. In this case, the predictor is said to be *infinitely surprised*.

These properties of the information supplied to the predictor by the text make it suitable for use as a measure of the surprise experienced by the predictor, as they agree with our intuitive notions about what it means to be surprised.

Average Information

We may measure the performance of a predictor that is presented with a text of N words by summing the information supplied to the predictor by each word in the text as given in Equation 2.3, where w_j is the j^{th} word in the text.

$$I_{total} = -\sum_{j=1}^{N} \log_2 p(w_j | w_1, w_2, \ldots, w_{j-1}) \tag{2.3}$$

This measure is equal to the surprise received by the predictor when it discovers the contents of the entire text, and is therefore equivalent to Equation 2.4.

$$I_{total} = -\log_2 p(w_1, w_2, \ldots, w_N) \tag{2.4}$$

The total information supplied to the predictor by the text provides a basis for comparing the performance of different prediction algorithms. We may normalise the total information to provide a measure that isn't dependent on the size of the text. This normalised measure gives the average surprise received by the predictor per word, and is given in Equation 2.5.

$$\overline{I} = -\frac{1}{N}\sum_{j=1}^{N} \log_2 p(w_j | w_1, w_2, \ldots, w_{j-1}) \tag{2.5}$$

Perplexity

An alternative performance measure proposed by Jelinek [34] is the perplexity of the predictor with respect to the text, defined in Equation 2.6.

$$PP = p(w_1, w_2, \ldots, w_N)^{-\frac{1}{N}} \tag{2.6}$$

The perplexity is a function of the probability of the sample text occurring according to the predictor. As the average information is a function of the same probability, we may give the perplexity as a function of the average information, and do so in Equation 2.7.

$$PP = 2^{\overline{I}} \tag{2.7}$$

Obviously, the perplexity is a monotone function of the average information. Therefore, both measures will always rank the predictions in the same order when measuring the performance of the language model. The advantage of the perplexity measure is that it gives the average number of equi-probable symbols that the predictor must choose between.

When all of the words in the vocabulary are equi-probable, the average information experienced by a predictor which uses the perfect language model will be a maximum. Equation 2.8 gives the maximum average information possible from a language with A words in its vocabulary, which occurs when $p(w) = \frac{1}{A}$ for all w.

$$\overline{I}_{max} = \log_2 A \tag{2.8}$$

It is therefore wise to use the perplexity measure when calculating the percentage improvement of one language model over another, as the units of the perplexity measure are words,[2] and its upper bound is calculable.

Another measure that may be used to determine the performance of the predictor is the average number of guesses it requires to correctly guess the sequent word. This measure is referred to as the *rank* of the predictor, and is denoted by the symbol R. It is interesting to note that the rank of the predictor is a monotone function of its perplexity, which in the limit satisfies Equation 2.9.

$$R = \frac{PP + 1}{2} \tag{2.9}$$

This Equation was obtained by finding the expected value of a number between 1 and PP, all numbers in the range being equi-probable.

Entropy

Another measure introduced by Shannon for use in Information Theory is the *entropy*, which is the surprise that the receiver expects to experience when it is shown the next word, given that the language model correctly represents the text [51]. It may be considered to represent the amount of uncertainty that the receiver has about which word is likely to come next. Entropy is defined in Equation 2.10, where $w[i]$ is the i^{th} word of an A word vocabulary.

$$H = -\sum_{i=1}^{A} p(w[i]) \log_2 p(w[i]) \tag{2.10}$$

[2] The other information measures are in units of *bits*.

The entropy function captures our intuitive notions about uncertainty. For example, when the receiver is completely certain about which word is coming next, the entropy $H = 1\log_2 1 + 0\log_2 0 = 0$. When the receiver believes that each word in the vocabulary has an equal chance of coming next, the entropy is maximum, and the receiver is completely uncertain. In this case, $H_{max} = I_{max} = \log_2 A$.

2.1.2 Playing the Shannon Game

Let us now play the Shannon Game. We will play with the constant $k = 5$, so that each string I give you contains five words. Read the string given below, and try to guess which word will follow it.

The cat sat on the

The chances are that the first word that popped into your head upon reading this string was 'mat', and that this was your guess. If you thought about other words that could occur, they were most likely all objects that small furry animals can sit on. You wouldn't, for example, have guessed the word 'ceiling'.

Continuing the game, I let you know that 'ceiling' *is* the next word in the string. You may think that this is a silly example, for everyone knows that cats don't sit on ceilings (apart from the fictitious variety).[3] Even so, this sentence is perfectly grammatical according to the rules developed by linguists [14].

The surprise you experience upon discovering the next word in the sequence is a useful measure of how accurate your guess was. If you were reading a novel and experienced a similar level of surprise, you would probably go back and re-read a portion of the text to make sure you hadn't made a mistake. When you read a book, it is thought that you make guesses about which words you are most likely to read next, and that you confirm these guesses by skimming the text with your eyes. Tests by Kutas [39] suggest that this is indeed the case.

[3] Fictitious cats, that is.

Imagine giving the same string to a computer program instead of a human being, and imagine that the program uses a grammar developed by linguists to make its predictions. As far as the program knows, the sentence is grammatical and therefore correct. Without any knowledge about the world, it is unable to disbelieve the sentence as you probably do.

It seems that knowledge about the world of discourse is important when making predictions about which words may occur in certain contexts.[4] A predictor with knowledge about the world of discourse, such as a human being, knows the words that occur after the string 'The cat sat on the' should represent some sort of object that small domestic animals may sit on. The computer program (or the linguist) make no such distinction. The best that they can do is say that a noun-phrase must come next, and that 'nervous green idea' is just as likely as 'mat'. This is due to the fact that linguistics is concerned with the form of a sentence, and not its meaning [14]. This isn't to say that we can't construct a grammar that accepts 'mat' and rejects 'nervous green idea' for the example above, but merely suggests that this is not what is commonly done.

A solution to this dilemma is to incorporate probabilistic information into the language model used by the program, in which case the language model is a stochastic grammar. Hence the program could make a better guess about which words are likely to follow a given string, as it could give its prediction in the form of a probability distribution over all possible words. In essence, this is the approach commonly used for word prediction.

To determine a stochastic grammar, the computer program could be trained on a large amount of sample sentences from the language. This would allow the program to approximate probability distributions of words in the language from their frequency of occurrence. A program that does this is inferring the language model from the text.

[4]This knowledge is not essential, though. Human beings are able to infer much about the meaning of words in nonsensical text – *The Jaberwocky* by Lewis Carol is an example.

2.2 Natural Language Grammatical Inference

To play the Shannon Game, the predictor requires a language model[5] of some sort. This model may be designed by hand, or it may be inferred. Because predictions should be ranked by their likelihood, a stochastic grammar is desirable.

An m^{th}-order Markov model is typically used for word prediction, and may be easily approximated from a finite set of sample sentences. A slightly modified 2^{nd}-order Markov model is used by IBM in their trigram predictor [34].

A popular belief in modern linguistics is that human beings do not infer grammars *per se*, but rather adapt a universal grammar that is present in their brains at birth to their language environment [14]. This linguistic theory is used to give an explanation of why human infants appear to learn language so quickly, given a sparse collection of sample sentences.

My aim is to show that Natural Language Grammatical Inference is possible from scratch, without any major form of innate knowledge being necessary. I intend to do this by designing a solution to the word prediction problem that incorporates a certain degree of Grammatical Inference.

As I have shown, the computer program is at a distinct disadvantage as it has only one source of knowledge – the text. Human beings, on the other hand, possess knowledge about the world they live in, and this may aid their language acquisition abilities. The tests performed by IBM indicate that human players typically score one-third of the perplexity of computer based predictors when playing the Shannon Game [34].

2.3 Why it is Important

I have presented the word prediction problem by playing the Shannon Game, and have mentioned that it is possible for computer programs to participate in this game

[5] In this dissertation, I use the terms *Language Model* and *Grammar* interchangeably, though the convention is normally to use *Language Model* only when referring to stochastic grammars. I do this because I am, in most cases, referring to stochastic grammars throughout.

in addition to human beings. What I have neglected to do so far is mention why I am interested in automating the Shannon Game in the first place.

We know that human beings use contextual information to make predictions about what is likely to occur in the short term future. This is sufficiently illustrated through the use of the Shannon Game in the previous section, and by the IBM tests [34]. As a further example, consider the tests performed by Kutas, where it was found that a significant amount of increased electrical activity in the brain resulted when the subject was given syntactically correct but semantically nonsensical sentences to read, such as 'He took a sip from the transmitter'. Kutas suggests that the presence of a late negative wave in the brain's electrical activity indicates the interruption of ongoing sentence processing by the semantically nonsensical word, and a subsequent re-processing of the sentence by the brain [39].

This project is important because it is exploring possible solutions to a problem that human beings find trivial, and a good solution would allow computer programs to process information in a more natural manner. Additionally, I believe that the language models currently used for word prediction, such as the trigram model employed by IBM and others, may be improved. And if the language models can be improved, then so too can the applications that use language models.

To illustrate the importance of this project by example, I devote the remainder of this Chapter to the presentation of some of the many applications that may employ language models.

Linguistic Knowledge

Currently, linguists develop grammars for languages that are approximations to English using a combination of their own language knowledge and black magic. Such grammars are useful, but the assumptions made in constructing them often defeat the purpose of their construction.

Natural Language Grammatical Inference may be used to develop a more appropriate grammar for the English language, without having to make any prior assumptions about its structure. Such a grammar would have the potential to make a contribution to linguistic knowledge, and shed some light on how Natural Languages work.

Machine Translation

First suggested by Warren Weaver [61], statistical machine translation entails the automatic translation of text written in one language to text written in another, using statistical information derived from sample corpora in each language.

Language models were presented as a solution to the machine translation problem by IBM [11, 12], where they were used to give a probability that an arbitrary English sentence was the translation of an arbitrary French sentence. That is, every English sentence was considered to be a possible translation of the given French sentence, and the language model[6] was used to estimate the probability that a given English sentence was the correct one.

Recognition Systems

Language models may be used to improve the accuracy of speech recognition systems, and IBM have done a lot of work in this area, reporting good results [10].

Other recognition systems may also benefit from this approach. Some examples are handwriting recognition systems and optical character recognition systems. In all cases, the recognition system may be unsure about which word actually occurred, and may use the predictions suggested by the language model to select between several possibilities.

An example is in the spoken sentence 'I read the red book'. The speech recognition system may be unable to differentiate between the words 'read' and 'red' based on an acoustic match only. A language model, however, may predict that 'read' is more likely to follow 'I' and that 'red' is more likely to follow 'the'. This process is commonly referred to as *word-sense disambiguation*.

Proof Reading

A human being predicts which words they are likely to see next when reading a novel. If their prediction turns out to be false, the reader may go back and scan a passage

[6] In this case, the language model was the IBM trigram model.

of text more carefully. Likewise, a computer program may proof-read a document by making predictions in much the same way, and highlighting sections of the text where the predictions were inaccurate. The highlighted words may turn out to be miss-spelt or the context they were used in may be grammatically inappropriate.

Typing Speed

The English language has a high level of redundancy. The actual words in a sentence may be represented using a smaller number of letters without a loss of meaning. For example, it is easy for a human being to understand the message below, even though it is missing eight letters.

Cn u rd ths msg?

A word processor which allows the user to enter words in their abbreviated form, and then automatically expands them to their full form, is one application of language models. Such a word processor would have the potential to greatly increase the typing speed of its users.

Disabled users may only be able to manipulate a single switch, so communicating via a computer can be a painstaking task. An application could allow the disabled user to select the next letter or word to be displayed from a list which is continually updated using the language model, and is sorted in order of probability. It has been found that such an application may improve the communication rate of disabled users by up to 50% [58].

Text Generation

Language models may be used to generate random sentences, and examples of this are given in Chapter 6. Perhaps an application could generate grammatically correct sentences to convey some meaning to the user. A database, for example, could use this program to display the results of a search using Natural Language. When incorporated with machine translation, it may be possible to configure the program to the users native language.

Music

It would be an interesting exercise to attempt to infer grammatical structure for a source of data other than Natural Language, such as classical music. Linguists may cringe at the concept that music has a grammar. If you can show them one, however, they may take you seriously.

If it is possible to infer a grammar from music, then it could be used to generate random compositions. The merit of such an application is debatable, but it sounds interesting.

Image Processing

A language is defined by a set of rules that determine valid strings of symbols. The symbols may be anything at all, from letters and words to musical notes and pixels on a computer screen.

A program attempting to find a square in a picture will look for four straight lines that are connected in a certain way. In this context, we may consider the straight lines to be symbols, and the square to be a valid 'string' of some language [1].

On another level, the pixels in the image may be the symbols, and the straight lines may be a valid string. This is analogous to the English language, where words are strings of letters and sentences are strings of words. The phenomenon whereby the strings of one level become the symbols of the next is common in languages, and this hierarchical method of decomposition is important when performing pattern recognition.

Returning to the square-detector, a language model may be used to predict where the next line is expected to be based on the previous lines it has seen. The surprise experienced by the program could be used as an indicator of how well a collection of lines represent a square. Alternatively, the program could estimate the probability that an arbitrary collection of lines formed a square, and could therefore detect squares that are partially hidden from view.

Chapter 3

Background and Concepts

The advent of the electronic computer saw its application to a vast array of problems that could only be solved given a large amount of computational power. Many of these problems turned out to be considerably more complicated than first thought. Grammatical Inference received a high level of initial interest, but was quickly abandoned due to this technological *naïveté*. Only recently have desktop computers gained enough computational power to make Grammatical Inference a valid field of research once more [40].

3.1 Language Modelling

Language modelling is the task of the linguist. Before the Chomskian revolution, linguists applied a distributional approach that used the contextual constraints of words as a basis for categorising them [27]. When Chomsky arrived, he brought with him sophisticated arguments against such probabilistic methods, and introduced a more formal approach based on well defined grammars [47]. Traditionally, when an application has required a language model of any sort it has been designed by hand rather than inferred, as this is usually found to give a more accurate model [42].

Formal Language Theory introduces terminology that may be confusing to the layman, because certain terms have loaded meanings. I have endeavoured to use neutral terminology which, though it may be non-standard, will help avoid ambiguity.

An *alphabet* is defined to be a finite set of A fundamental units called *symbols*, out of which structures may be built. The alphabet set is denoted by the Σ symbol. For example, $\Sigma = \{a, b, c\}$ indicates that there are three symbols in the alphabet – a, b and c, and hence $A = 3$. Note that I have previously referred to the alphabet as the vocabulary, as this is the standard term used when speaking of the set of all English words. I will use the more formal term in future.

The *language* is a set of finite strings of symbols from the alphabet, and is denoted by the L symbol. Although the strings must be finite in length, the set of all strings (i.e. the language) may or may not be finite. The language set is specified by a *grammar* for the language.

An example may help to clarify the above terminology. If we consider English text on the letter level, the alphabet is the set of all letters and the language is the set of all words. Similarly for English text on the word level, the alphabet is the set of all words and the language is the set of all grammatical sentences.[1]

The language set may be finite or infinite. A *finite language* consists of only a finite number of strings, while an *infinite language* contains an infinite number of strings. English language on the letter level is finite, as there are only a finite number of words. Linguists assume that English language on the word level is infinite, (i.e. that there are an infinite number of sentences), but this point is debatable. The main reason for this assumption is that infinite languages are easier to model, as recursive structures may be used, and not that speakers of English are capable of parsing sentences which are billions of words long [15].

A grammar provides a specification for the strings in the language. A string that is in the language is a *valid string*, while a string that isn't in the language is an *invalid string*. A *recognition grammar* is able to test the validity of a given string. That is, given an arbitrary string of symbols from the alphabet, the recognition grammar may be used to determine whether the string is in the language or not. A *generative grammar* is able to generate valid strings automatically.

As an example of the concepts presented in this section, consider the simple Finite Automaton given in Figure 3.1.

[1] When I refer to the 'letter level', I mean that the symbols in the alphabet set are English letters. Likewise, 'word level' means that the symbols in the alphabet set are English words.

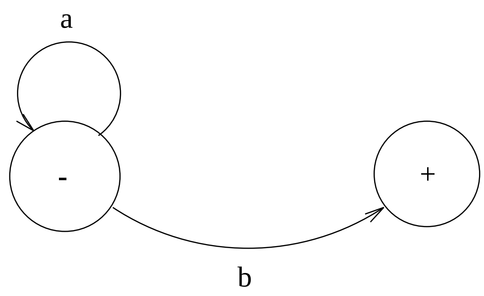

Figure 3.1: A simple finite automaton

In this simple case, the alphabet $\Sigma = \{a, b\}$, and the language is infinite.[2] The language set $L = \{b, ab, aab, aaab, aaaab, \ldots\}$. That is, the language contains strings beginning with zero or more a's and ending with a single b. The language contains no other strings.

Formal Language Theory is concerned with language syntax only. That is, the semantics or meaning of sentences (even English sentences) is not taken into account. This is why we saw that a program that uses a formal English-like grammar may parse the sentence 'The cat sat on the nervous green idea' without so much as a hiccough.

A Formal Grammar for English[3] may be very much like the grammar illustrated above, only a whole lot more complex. It may be another type of grammar that is more powerful than the one shown above, and it may be a *stochastic grammar* so that each string it generates or accepts is assigned a probability of occurrence.

When we think of English, we think of nouns and verbs, adjectives and conjunctions, pronouns and adverbs. A grammar for English may use these syntactic categories in a hierarchical fashion. For example, it may have a rule stating that a verb-phrase is followed by a noun-phrase, and it may define a noun-phrase as an adjective followed by a noun. Figure 3.2 illustrates the hierarchical generation of a simple sentence [15].

Many modern linguists approach the problem of language acquisition with the assumption that a universal grammar is present in our brains at birth, and that we

[2] The recursive loop in the Automaton allows an infinite number of valid strings to be generated.
[3] Or, more properly, pseudo-English.

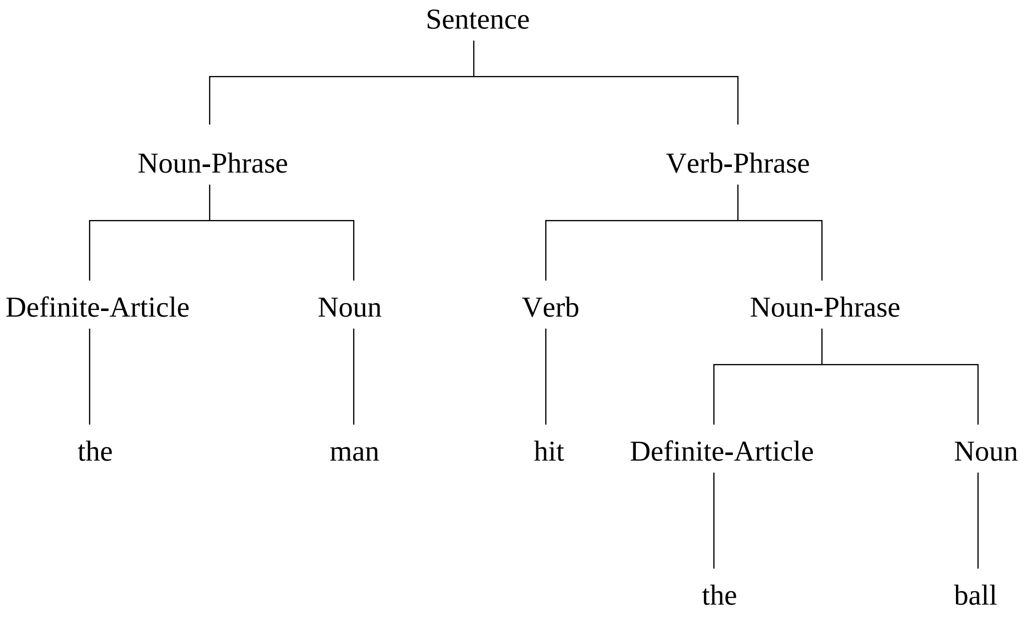

Figure 3.2: Generation of a valid string by a simple hierarchical grammar

learn a language by adapting this grammar to it. The universal grammar is considered generic enough to be applied to any human language, and is assumed to contain structures that are common across all human languages. The notion of Grammatical Inference for Natural Languages is strictly rejected.

Shannon has shown that a *Markov model* may be used to generate a language that is an approximation to English [51]. A Markov model is a stochastic recognition grammar, and consists of a finite number of states. A Markov model of order m has m^A states, where A is the number of symbols in the alphabet. The m^{th} order Markov model effectively 'remembers' the previous m symbols. A diagram of a simple 2^{nd}-order Markov model is given in Figure 3.3.

In the Markov model, each transition is governed by a probability of occurrence. Hence if the model is in state aa, the probability of going to state ab is $p(b|aa)$. This probability is conditional on the previous two symbols, represented by the state aa.

We may alter the Markov model so that it emits a symbol whenever a state transition is initiated. This gives the model the ability to generate strings of symbols, and thus turns it into a stochastic generative grammar.

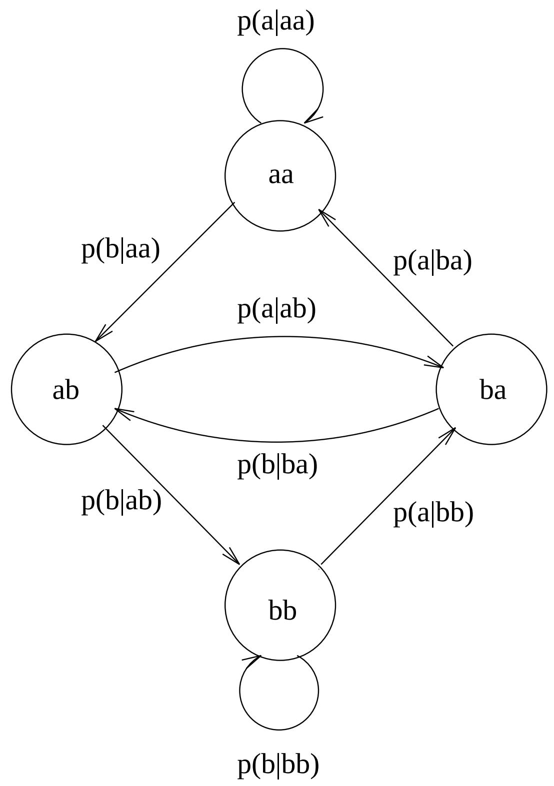

Figure 3.3: A 2^{nd}-order Markov model

Shannon's approximations to English worked by constructing an m^{th}-order Markov model for English on either the letter level or the word level. He did this by either using a table of the appropriate frequencies, or by using an equivalent method which entailed flipping through a book [51]. In the latter case, he found an approximation to the book's language. An example of one of the strings Shannon generated when using a 2^{nd}-order Markov model on the word level is shown in Figure 3.1.

THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER THAT THE CHARACTER OF THIS POINT IS THEREFORE ANOTHER METHOD FOR THE LETTERS THAT THE TIME OF WHO EVER TOLD THE PROBLEM FOR AN UNEXPECTED

Figure 3.4: Shannon's 2^{nd}-order approximation to English

As we intuitively expect, Shannon showed that as the order of the model increases, the language generated by it more closely approximates the original text [50].

Using this type of stochastic grammar as an approximation to English is strictly rejected by Chomsky and the Transformationalists, for reasons that will be discussed in the next two sections.

3.2 Grammatical Inference

In [14], Chomsky explains that

> ...linguistic theory is concerned primarily with the ideal speaker-listener, in a completely homogeneous speech-community, who knows its language perfectly and is unaffected by such grammatically irrelevant conditions as memory limitations, distractions, shifts of attention and interest, and errors in applying his knowledge of the language in actual performance

Clearly these assumptions do not hold when we are concerned with the language that ordinary people use when conversing. Indeed, it is debatable whether a perfect grammar exists for English at all, as the language itself evolved from a general mish-mash of other languages, each of which possessed certain idiosyncrasies. Unfortunately for the linguist, English has all of them.

To avoid the linguistic bias that results when formal grammars are designed by hand, and to capture a more subtle and complex structure than may be apparent to the linguist, *Grammatical Inference* may be used. Figure 3.5 shows the simple model of a Grammatical Inference Engine proposed by Fu [22].

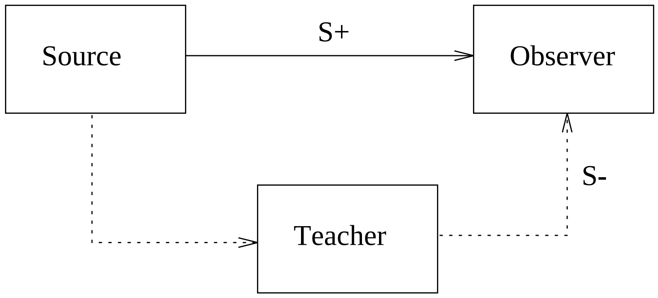

Figure 3.5: Structure of a Grammatical Inference Engine

In the Figure, $S+$ represents a set of valid example sentences, while $S-$ represents a set of invalid example sentences. The valid sentences may be taken from samples of the language itself, while the invalid sentences must be formulated by an external teacher who has some additional knowledge about the language [22].[4]

Given a set of strings that the grammar is supposed to generate, the Grammatical Inference problem is one of inferring a grammar that satisfies these strings, and is also able to generalise to other unseen strings [40]. Traditionally, Grammatical Inference has been applied to formal language models, either non-stochastic [22] or stochastic [23]. This allows the quality of the Inference Engine to be tested, as in many cases the optimal grammar is known.

Even so, it has been proven that there exist very simple grammars that cannot be inferred given a finite subset of positive data only [25]. It seems, then, that inferring a grammar for Natural Languages may not only be difficult, but downright impossible. In most cases however, we are interested in inferring a grammar that suitably approximates the language, and not the optimal grammar [40]. Indeed, spoken English has

[4]The external teacher is usually a native speaker of the language, and is often the linguist himself.

a loose grammatical structure not easily defined, and understandable utterances may be ungrammatical according to linguistic conventions [31]. We may therefore wish to infer a grammar that can tolerant ungrammatical (or noisy) strings, particularly if we are interested in using the inferred grammar as a language model for speech recognition.

The difference between Grammatical Inference and the Chomskian approach to language acquisition is shown in Figure 3.6, and was originally given by Chomsky [15].

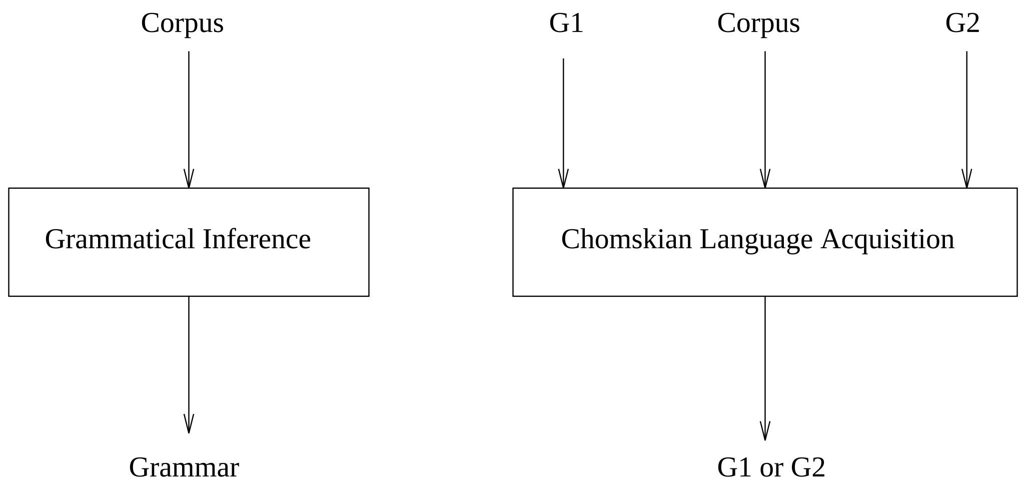

Figure 3.6: Grammatical Inference versus Chomskian Language Acquisition

Grammatical Inference returns a grammar when given a training corpus, while Chomsky's method entails selecting the best grammar from a set of candidate grammars when presented with the same training corpus. Chomsky rejects the notion that the grammar may be inferred, and instead proposes that suitable grammars are known to the system *a priori*.

As I mentioned in the beginning of this Chapter, Grammatical Inference is a computationally expensive problem. This has been the main factor that has inhibited its intense investigation [40].

3.3 Word Prediction

The word prediction problem, or the Shannon Game, makes use of a grammar[5] to determine the symbols that are most likely to appear next based on the current context.

Human beings are experts at the Shannon Game, and beat the best computer implementations by a factor of three or more [34]. Consider how proficient we are at deciphering bad handwriting, or maintaining a conversation with an elderly relative over a bad telephone connection. In both these cases, we use contextual information to guess at words that were indecipherable or unheard.

Chomsky's most convincing argument against probabilistic models is drawn from language acquisition data. It is noted that in many cases the dependencies between words in a sentence may be such that probabilistic models of low order are unable to take them into account. For example, consider the rather morbid sentence given below.

> The man who used to live down the street from us for about five years shot himself yesterday afternoon.

In this sentence, the seventeenth word 'himself' depends on the second word 'man'. When predicting what comes after 'shot', we would do better if we remembered that we are talking about a man. Markov models of order less than fifteen are unable to take this useful piece of information into account.

Another Chomskian argument against probabilistic models is that nonsensical grammatical sentences and ungrammatical sentences may be rejected on the same grounds by a probabilistic model, in that they are both equally remote [15]. A grammar should be able to identify the first sentence as grammatical[6] and the second as ungrammatical.

Clearly, it is desirable to take these arguments into account when developing a word prediction system. The first problem cited by Chomsky may be avoided by upwriting

[5] Which may or may not have been automatically inferred.
[6] Grammatical, that is, but highly unlikely.

and chunking the text, and I will discuss these concepts in the following Chapter. The second argument is of more concern, but may not hinder us at all if we consider spoken English to be only semi-grammatical.

3.3.1 The IBM Trigram Model

The most widely used language model for speech recognition was developed by IBM for their Tangora speech recognition system [8, 4, 10, 34, 33, 7, 5, 35, 6]. The *IBM trigram model* is derived from a Markov model for English text.

An n-gram is a string of n symbols. In the IBM trigram model, all consecutive strings of three symbols are extracted from a training text, and a count of how many times each of these trigrams occur is kept. All of the trigrams with the first two symbols matching are grouped together. Because the number of occurrences of each trigram is known, the relative frequencies of the third symbol may be calculated.

This trigram data effectively determines a 2^{nd}-order Markov model, with transition probabilities approximated by the calculated frequencies.[7] This is essentially exactly what Shannon was doing with his 2^{nd}-order approximation to English.

The motivation behind the trigram model of language is that it is based on the approximation given in Equation 3.1.

$$p(w_i|w_1, w_2, \ldots, w_{i-1}) \approx p(w_i|w_{i-2}, w_{i-1}) \tag{3.1}$$

That is, it is assumed that all previous words determine which word will occur next, and that this may be approximated using the two previous words only [8].

To predict which symbols may follow any given sequence of two symbols is a simple process of finding the matching trigrams and giving the prediction as the calculated frequency vector. Unfortunately, it was found that in a new text, a large percentage (around 23%) of trigrams deduced from a 1.5 million word training corpus hadn't been seen before, and the model was unable to predict possible sequents for these unknown trigrams [32].

[7] An n-gram model is therefore equivalent to an $(n-1)^{th}$-order Markov model.

To cater for this, the IBM model was changed to *fallback* to bigram and even unigram distributions if the trigram hadn't been encountered before. So if the trigram 'nervous green' hadn't been seen in the training text, the bigram 'green' would be used to make the prediction. And if this bigram hadn't been seen either, then the word frequencies themselves would be used.[8]

A slightly better prediction was found to result if, instead of falling back only when needed, the trigram, bigram and unigram distributions were *interpolated*, as in Equation 3.2. Note that the interpolation weights are chosen such that $\alpha + \beta + \gamma = 1$.

$$p(w_3|w_1, w_2) = \alpha f(w_3|w_1, w_2) + \beta f(w_3|w_2) + \gamma f(w_3) \tag{3.2}$$

This interpolation technique produces a *smoothing* of the predicted probability distribution, with the result that more words in the alphabet are assigned non-zero probabilities by the predictor. The drawback to this interpolation model is that the optimal values of the weights α, β and γ may be expensive to find [34].

A further amendment to the model that was found to improve its performance was the addition of word equivalence classes. Each word in the corpus was *tagged* with the class it belongs to,[9] and this additional information was used when making the predictions.

It is possible to infer these equivalence classes automatically [47] by making use of the linguistic replacement rule 'Can a word be replaced by a word of known type? If so, it is a word of that type' [45]. A context window is used to determine which words are valid replacements for a given word. This is in effect equivalent to the distributional approach applied in linguistics before the Chomskian revolution.

[8]Here we encounter a major drawback of the IBM model. Fallback to the unigram level will always cause the most frequent word to be predicted, which is in most cases the word 'the'. This is clearly absurd.

[9]For example, a *verb* may be a class.

Chapter 4

The Upwrite Predictor

Before beginning this Chapter, I must define what I mean by a predictor. A predictor is a prediction algorithm that gives its prediction in the form of a probability distribution, which may be considered to be a vector of real numbers which sum to one. The knowledge that the predictor requires may be gained only from a training corpus,[1] while the Shannon Game is played using a different testing corpus. These constraints alone present many problems which must be resolved when designing an effective prediction algorithm.

One such problem is that a large proportion of the n-grams inferred from a Natural Language training corpus will have occurred only once in the corpus, even for small n. Similarly, a large proportion of symbols from the alphabet will have occurred only once in the training text. Kupiec found that over 40% of the words in the 1 million word English corpus he was using occurred only once [38].

A further problem is that quite a large percentage of the n-grams in a testing corpus will not have occurred in the training corpus at all. IBM, for example, found that 23% of the trigrams in their testing corpus of 1.5 million words were absent in their training corpus of 300 thousand words [32]. Furthermore, often the testing corpus will contain words that weren't even in the training corpus.

The ideal predictor would be able to generalise its language model to overcome the sparseness of n-gram distributions, smooth its prediction to account for unseen n-

[1] This is the Grammatical Inference process.

grams, execute adaptively[2] to minimise the effects of new symbols, and infer a hierarchical stochastic grammar to overcome Chomsky's argument against automatic inference.

The *Upwrite predictor* overcomes many of the drawbacks of the IBM predictor, and provides better solutions to the problems mentioned above [2]. I use the Upwrite predictor in this project, and present the necessary algorithms for Grammatical Inference and prediction in this Chapter. A simple example of the predictor is given in Section 4.6, which should be referred to when necessary to clarify any of the introduced concepts.

4.1 Statistical Analysis

As with the IBM predictor, the Upwrite predictor begins by inferring an $(n-1)^{th}$-order Markov model for the text. This is achieved by an algorithm that performs statistical analysis on a training corpus. To maintain compatibility with the IBM predictor, I decided to use a 2^{nd}-order Markov model, though this need not be the case. An n-gram model for $n = 3$ seems appropriate, though, as the number of distinct n-grams goes up quickly with n, and the chances of seeing the same n-gram twice diminish rapidly.

The predictor should be able to work on any language, and so the symbols used in the Markov model should be selectable by the user.[3] For example, if English text is being used, the predictor may operate on the letter level or on the word level. For this reason, I will talk about symbols from the text, where the symbols may be letters, words, or any other structure the user deems appropriate.

Smith suggests that if there is no theoretical basis for assigning probabilities to the Markov model, then the probabilities may be estimated by their relative frequencies [53]. This is achieved by collecting sequences of n symbols from the text, and partitioning these n-grams into groups based on whether their initial string of $(n-1)$

[2] An adaptive predictor continues to learn as it performs the prediction process.

[3] The order of the Markov model should also be user-selectable, though I generally assume that it be 2^{nd}-order.

symbols are identical. Each of the resulting contexts[4] is associated with a vector of counts for the n^{th} symbol.

Assuming trigrams, the analysis algorithm collects strings of three symbols w_i, w_j and w_k, and partitions these trigrams into groups that share the same first two symbols. For each of these contexts $\langle w_i, w_j \rangle$, a count vector $c_{i,j}$ records the number of occurrences of each symbol of the alphabet in the third position w_k. This vector therefore has A elements, one for each symbol in the alphabet.

The probability distribution of symbols appearing after the context $\langle w_i, w_j \rangle$ may be estimated by calculating the frequency vector $f_{i,j}$. This vector is simply a normalised version of $c_{i,j}$, such that the k^{th} element in the vector is given by Equation 4.1. Note that $f_{i,j}$ is a vector in $\Re^A$ and that its elements sum to one.

$$f_{i,j}[k] = \frac{c_{i,j}[k]}{\sum_{x=1}^{A} c_{i,j}[x]} \tag{4.1}$$

The vector $f_{i,j}$ provides an estimation of the conditional probabilities for the state of the Markov model that corresponds to the context $\langle w_i, w_j \rangle$. The conditional probabilities may be calculated from the frequency vector as shown in Equation 4.2.

$$p(w_k | wi, w_j) \approx f_{i,j}[k] \tag{4.2}$$

Thus the statistical analysis is able to infer a Markov model that can generate approximations to the original training corpus. As the size of the training corpus grows, the inferred relative frequencies will approach the actual conditional probabilities, provided that the Markov model sufficiently approximates the text [53].

4.2 Entropic Chunking

It is undesirable to provide the predictor with innate knowledge about what constitutes a word in English text. It would be better if the predictor could perform statistical analysis on the lowest level,[5] and use the inferred Markov model to discover words for itself. This is indeed possible, using a process known as *entropic chunking*.

[4] I call the initial string of $(n-1)$ symbols a *context*.

[5] The lowest level in English text may be considered to be the letter level. If we are looking at images of text then it is the pixel level.

Entropic chunking works by using a Markov model for the lowest level symbols in a text. The algorithm runs through the text, estimating the entropy H for each of the symbols in the text by using the context that each symbol appears in.[6] When this entropy is found to exceed a certain, user-defined threshold, H_{thresh}, a separator symbol is deemed to have been found and the group of symbols between two of these separator symbols are chunked into 'words'. This chunking process is also referred to as an *S-Upwrite*,[7] as sequences of symbols are being *upwritten* to a single, higher level symbol.

As an example of this process, consider the segment of English text given below, which is the first three sentences of 'The Adventures of Sherlock Holmes', by Sir Arthur Conan Doyle.

> To Sherlock Holmes she is always the woman. I have seldom heard him mention her under any other name. In his eyes she eclipses and predominates the whole of her sex.

Statistical analysis was performed using the entire Sherlock Holmes text on the letter level. Using entropic chunking, with a judicious choice of H_{thresh}, the segment of text shown was chunked into symbols to give the new text below.[8]

> (To)(Sherlock)(Holmes)(she)(is)(always)(the)(woman.)(I have)
> (seldom)(heard)(him)(mention)(her)(under)(any)(other)(name.)
> (In)(his)(eyes)(she)(eclipses)(and)(predominates)(the)(whole)(of)
> (her)(sex.)

The text has been divided fairly successfully into English words, using just the statistical properties of strings of three letters from the training corpus. This process was done completely automatically, without user intervention, and with no innate knowledge about the language being required.

It is also possible to chunk the new text. Indeed, performing multiple levels of chunking is important when deriving a hierarchical language structure. Performing a second chunking pass produces the text shown below.

[6]Recall that the entropy is a measure of the uncertainty that the predictor has about what is coming next, and was defined in Equation 2.10.
[7]The S in S-Upwrite stands for *Sub-Object*.
[8]Programs were written to perform the statistical analysis and chunking processes.

> (To)(Sherlock Holmes)(she is always)(the woman)(. I)(have)(seldom heard him mention her under any other)(name. In)(his eyes)(she eclipses and predominates the whole)(of her)(sex.)

In this case, the algorithm has discovered how to chunk groups of words into 'sub-sentences'. Note that 'Sherlock Holmes' has been chunked into one symbol, as has 'the woman' and 'his eyes'. These chunks may be likened to some of the structures in a hierarchical decomposition of English, as presented in Figure 3.2. For instance, 'his eyes' is an example of a noun-phrase.

I have shown that entropic chunking may be used by the predictor to successfully discover words from a text without any innate knowledge being required. The text may be upwritten[9] to a sequence of words rather than a sequence of letters. This upwrite process may continue in a hierarchical fashion, until only a single entity remains.

Unfortunately, I was unable to incorporate the chunking process into the final implementation of the predictor due to computational overheads. This Section was merely intended to show that it is possible for a program to identify words in a text by itself, without any innate language knowledge whatsoever. Given that it is possible to do this, hard-wiring knowledge about words into the program can be justified.

4.3 Clustering

The statistical analysis determines a collection of vectors in $\Re^A$ that represent probability distributions.[10] Vectors that are close in the alphabet space represent probability distributions that are similar. If groups of distributions can be found, then they could be *clustered* into a single distribution that may occur in many different contexts.

I will commonly refer to this process as a *Q-Upwrite*, where Q stands for *Quotient-Object*, as the symbol is being upwritten to the cluster of which it is a member.

[9] Here I am referring to the *S-Upwrite*, rather than the *Q-Upwrite* which will be presented in a following section.

[10] Recall that $A = |\Sigma|$ is the number of symbols in the alphabet Σ.

The upwritten distributions represent a syntactic category, or an equivalence class, of symbols.

To find equivalence classes, groups of vectors that lie close to one another must be aggregated to allow them to be upwritten to a single cluster. Consider for example the data given in Figure 4.1, where three distinct clusters of points are visible. This data represents the height and weight measurements of three different types of object.

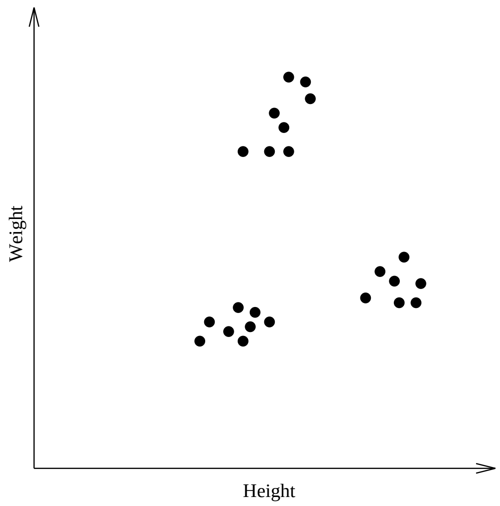

Figure 4.1: Two dimensional data with three distinct clusters

Clustering algorithms use some sort of similarity measure to find groups of objects that are similar to each other. In the case where the objects are vectors in some space, one of the possible similarity measures is the distance that two vectors are separated by [28].

I will only be concerned with clustering algorithms that operate on vectors of real numbers, using a distance metric as a similarity measure. It is questionable which distance metric to use in a large dimensional space, particularly because it is virtually impossible to 'eyeball the data'. I will use the Euclidean metric, given in Equation 4.3, unless a particular algorithm requires a specific metric. Note that C_j is the centroid[11] of the j^{th} cluster, while x is the data vector.

$$D_j = \|C_j - x\| = \sqrt{\sum_{i=1}^{A}(C_j[i] - x[i])^2} \tag{4.3}$$

[11] The *centroid* of a cluster is the vector that lies in the exact centre of the cluster.

I initially used the *dog-rabbit* clustering algorithm [41] because of its origins in the modelling of biological neural behaviour, and its demonstrated superiority over more traditional algorithms. The dog-rabbit algorithm tends to be quite slow, so when speed is of the essence I use the less elegant but more efficient *k-means* clustering algorithm.[12]

4.3.1 The Dog-Rabbit Clustering Algorithm

The dog-rabbit clustering algorithm has an ancestry in work on modelling biological neuron behaviour, and overcomes many of the deficiencies of traditional clustering techniques [41].

Initially, a set of cluster centroids are initialised to random positions in the data space. The algorithm sequentially moves these centroids towards the data vectors, favouring vectors that occur in clusters.

During a single iteration of the algorithm a data vector is chosen at random, and each of the centroids is moved towards this data vector by an amount given by the appropriate movement equation. The distance moved by the closest centroid is given in Equation 4.4, where the closest centroid is determined via the Euclidean metric in Equation 4.3.

$$C'_j = C_j + \frac{2D_j}{(1+D_j)^{v_j}}(x - C_j) \tag{4.4}$$

The movement of the remaining centroids is inhibited by an amount proportional to their distance from the data vector, modelling lateral inhibition in biological neurons. The relevant movement equation is given in Equation 4.5, where Λ is the inhibition constant.

$$C'_j = C_j + \frac{D_j}{\Lambda + D_j}\frac{2D_j}{(1+D_j)^{v_j}}(x - C_j) \tag{4.5}$$

In both movement Equations, v_j is the *fatigue* of the centroid. As the value of the fatigue increases, the centroid will move by a lesser amount. This property of the movement equation models neural habituation. The fatigue of each centroid is initially set to 1, but is incremented whenever the centroid comes within a certain, user-defined radius of the data vector x.

[12] Actually, I devised a clustering algorithm myself for this purpose, which I later discovered to be the k-means clustering algorithm.

This process is iterated until the amount moved on average by the centroids reaches some lower bound. Centroids that failed to converge to a cluster may be easily pruned at the end of the process, and each data vector may be assigned to the cluster corresponding to the closest centroid. Any data vectors that don't fall close to a centroid are left un-clustered.

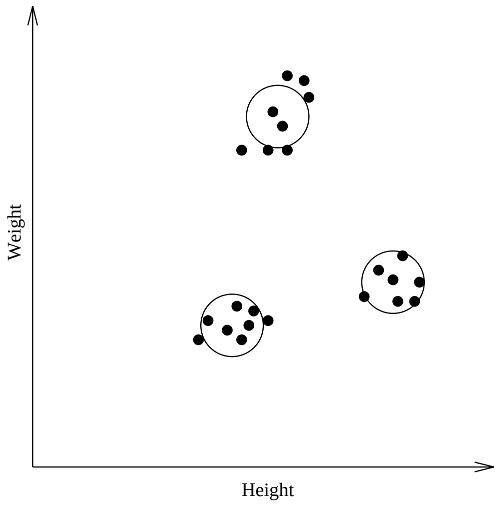

Figure 4.2: An example of data clustered tith the Dog-Rabbit algorithm

In Figure 4.2, I give the result of executing the dog-rabbit clustering algorithm on the data originally presented in Figure 4.1, using three centroids. Note that the algorithm has successfully found all three of the clusters.

4.3.2 The K-Means Clustering Algorithm

The k-means clustering algorithm begins by assigning the initial position of each centroid to data vectors chosen at random. Once this has been done, a data vector x is selected, and its distance from each centroid is calculated using the Euclidean metric.

The closest centroid is found, and its position is updated if it is within some user-defined distance from the data vector. Assuming that the centroid has come closest to K data vectors in the past, its new position will be calculated as the mean of all vectors it has come closest to, as shown in Equation 4.6.

$$C_j' = \frac{KC_j + x}{K + 1} \tag{4.6}$$

This process is iterated until the centroids remain at a fairly constant position, at which stage each data vector is assigned to the cluster corresponding to the closest centroid, with no assignment being made when this distance exceeds the defined cluster radius.

Figure 4.3: An example of data clustered with the K-Means algorithm

The application of the k-means clustering algorithm on the data in Figure 4.1 is shown in Figure 4.3, using three centroids. In this example, only two of the three clusters were found. The larger of the two clusters was split into smaller clusters by the algorithm.

4.4 Upwriting

One way of inferring a hierarchical grammar is to *upwrite* symbols and sequences of symbols in the text to higher order symbols, as the example given in Figure 4.4 demonstrates. Each upwrite performed acts as a *compressor* for the text, as it reduces the number of distinct symbols in the text. The final grammar may be considered to be a compressed version of the training text, or at least an approximation to it.

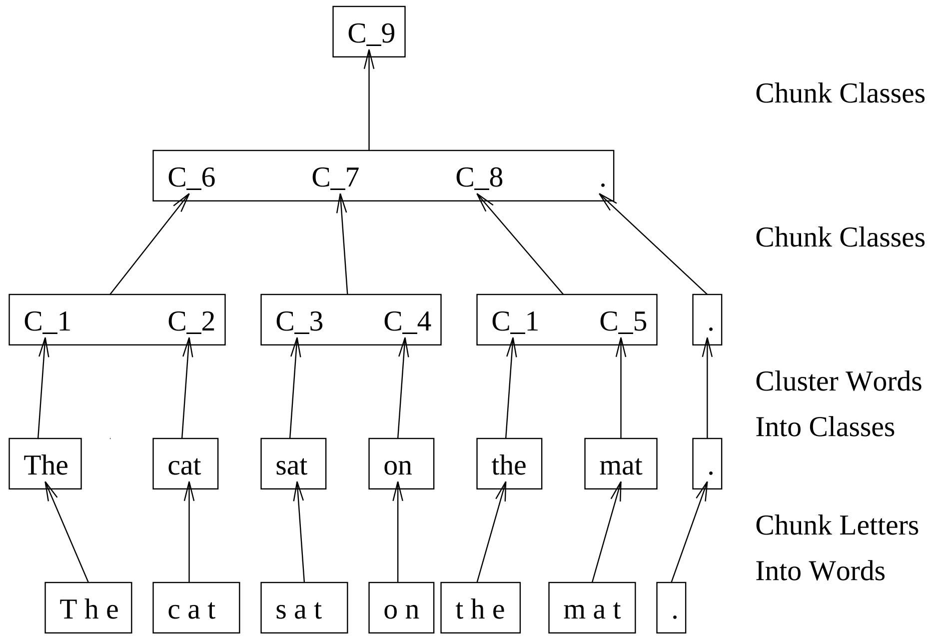

Figure 4.4: An example of the Upwrite process

Once some clusters have been found in the space, the next stage is to upwrite the original source text using the syntactic categories found. This process aims to reduce the number of symbols in the alphabet, and hence the dimensionality of the data. Dimensionality reduction has been shown to improve the performance of language models [19].

To upwrite a symbol in the text, the context that the symbol appears in is used. The symbol is upwritten to the cluster which the distribution of the context has been assigned to, or to the frequency vector if no suitable cluster exists.[13]

If the context $\langle w_i, w_j \rangle$ hasn't been encountered before, then the symbol cannot be upwritten. In this case, the Q-Upwrite algorithm falls back to the bigram context $\langle w_j \rangle$ to perform the upwrite. If the bigram context hasn't been seen either, then the symbol is left alone.

I made a slight modification to this process to cater for the fact that the implemented predictor is non-adaptive. In the modification, if the bigram context hasn't been seen before, one of two events may occur. If the symbol has been seen before, it

[13]The frequency vector is assumed to be the only member of a cluster with itself as the centroid.

is left alone. If the symbol is new, however, then it is upwritten to its prediction. This modification was necessary because leaving a new symbol alone would cause the prediction of the next symbol to fallback to the unigram distribution, an undesirable situation.

The ability to infer a hierarchical grammar is desirable, and to do this many levels of chunking, clustering and upwriting are required. The original text may undergo an upwrite process to give a new text, which in turn may be upwritten. This process may be repeated *ad infinitum*, or at least until a single symbol remains.

4.5 Predicting

The Upwrite predictor makes use of the inferred hierarchical grammar. Starting at the highest level of the grammar, contextual information is used to predict a probability distribution for the next symbol in much the same way as the IBM fallback predictor does. This prediction is then *downwritten* to the next lower level, and the process is repeated. The predictor will eventually downwrite to the lowest level, giving the prediction in the form of a probability distribution over the alphabet.

I have chosen to implement only a single Q-Upwrite, and no S-Upwrite, because of the computationally prohibitive aspects of performing multiple upwrites.[14] Hence a symbol is upwritten to a class, and the process stops there. The prediction algorithm gives a probability distribution for the next class, and downwrites this to give a prediction for the next symbol. As an example, consider the text below, with the prediction to be given for the unknown symbol $w_?$.

$$w_g, w_h, w_i, w_j, w_?$$

The predictor uses the context $\langle w_g, w_h \rangle$ to determine the corresponding cluster centroid C_i, and $\langle w_h, w_i \rangle$ to determine C_j. Thus the context $\langle w_i, w_j \rangle$ is upwritten to the context $\langle C_i, C_j \rangle$.

[14]The predictor that I have implemented is therefore the *Q-Predictor*, rather than the true Upwrite predictor.

The predictor must now give a probability distribution for the next class, and downwrite this to give the prediction. This may be done by the equivalent operation whereby the context $\langle C_i, C_j \rangle$ is downwritten to all possible symbol contexts. That is, the context $\langle C_i, C_j \rangle$ is downwritten to the symbol contexts $\langle w_{i1}, w_{j1} \rangle, \langle w_{i1}, w_{j2} \rangle, \ldots, \langle w_{iN}, w_{jN} \rangle$. Each of these symbol contexts has a probability of occurrence, given by Equation 4.7.

$$p(\langle w_{ia}, w_{jb} \rangle) = C_i[a]C_j[b] \tag{4.7}$$

For each $\langle w_{ia}, w_{jb} \rangle$, the corresponding frequency vector $f_{ia,jb}$ is found. The prediction vector $G_{i,j}$ is calculated by taking the weighted sum of all frequency vectors. That is, the final prediction is as given in Equation 4.8.

$$G_{i,j} = \sum_{a=1}^{A} \sum_{b=1}^{A} p(\langle w_{ia}, w_{jb} \rangle) f_{ia,jb} \tag{4.8}$$

One unavoidable deficiency of all prediction algorithms is that they will encounter problems whenever a new symbol appears. To account for this, an additional element is included in the prediction vector, which gives the probability that a new symbol will occur. This probability may be estimated using Turing's approximation as defined Equation 4.9, which states that the probability of a new symbol being encountered is inversely proportional to the number of symbols encountered to date [26].

$$p(w[A+1]) \approx \frac{1}{N+1} \tag{4.9}$$

This Equation determines the curve shown in Figure 4.5. The prediction vector therefore exists in $\Re^{A+1}$, and must be normalised to take the extra element into account.

Even though a new symbol may occur, it will very likely be upwritten by the predictor. It makes sense, therefore, to base Turing's approximation on the fact that the upwritten class is assumed to be the correct one for the new symbol. That is, the N in Equation 4.9 would represent the number of times the upwritten class had been seen before, rather than the number of words encountered. Because of time limitations, I was unable to implement the approximation in this way. Therefore, the probabilities of new symbols are estimated to be quite a bit lower than they should be by my program. When giving results, I avoid this problem by ensuring that the testing corpus contains no words that weren't in the training corpus.

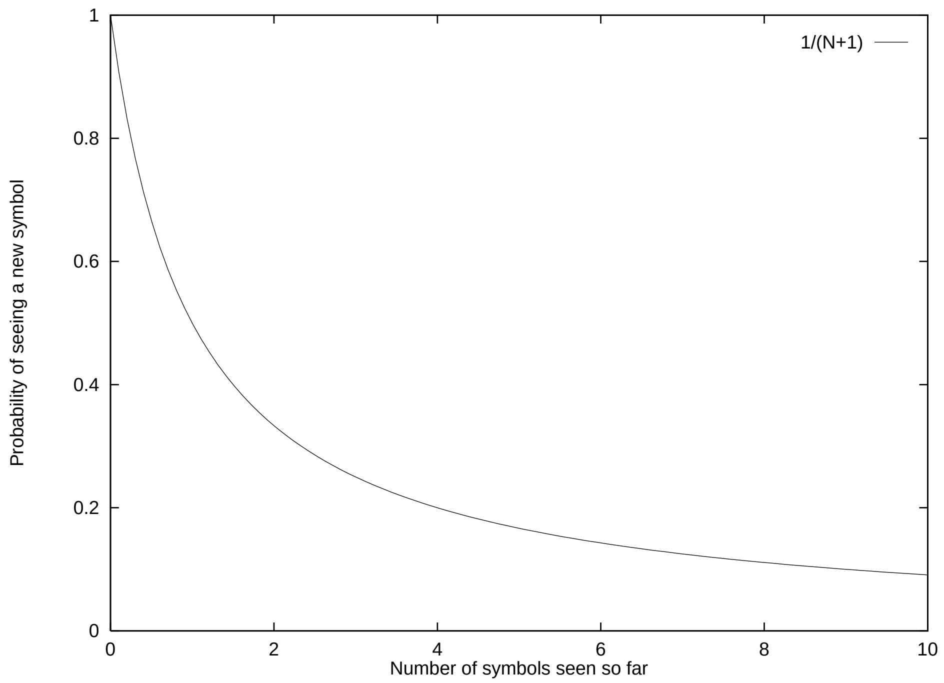

Figure 4.5: Turing's approximation of the probability of a new symbol

In rare cases the predicted distribution may be empty. Fallback may be used in such situations, by which the upwritten bigram or unigram contexts are used to make the prediction. Another solution is to use the interpolation technique, where the predicted vector is the weighted sum of trigram, bigram and unigram predictions.

4.6 An Example

As an example of the Q-Predictor, the text *acdbceacebcfbceacf* will be processed on the letter level to determine a language model. The Hidden Markov Model[15] shown in Figure 4.6 was used to generate this text.

The prediction algorithm will use the inferred grammar to make predictions about the testing corpus *acqbcdaef*, which was designed by hand to contain the unseen valid trigram $\langle d, a \rangle$, the new symbol q, and the glitch symbol e. Note that a glitch symbol is a valid symbol which has been used out of context.

[15]Hidden Markov Models are described in Section 5.1.

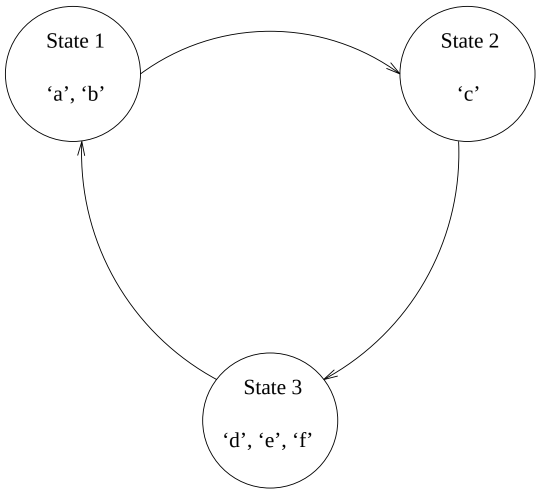

Figure 4.6: The Hidden Markov Model used to generate the training corpus

Statistical Analysis

Strings of three sequential letters taken from the training corpus are divided into groups of strings that share the same first two letters. The third letter is used to determine a count vector for each group of strings. This process gives the count vectors as below. Note that the elements of all the vectors are sorted alphabetically, such that the leftmost element corresponds to the symbol a, and the rightmost element corresponds to the symbol f.

$$
\begin{array}{lll}
c_{a,c} = [000111] & c_{c,d} = [010000] & c_{d,b} = [001000] \\
c_{b,c} = [000021] & c_{c,e} = [210000] & c_{e,a} = [002000] \\
c_{e,b} = [001000] & c_{c,f} = [010000] & c_{f,b} = [001000]
\end{array}
$$

Normalising the count vectors produces the frequency vectors shown below, which are approximations of the conditional probability distribution of sequent letters given the context.

$$
\begin{array}{lll}
f_{a,c} = [000\frac{1}{3}\frac{1}{3}\frac{1}{3}] & f_{c,d} = [010000] & f_{d,b} = [001000] \\
f_{b,c} = [0000\frac{2}{3}\frac{1}{3}] & f_{c,e} = [\frac{2}{3}\frac{1}{3}0000] & f_{e,a} = [001000] \\
f_{e,b} = [001000] & f_{c,f} = [010000] & f_{f,b} = [001000]
\end{array}
$$

This ends the acquisition of trigram statistics, which gives a 2^{nd}-order Markov model for the text. The Q-Predictor requires bigram and unigram statistics also, and these are collected in a similar fashion.

Clustering

A clustering algorithm may be used to find groups of similar frequency vectors. I have done this by inspection, giving the five cluster centroids shown below.

$$
\begin{aligned}
C_1 &= \tfrac{1}{4}(f_{c,d} + f_{c,f} + f_d + f_f) && = [010000] \\
C_2 &= \tfrac{1}{6}(f_{d,b} + f_{e,a} + f_{e,b} + f_{f,b} + f_a + f_b) && = [001000] \\
C_3 &= \tfrac{1}{2}(f_{c,e} + f_e) && = [\tfrac{2}{3}\tfrac{1}{3}0000] \\
C_4 &= \tfrac{1}{3}(f_{a,c} + f_{b,c} + f_c) && = [000\tfrac{1}{6}\tfrac{1}{2}\tfrac{1}{3}] \\
C_5 &= f_\Sigma && = [\tfrac{1}{6}\tfrac{1}{6}\tfrac{1}{3}\tfrac{1}{18}\tfrac{1}{6}\tfrac{1}{9}]
\end{aligned}
$$

Upwriting

The text is upwritten by taking a string of two symbols from the text, say $\langle a, c\rangle$, and upwriting the next symbol to the corresponding cluster, in this case C_4. This process may be done on the entire testing corpus, giving the upwritten text shown below.

$$\text{acqbcdaef} \mapsto \text{a } C_2C_4 \text{ b } C_2C_4C_1C_2C_3$$

The Q-Upwrite process fails to upwrite the first letter of the text, and falls back to the bigram context when upwriting the second letter. This is because at the beginning of the text, the Q-Upwrite algorithm doesn't know what the current context is, as it hasn't seen any letters previously. Fallback also occurs when upwriting the letters that follow the new letter q, and the new contexts $\langle d, a\rangle$, $\langle a, e\rangle$ and $\langle e, f\rangle$.

Predicting

The context $\langle a, c\rangle$ is followed by the new symbol q. The Q-Predictor upwrites the context to $\langle a, C_2\rangle$, as in the previous Sub-section, and gives the prediction for the next class as C_4. This class is downwritten to a distribution over the alphabet, and Turing's approximation is taken into account to give the final prediction as shown below.

$$G_{a,c} = [000\tfrac{19}{120}\tfrac{19}{40}\tfrac{19}{60}\tfrac{1}{20}]$$

This prediction agrees with data seen in the training corpus, as the symbols d, e and f are assigned non-zero probabilities of occurrence. The predicted distribution gives $p(q) = \frac{1}{20}$, as q is a new symbol, and so the surprise experienced by the predictor is $I = \log_2 20 = 4.3$ bits. The IBM predictor would have made the same prediction in this case.

The context $\langle c, q\rangle$ is upwritten by the predictor to the context $\langle C_2, C_4\rangle$, which gives the prediction for the next class as $\frac{1}{6}C_1 + \frac{1}{2}C_3 + \frac{1}{3}C_1$. The prediction for the next class is downwritten to the symbol distribution, giving prediction below.

$$G_{c,q} = [\tfrac{20}{63}\tfrac{40}{63}0000\tfrac{1}{21}]$$

The sequent symbol was b, and the predictor assigned this symbol the probability $p(b) = \frac{40}{63}$. The surprise experienced by the predictor is therefore $I = \log_2 \frac{63}{40} = 0.6$ bits. In this particular case, the IBM predictor would have fallen back to the unigram distribution, as the presence of the new q symbol means that neither the trigram nor the bigram distributions exist. This fallback would have resulted in 4.4 bits of surprise, a considerably poorer result than that given by the Q-Predictor.

As a final example, the context $\langle a, e\rangle$ consists of previously encountered symbols, but didn't occur in the training text. The Q-Predictor upwrites this context to $\langle C_1, C_2\rangle$, giving the predicted sequent class as C_4. The prediction for the sequent symbol is determined by downwriting the class prediction, and is given below.

$$G_{a,e} = [000\tfrac{25}{156}\tfrac{25}{52}\tfrac{25}{78}\tfrac{1}{26}]$$

The actual sequent symbol was f, which was assigned $p(f) = \frac{25}{78}$ by the predictor. The experienced surprise is hence $I = \log \frac{78}{25} = 1.6$ bits. For the same context, the IBM predictor would have fallen back to the bigram distribution, as the trigram hasn't been seen before. This would have resulted in the letter a being predicted as the most likely sequent symbol, causing a surprise of 4.7 bits.

Having predicted sequents for each context in the testing text, the predictor may be scored by calculating the average information provided to it by the text, and deriving from this the perplexity of the predictor. An additional piece of useful information is the number of times the predictor was infinitely surprised, I_∞.[16] The number of infinite surprises is an indication of how well the predictor was able to generalise its knowledge about the language.

Both the Q-Predictor and the IBM predictor were executed on the training corpus presented in this section, and the results of this are summarised in Table 4.1.

Algorithm	I_∞	$\overline{I}$	PP	Fallback
IBM	5	3.13	8.76	5
Q-Predictor	3	2.25	4.75	3

Table 4.1: Results of the Q-Predictor versus the IBM predictor

The Q-Predictor gave a 46% reduction in perplexity over the standard IBM predictor for this example. The fact that the Q-Predictor experienced less infinite surprises than the IBM predictor is indicative of the improvement of the language model. Similarly, the fact that fallback was required less often is a similar measure of generality.

Upper and lower bounds for the predictors score may be calculated by using the ideal grammar for the text. These calculations are based on the assumption that the training corpus does not contain new symbols or unseen contexts. These minimum and maximum scores are shown below.

[16]The predictor experiences an infinite surprise whenever a new symbol is encountered, or when the probability that it assigned to the symbol is zero. In both cases, the score of the predictor is calculated using Turing's approximation to keep the numbers finite.

$$\begin{aligned} \overline{I}_{min} &= \tfrac{1}{3}(\log_2 2 + \log_2 1 + \log_2 3) &&= 0.86 \\ \overline{I}_{max} &= \log_2 6 &&= 2.58 \end{aligned}$$

The IBM predictor experiences an average surprise which is greater than the theoretical maximum. This is because the testing corpus used actually deviated from the language produced by the grammar. The Q-Predictor scored much better, with an average surprise slightly less than the theoretical maximum.

These results are promising, as they show that the Q-Upwrite predictor may outperform the IBM predictor in some cases. In particular, the results indicate that the Q-Predictor copes better with sparse data, and is able to generalise its observations to cater for unseen trigrams.

Chapter 5

Testing and Results

In this Chapter, I present the results of testing the IBM predictor and the Q-Predictor on three different sources of text. In each case, it is important to select a testing text that contains trigrams that weren't in the training text, as the whole point of performing prediction is to guess which symbols are likely to follow a hitherto unseen segment of text.

In each of the following sections I introduce a source of text, dividing it into a training corpus and a testing corpus. Five different algorithms are used to make predictions about the testing corpus, and their performance is compared using several of the measures defined in Chapter 2.

The five prediction algorithms used are the IBM predictor, the interpolated IBM predictor, the Q-Predictor, the interpolated Q-Predictor and the clustered Q-Predictor.[1] These five different predictors allow a comparison of the IBM predictor and the Q-Predictor to be made, while enabling an analysis of the benefits provided by interpolation and clustering.

When interpolation is used, the weights are assumed to be $\alpha = \frac{16}{21}$, $\beta = \frac{4}{21}$ and $\gamma = \frac{1}{21}$. These are almost certainly not the optimal weights, but have been found to produce reasonable results for many texts. Provided that these weights remain

[1]The Q-Predictor doesn't require clusters to be found by default. The clustered Q-Predictor does use cluster information, but isn't interpolated.

constant, comparisons are possible between the IBM predictor and Q-Predictor. In the ideal case, optimal weights would be estimated for each prediction algorithm and training corpus, a process which tends to give quite different results depending on the predictor and corpus being used [21].

Clustering algorithms tend to be very slow on data such as that produced by the statistical analysis of a large Natural Language corpus. Indeed, the clustering stage proves to be the bottleneck of the Q-Predictor. For this reason, the Q-Predictor has been designed such that it is able to execute without clusters having been found. When it is necessary to find clusters, I have opted to use the k-means algorithm. This is purely because it is considerably more efficient than the dog-rabbit algorithm, and not because of the quality of the clusters produced. The dog-rabbit algorithm tends to produce smoother cluster centroids,[2] a property which is beneficial when predicting, as it increases the number of downwritten contexts that exist.

5.1 Artificial Grammars

Using a large Natural Language corpus is desirable in a project such as this. Unfortunately, though, a large corpus with a large alphabet tends to take a long time to process. During program development, therefore, I used artificial grammars to generate small amounts of low-dimensional text for testing of the developed programs.

A *Hidden Markov Model* was used to generate small artificial languages.[3] An example of such a model is given in Figure 5.1. Each state in the model has a single transition coming out of it, and a single transition feeding into it. A state contains a number of equally likely symbols, one of which is emitted at random whenever the state is entered.

Note that the model in the Figure has been designed to produce sentences of varying lengths, and to include the ambiguous word 'orange'. State 4 in the model contains

[2] Centroid vectors produced by the dog-rabbit algorithm tend to have a greater number of non-zero elements than their k-means counterparts. See the fruit clusters in Section 5.1.1 for an example of this.

[3] Two programs were developed to generate texts using Hidden Markov Models. Their implementation and usage is given in Sections B.8, C.3 and C.4.

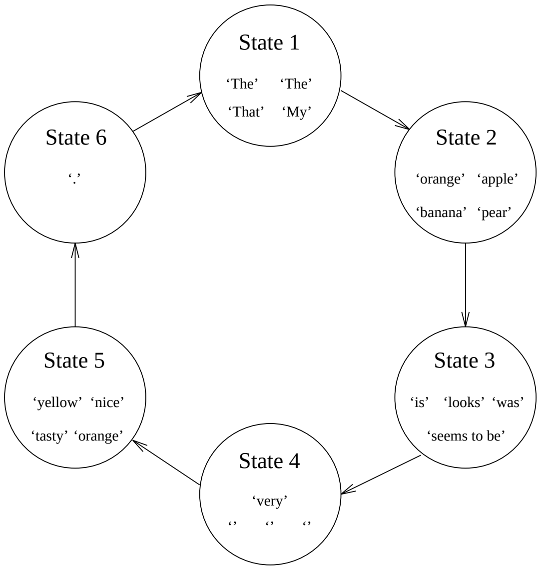

Figure 5.1: The Hidden Markov Model used to generate the corpora

four strings, all but one of them empty. This is done so that on average one in every four strings generated by the model will contain the extra word 'very'. State 1 in the model contains two versions of the word 'The'. This results in the word 'The' having twice the likelihood of the other two symbols in the state. An example of the output produced by the model is shown below.

> The pear looks nice. That orange looks tasty. My orange seems to be nice. The apple was yellow. That pear was orange.

The training corpus used in this Section was generated by the model, producing a corpus of 953 symbols. The testing corpus was generated by the same model, in such a way that sentences[4] in the testing corpus were absent in the training corpus. Additionally, all occurrences of sentences beginning with the context 'The orange' were removed from the training corpus. The testing corpus is shown below in its entirety.

> My banana is yellow. The apple seems to be nice. That orange looks orange. The orange looks nice. The orange is very nice.

[4]By sentences I mean sequences of symbols delimited by the '.' symbol.

In no way am I suggesting that the languages generated by Hidden Markov Models may serve as an approximation to Natural Language. I have used them merely to generate low-dimensional data for testing the programs.

It is interesting to note that the Q-Predictor infers a Hidden Markov Model from the training corpus. The classes inferred by the Q-Predictor are equivalent to the states in a Hidden Markov Model, as the classes contain a probability distribution of symbols. The IBM predictor, on the other hand, infers a standard Markov model from the text, and requires interpolation techniques to improve its performance.

5.1.1 Clustering

I tested the clustering algorithms on the data generated by a statistical analysis of the training corpus. Both of the algorithms performed 10,000 iterations, and used a cluster radius of 0.4. The dog_rabbit algorithm used an inhibition factor of 2.0, and incremented the dog's fatigue by 0.2 each time it neared a data vector. Three clusters were found by each algorithm.[5]

The k-means algorithm successfully terminated after 18 seconds, and generated an *xgobi* compatible file, allowing the data to be visualised. A plot of the clusters found by the k-means algorithm is shown in Figure 5.2. This plot is a projection of points in $\Re^{19}$ onto a plane. Note that the filled circles represent the cluster centroids themselves, the tiny crosses represent unclustered vectors, the empty circles represent vectors belonging to the cluster that corresponds to State 5 of the model, the large crosses represent vectors belonging to the cluster that corresponds to State 2 of the model, and the empty squares represent vectors belonging to the cluster that corresponds to State 3 of the model.

As an example, the cluster corresponding to State 2 in the model gives the symbol probabilities shown below. The training corpus is quite small, and doesn't contain every possible sentence in the language. For this reason, the symbol probabilities

[5]Only three cluster centroids were used by each algorithm although six clusters exist in total (one for each state in the model). It is unrealistic to expect all six to be found, as some states contain strings of more than one symbol. An S-Upwrite would probably be useful here, as it could amalgamate these strings of symbols into a single, higher-order symbol.

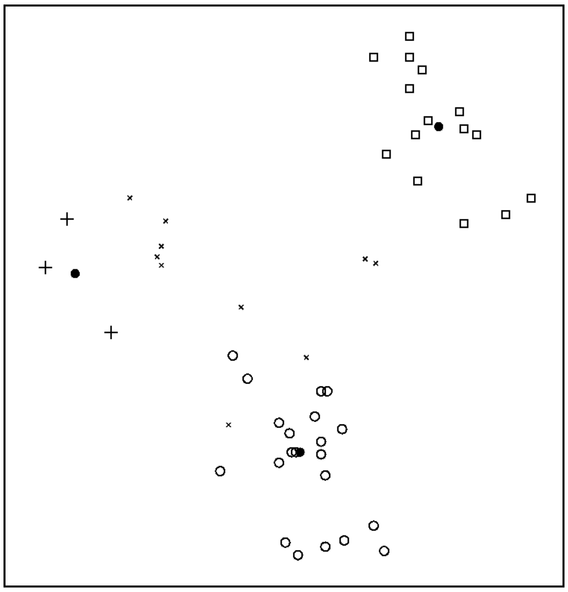

Figure 5.2: Clusters found by the K-means algorithm

don't match the state probabilities, but are expected to converge to them as the size of the corpus grows.

$$\begin{aligned} p(pear) &= 0.21 \\ p(orange) &= 0.35 \\ p(apple) &= 0.19 \\ p(banana) &= 0.25 \end{aligned}$$

The dog-rabbit algorithm took 38 seconds to execute, which is indicative of its computational complexity. A plot of the clusters found by the algorithm is shown in Figure 5.3. Note that although the same data was used, the two plots look different because they were produced by projecting points in $\Re^{19}$ onto two different planes. An example of the symbol probabilities given by the cluster corresponding to State 2 of the model is shown below. Note that symbols not in the state are assigned non-zero probabilities, a feature of the algorithm that helps to smooth the predictions made. These probabilities will tend to zero as the cluster converges towards that State that it corresponds to.

$$
\begin{aligned}
p(The) &= 0.05 \\
p(pear) &= 0.12 \\
p(looks) &= 0.03 \\
p(nice) &= 0.03 \\
p(.) &= 0.08 \\
p(orange) &= 0.20 \\
p(tasty) &= 0.04 \\
p(My) &= 0.02 \\
p(seems) &= 0.02 \\
p(to) &= 0.03 \\
p(be) &= 0.01 \\
p(apple) &= 0.09 \\
p(was) &= 0.02 \\
p(yellow) &= 0.04 \\
p(That) &= 0.02 \\
p(very) &= 0.04 \\
p(is) &= 0.03 \\
p(banana) &= 0.13
\end{aligned}
$$

Both of the clustering algorithms find clusters that correspond to states in the original Hidden Markov Model. This is encouraging, as it suggests that an approximation to the original model may be found by upwriting text using the clustered distributions.

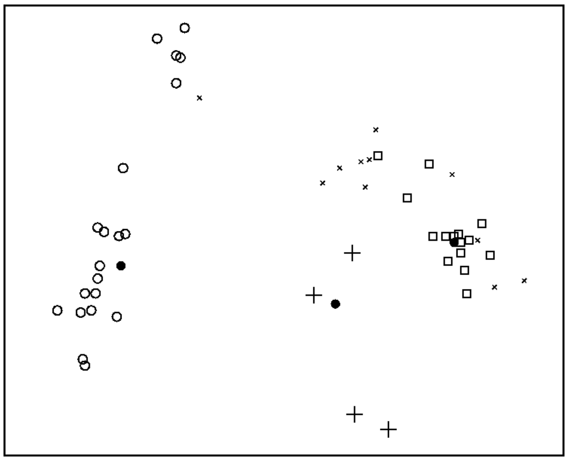

Figure 5.3: Clusters found by the Dog-Rabbit algorithm

As an example of this, the string 'That orange looks nice' was upwritten to the string 'That C3 C1 C2', where C1, C2 and C3 correspond to the three clusters found by the k-means algorithm. The word 'That' was not upwritten, because no cluster was found for the words in State 1 of the model.

5.1.2 Prediction

The five different prediction algorithms were trained and tested on the corpora presented in this Section, and the results of this are summarised in Table 5.1. The columns of the Table represent the number of times the predictor was infinitely surprised, the average surprise received by the predictor, the perplexity of the predictor, the rank of the predictor, the number of times the symbol was guessed correctly on the first try,[6] and the number of times that fallback was involved during prediction.[7]

Algorithm	I_∞	$\overline{I}$	PP	R	Correct	Fallback
IBM	3	2.72	6.60	4.74	11	2
Interpolated IBM	0	2.44	5.43	3.18	11	-
Q-Predictor	0	2.15	4.45	3.07	11	3
Interpolated Q-Predictor	0	2.24	4.73	3.18	11	-
Clustered Q-Predictor	0	2.04	4.12	3.00	11	3

Table 5.1: Results using the Artificial Corpora

From the table, it is seen that the IBM predictor performed the worst, with a received surprise of 2.72 bits per symbol. The clustered Q-Predictor gave the best results with its surprise of 2.04 bits per symbol. Because the testing corpus contained no new symbols, infinite surprise resulted only when the predictor assigned a probability of zero to the symbol that actually occurred. The IBM predictor was the only algorithm to do this.

Algorithm	Reduction in perplexity
Interpolated IBM	17.7%
Q-Predictor	32.6%
Interpolated Q-Predictor	28.3%
Clustered Q-Predictor	37.6%

Table 5.2: Reduction in Perplexity with the Artificial Corpora

Interpolation improves the performance of the IBM predictor, but degrades the performance of the Q-Predictor for this example. This is probably due to the fact that

[6] The training corpus contains 27 symbols, so around 41% of the predictions were right the first time.

[7] The number of fallbacks is the number of times the predictor fell back to the bigram distribution plus the number of times the predictor fell back to the unigram distribution. This number is not valid for interpolated predictors, which always use all three distributions.

interpolation provides a smoothing of the prediction, which the Q-Predictor does by default when it upwrites symbols to their corresponding classes. Interpolation for the Q-Predictor therefore introduces unnecessary additional information, such as the unigram distribution, which results in a reduction of the probabilities of symbols in the predicted class.

The perplexity of the Hidden Markov Model itself is 4, as each state in the model contains 4 equi-probable symbols. The clustered Q-Predictor has a perplexity which is remarkably close to the optimal. This result contrasts with the relatively high perplexities of the IBM predictors.

Table 5.2 gives the reduction of perplexity provided by each of the prediction algorithms over the standard IBM predictor. These results are quite encouraging, as they indicate the extent of the improvement that the Q-Predictor is capable of providing.

5.2 Juvenile Text

The primary problem with artificial grammars are that they *are* artificial. I am ultimately interested in making predictions about Natural Language text, and so it would be nice to use a Natural Language corpus for testing during program development. The catch is that a small corpus of only a few dozen words is required, to minimise the time taken for testing.

A suitable source of text is the books written to teach young children how to read. These books are based on a small vocabulary, and tend to employ a method of repetition, whereby each sentence in the text is a small modification of previous sentences. This allows the child to learn syntactic categories for words from their contextual use. Consider the training corpus given below, which consists of all but the last sentence of the book 'Who Will Be My Mother?'.[8]

> Mother sheep died. Lamb had no mother.
> 'Maa-maa, maa-maa,' cried Lamb. 'Who will be my mother?'

[8] I must thank Miss Leoni Weir, of Nedlands Primary School, for allowing me to raid their library and frighten the children.

Lamb went to the horse.
'Horse, Horse, will you be my mother?'
'I am a horse,' said the horse. 'I can't be your mother.'
Lamb went to the bull.
'Bull, Bull, will you be my mother?'
'I am a bull,' said the bull. 'I can't be your mother.'
Lamb went to the rabbit.
'Rabbit, Rabbit, will you be my mother?'
'I am a rabbit,' said the rabbit. 'I can't be your mother.'
Lamb went to the hen.
'Hen, Hen, will you be my mother?'
'I am a hen,' said the hen. 'I can't be your mother.'
Lamb went to the boy.
'Boy, Boy, will you be my mother?'
'I am a boy,' said the boy. 'I can't be your mother.'
'Maa-maa, maa-maa.' Lamb cried and cried.
'All right,' said the boy.

The testing corpus consists of the last sentence of the book, which is 'I will be your mother.'. All the words in this sentence occurred in the training corpus, though not in the same contexts.

The training corpus contains only 176 words, with an alphabet size of 37 words. This contrasts with the artificial grammar presented in the previous section, which used a training corpus of 953 words, with an alphabet size of 19 words. Consequently, the predictor is provided with less information about the language, while the language itself is more complex.

5.2.1 Clustering

The k-means clustering algorithm was used on the data extracted from the training corpus by the statistical analysis to find two clusters in the text. This result isn't too bad, considering the small amount of data available. The first cluster found contains punctuation, and the corresponding symbol probabilities are shown below.

$$\begin{aligned} p(.) &= 0.67 \\ p(,) &= 0.33 \end{aligned}$$

The second cluster found contains the animals themselves,[9] and its symbol probabilities are shown below.

$$\begin{array}{ll} p(horse) & = 0.19 \\ p(bull) & = 0.19 \\ p(rabbit) & = 0.19 \\ p(hen) & = 0.19 \\ p(boy) & = 0.24 \end{array}$$

This second cluster is the one that we would expect young children to be able to recognise. This is because the words in this cluster are the ones that make sentences in the text different, and hence may be considered to be the *interesting* words.

5.2.2 IBM Predictor

The IBM predictor was used to make predictions about the testing corpus. As an example of the quality of the predictor,[10] consider Table 5.3, which summarises the predictions made by the predictor about the testing corpus. Each column of this table represents a particular word in the sentence. The word at the top of the column is the word that actually occurred, while the list of words underneath it are the ones suggested by the predictor, sorted in order from most likely to least likely.[11]

I	will	be	your	mother	.
I	can't	you	my	mother	.
Lamb	am	be			
Hen					
Rabbit					
Bull					
Horse					
Who					
Maa					

Table 5.3: Predictions made by the IBM predictor

The IBM predictor tends to make fairly sparse predictions. Apart from the the first word of the sentence, when the unigram distribution is used, all other predictions

[9]The k-means algorithm makes the not unreasonable inclusion of the 'boy' in the animal class.
[10]I am referring to the standard non-interpolated IBM predictor here.
[11]Words that were assigned probabilities below a certain threshold are not included in the table.

consist of only one or two words. This sparseness results in erroneous predictions for the words 'will' and 'your'.

5.2.3 Q-Predictor

A summary of the predictions made by the Q-Predictor about the testing corpus is given in Table 5.4.

I	will	be	your	mother	.
I	,	be	be	mother	.
Lamb	went	a	your		?
Hen	can't		my		
Rabbit	am				
Bull	will				
Horse	-				
Who					
Maa					

Table 5.4: Predictions mmde by the Q-Predictor

The Q-Predictor tends to suggest more possible symbols than the IBM predictor does. All of its predictions are accurate, as the actual word is always a member of the suggested list of words. This means that the Q-Predictor doesn't experience infinite surprise at all.

All five of the algorithms were used to make predictions about the testing text, and the results of this are summarised in Table 5.5.

Algorithm	I_∞	$\bar{I}$	PP	R	Correct	Fallback
IBM	2	3.28	9.75	13.14	4	1
Interpolated IBM	0	2.99	7.96	7.28	4	-
Q-Predictor	0	2.10	4.29	2.43	4	3
Interpolated Q-Predictor	0	2.29	4.88	2.43	4	-
Clustered Q-Predictor	0	2.17	4.50	2.43	4	3

Table 5.5: Results using the Juvenile Corpora

The first thing that must be noted about these results is that the rank of the predictor is inaccurate when infinite surprise is involved, causing the relation in Equation 2.9

to be violated. As explained in Section 2.1.1, this is due to the technique being used to calculate the surprise and rank of the predictor in these situations.

Another interesting observation is that the clustered Q-Predictor performed slightly worse than the standard Q-Predictor. This is simply due to the fact that the only symbol upwritten in the testing corpus is the '.' symbol, and this upwrite reduces the probability assigned by the predictor to the '.' symbol, which is the one that actually occurred.

For this example, the standard Q-Predictor attained the best result with a surprise of 2.10 bits per symbol, while the IBM predictor scored worst with 3.28 bits per symbol. The reduction in perplexity provided by the prediction algorithms over the IBM predictor is given in Table 5.6.

Algorithm	Reduction in perplexity
Interpolated IBM	18.4%
Q-Predictor	56.0%
Interpolated Q-Predictor	49.9%
Clustered Q-Predictor	53.8%

Table 5.6: Reduction in perplexity with the Juvenile Corpora

The percentage improvements provided by the Q-Predictors are quite drastic, even when compared with the interpolated IBM predictor. The fact that such a large improvement can be attained when the training corpus is so small is quite promising.

5.3 Sherlock Holmes

So far, tests have been performed using very simple texts which contain a small number of words drawn from a limited alphabet. The size of the testing corpus has also been quite small. To perform more meaningful tests, a 'real' Natural Language text must be used. I decided to use 'The Adventures of Sherlock Holmes' by Sir Arthur Conan Doyle as a source for the corpora, due to the fact that it is readily available from the Oxford University FTP site.

The Sherlock Holmes text is 127,405 symbols long, with an alphabet of 8545 symbols. Although this is a marked improvement over the previous corpora I have used, it is considerably less text than the 1.5 million word corpus used to train the IBM predictor [34]. The Sherlock Holmes text consists of a number of short stories about the exploits of the famous detective. Each of these stories has a similar style of writing, and words relating to locations and characters tend to recur. Furthermore, a large proportion of the text involves dialogue between two or more characters, and this is structured in a consistent manner.

The testing corpus was formed by taking the first paragraph from the text, and removing all of the sentences within it that contained symbols not present in the remainder of the text. This gave the testing corpus which is shown below in its entirety.

> To Sherlock Holmes she is always the woman. I have seldom heard him mention her under any other name. It was not that he felt any emotion akin to love for Irene Adler. He was, I take it, the most perfect reasoning and observing machine that the world has seen, but as a lover he would have placed himself in a false position.

The training corpus therefore consisted of the entire Sherlock Holmes text, minus the first paragraph.

5.3.1 IBM Predictor

The standard IBM predictor performed quite poorly on the training corpus. For example, after the string ‘have seldom’, the IBM algorithm gave the sequent word as ‘seen’ with probability 1, whereas the word that actually occurred was ‘heard’. The string ‘have seldom’ appears only once in the training text, and this accounts for the poor performance of the predictor in this case.

Another case where the algorithm gives a prediction which turns out to be insufficient is after the string ‘that he’. The symbols predicted in this case are given below with their associated probabilities.[12]

[12] Symbols with probabilities less than 0.01 have been omitted from this list.

$$
\begin{aligned}
p(was) &= 0.20 \\
p(had) &= 0.20 \\
p(is) &= 0.12 \\
p(has) &= 0.07 \\
p(would) &= 0.05 \\
p(might) &= 0.04 \\
p(must) &= 0.03 \\
p(should) &= 0.03 \\
p(could) &= 0.03 \\
p(will) &= 0.02 \\
p(thought) &= 0.01
\end{aligned}
$$

All of the predicted symbols seem reasonable, and this is not surprising as they are based on a trigram that occurred frequently in the training text. Even so, the predictor assigned a probability of 0 to ‘felt’, the word that actually occurred.

5.3.2 Q-Predictor

The Q-Predictor has been shown to be able to generalise more about the structure of a language than the IBM predictor. Consider, for example, the prediction given for the context ‘have seldom’, which is shown below.

$$
\begin{aligned}
p(doubt) &= 0.05 \\
p(to) &= 0.03 \\
p(a) &= 0.03 \\
p(,) &= 0.03 \\
p(the) &= 0.02 \\
p(.) &= 0.02 \\
p(of) &= 0.02 \\
p(one) &= 0.02 \\
p(more) &= 0.02 \\
p(") &= 0.02 \\
p(it) &= 0.02 \\
p(you) &= 0.01 \\
p(little) &= 0.01 \\
p(in) &= 0.01 \\
p(be) &= 0.01 \\
p(been) &= 0.01 \\
p(my) &= 0.01 \\
p(seen) &= 0.01
\end{aligned}
$$

The word that actually occurred was 'heard', and this appeared in the 43^{rd} position of the sorted prediction vector. Note that even the most likely word was assigned quite a low probability by the predictor. Similarly, the Q-Predictor included the word 'felt' in its prediction for the sequent symbol to the context 'that he'. In this case, the word appeared as the 57^{th} element of the sorted prediction vector.

Table 5.7 summarises the performance of the five predictors on the Sherlock Holmes testing text. Unfortunately, results were unavailable for the clustered Q-Predictor due to the fact that the clustering algorithms took an inordinately long time to complete, and even then were found to have produced poor clustering of the data.

Algorithm	I_∞	$\overline{I}$	PP	Correct	Fallback
IBM	41	11.69	3307.32	9	20
Interpolated IBM	0	8.76	435.06	8	-
Q-Predictor	29	11.53	2966.25	3	3
Interpolated Q-Predictor	0	8.99	510.24	3	-
Clustered Q-Predictor	-	-	-	-	-

Table 5.7: Results using the Sherlock Holmes Corpora

Once again, the standard IBM predictor performs the worst with a surprise of 11.69 bits per symbol. The Q-Predictor does provide a minor improvement over the IBM predictor, but it is the interpolated IBM predictor which wins overall with 8.76 bits per symbol.

Note that the number of guesses that were right the first time is highest for the standard IBM predictor, which was the worst predictor overall. This measure isn't really indicative of the performance of the predictor, as sparse predictions may give a guess correctly the first time, while smoother predictions will tend to contain the correct symbol, though not necessarily in the top position.

One interesting note is that the non-interpolated predictors experience quite a high number of infinite surprises, given that there are only 69 symbols in the testing corpus. The IBM predictor is infinitely surprised 59% of the time, while the Q-Predictor is infinitely surprised 42% of the time. This high level of surprise is caused by the fact that most of the trigrams in the testing corpus have only been seen once before, and therefore have sparse distributions. As the Q-Predictor upwrites symbols based on trigram distributions, a sparse distribution will cause the upwrite process to fail.

The reduction in perplexity provided by these algorithms over the standard IBM predictor is summarised in Table 5.8.

Algorithm	Reduction in perplexity
Interpolated IBM	86.8%
Q-Predictor	10.3%
Interpolated Q-Predictor	84.6%
Clustered Q-Predictor	-

Table 5.8: Reduction in perplexity with the Sherlock Holmes Corpora

The fact that interpolation provides such an astonishing difference in the results of the prediction algorithms suggests that not enough data was used during the statistical analysis phase. This lack of data caused most trigrams in the text to have sparse distributions,[13] which severely degrades the performance of the non-interpolated predictors.

Unfortunately I do not have the computational resources to test the Q-Predictor on a source of text large enough to give better trigram distributions, such as the 1 billion word corpus suggested by IBM [34]. This is unfortunate, as results for corpora with small alphabets indicate that the level of improvement attainable is quite high, providing that the language model is of a sufficient quality.

5.4 Analysis of Results

In this Chapter I have performed a head-to-head comparison between the Q-Predictor and the traditional IBM predictor. Results indicate that the Q-Predictor is able to make better generalisations about the language than the IBM predictor. This fact is illustrated by the smoother prediction distributions produced by the Q-Predictor, and the quality of predictions for words that follow new contexts. The fact that the

[13] In many cases, the distributions contain only a single non-zero element.

Q-Predictor doesn't need to fallback as often as the IBM predictor does indicate that more information is being extracted from the text in order to make the prediction.

The Q-Predictor was shown to give vastly superior results than the IBM predictor for the two test cases that involved small corpora, and these results bode well for large Natural Language corpora. It was found that neither predictor performed well on the Sherlock Holmes corpus, and this was attributed to the fact that most trigrams in the corpus occurred only once.

It was found that interpolation techniques improved the performance of the IBM predictor and degraded the performance of the Q-Predictor. This observation wasn't true for the Sherlock Holmes corpus, due to the fact that the Q-Predictor was infinitely surprised many times, and so interpolation provided an improvement.

It was demonstrated that upwriting the text using word equivalence classes derived by clustering the n-gram data may improve the results of the Q-Predictor. However, the clustering algorithms were found to be too slow for realistic results to be obtained, and didn't produce the quality of clustering that was hoped. Unfortunately, therefore, results were unavailable for the clustered Q-Predictor on the Sherlock Holmes text.

Overall, the results suggest that the non-interpolated Q-Predictor consistently performs better than the non-interpolated IBM predictor. Even when using the Sherlock Holmes corpus, the Q-Predictor provides a respectable perplexity reduction of over 10%. For the smaller corpora, the non-interpolated Q-Predictor easily out-performed the IBM predictor – in fact, it even out-performed the interpolated IBM predictor.

Chapter 6

Applications

So far I have presented two applications of inferred grammars in this dissertation, namely entropic chunking and word prediction. Inferred grammars may be used for many other applications, and some of these were discussed in Section 2.3. In this Chapter, I present two applications that make use of the Q-Predictor. The first of these generates random texts, while the second proof-reads documents.

These applications are merely intended to show that the Grammatical Inference Engine and the Q-Predictor may be easily used to develop interesting programs, and are not supposed to represent serious uses of the algorithms.

6.1 Random Text Generation

In Section 3.1, I briefly mentioned Shannon's Markov model of English text, and gave an example of random text generated by Shannon using a 2^{nd}-order Markov model. I decided to write a program that could perform similar tasks, as such a program may allow one to gain an intuitive feel for how accurate a given language model is.[1]

[1] A realistic output from a text generation program merely shows that a suitable generative grammar for the training text has been inferred, and not that the grammar is a good representation of the language itself. A better test would be to calculate the ratio of grammatically correct generated sentences not in the training text to the total number of sentences generated.

The generation algorithm works by initialising the n-gram using the first $(n-1)$ symbols from the training text. These symbols are displayed to the output file, and a distribution of possible sequent symbols is given by the Q-Predictor. The algorithm selects the next symbol at random, such that it satisfies the predicted probability distribution. The selected symbol is displayed to the output file, and is used to make a new n-gram by concatenating it onto the original string of $(n-1)$ symbols. The process is repeated, using the new n-gram, for the desired number of iterations.

The output file created by the generation program contains the random text. The statistical properties of this text converge towards those of the language model as the length of the generated text increases. This means that if a grammar was inferred using the generated text as a training corpus, it would be equivalent to the original grammar in the limit.

An example of running the generation algorithm on the letter level using the Sherlock Holmes corpus[2] as the training text is shown below.

> A my vin th ser?"
>
> "I prom.
>
> "So said an nothe and sommoseemand wited ye. Wat pre shicell, itink Ha! I cartmarb hat younning nard ever nothre."

Note that this text does satisfy some English grammatical rules. For example, the next character after a full-stop is capitalised. The generated 'words' aren't valid English words, but are pronounceable. This is due to the fact that the last two letters constrain the type of the next letter.

Executing the generation algorithm using the same training corpus, but this time on the word level, creates a random text which is shown below.

> A Scandal in all, time care Watson Lee you dig would - answer bed management' help 'more beside 'credit I do. long remain you could? well I single watch been, dressed I just down up, this planked a four that looking is arm's moment by I in so date.

[2]All examples in this Section were trained on the entire Sherlock Holmes corpus, and were initialised using the symbols from the first sentence in the corpus – 'A Scandal in Bohemia'.

We would expect the generated text to approach the training text as the order of the language model increases.[3] A second version of the generation program was written in order to illustrate this phenomenon.[4] This program simply uses the $(n-1)^{th}$ order Markov model to generate random texts. Examples of the output it produces when operating on the letter level are shown below.

Order	Generated text
2	A musty," st of ishur proacte show posser har yon may on to hat trome clesighttham sat ham gractell, gres!
3	A Scart, at ash. 6d., Paran ther now tool it besting for sill day oned in my she coppers, tass the can Englar Watson?"
4	A Scarley, it for the to avoid of grief words in the wait and first?" "You depend worker. She troubles, I should however. It is not know will let us huge left his he was still, thought escape, half our dinner of copying stood have out morning my heavy more. The little definges."
5	A Scandal and fail to me to be alive virtues and of you would go above to Holmes late, but I saw her clothes once.

Note that the text generated becomes more like the original text as the order of the model increases, and that a 5^{th}-order model generates a text in which most of the 'words' are valid.

Texts generated by the program on the word level are shown on the next page. Note that even for a 2^{nd}-order model the generated text is quite readable, and the source of the text is obvious. The 4^{th}-order model generates a text that agrees with the original for the first few hundred words, while the 5^{th}-order model, (which is not shown in the table), generates the entire original corpus perfectly. This is because the training corpus contains only a single occurrence of each 6-gram, and so the algorithm that generates the text can only ever choose one symbol for each context.

[3] This is the result given by Shannon [51].

[4] This program was necessary because my implementation of the Q-Predictor is restricted to using a 2^{nd}-order language model.

Order	Generated text
2	A Scandal in Bohemia I To Sherlock Holmes was for the Continent. "Thank you, Miss Honoria Westphail, who probably know so much, as I felt the draught.'"
3	A Scandal in Bohemia I To Sherlock Holmes she is always the woman. I believe that a single lady can get on very nicely upon an income of about 60 pounds. Each daughter can claim an income of about 60 pounds. My God, my God, my God!
4	A Scandal in Bohemia I To Sherlock Holmes she is always the woman. I have seldom heard him mention her under any other name.

6.2 Document Proof Reading

The proof-reading algorithm works by reading $(n-1)$ letters from the text, and uses the resulting context to derive a probability distribution for the sequent letter. When the next letter is read by the algorithm, the surprise experienced by the predictor may be calculated. The symbol which occurred is deemed to be incorrect if the calculated surprise exceeds a certain threshold value.

The program was trained using the Sherlock Holmes corpus, and inferred a 2^{nd}-order Markov model for the text. When presented with the word 'Phantasmogorical', which unsurprisingly doesn't appear anywhere within the Sherlock Holmes text, the proof-reading algorithm makes the following suggestions, where the $\wedge$ represents whitespace.

Phantasmogorical
C 3 Replace a with: i
C 4 Replace n with: l
C 5 Replace t with: e d l a y o
C 6 Replace a with: $\wedge$ n d s - '
C 10 Replace g with: k o p n
C 11 Replace o with: e i
Phile$\wedge$smokerical

Note that for each letter in the word that is deemed to be incorrect, the program supplies a list of possible replacement letters, sorted by their likelihood. The most likely replacement letter is chosen by the program, and the 'corrected' output is given as 'Phile smokerical'.

To demonstrate via a rather contrived example that this application may occasionally give a corrected version of erroneous input text, the string 'To Sherlick Homes, she was always the womxx.' was presented to the program. The corrections made by the program are shown below.

To Sherlick Homes, she was always the womxx.
C 9 Replace i with: o y e a
C 15 Replace m with: l w r s u t p o n
C 16 Replace e with: m d l b
C 17 Replace s with: e a
C 17 Replace $\wedge$ with: s n r d
C 18 Replace , with: $\wedge$ ' b s t h
C 41 Replace x with: a e
C 42 Replace x with: n l r
To Sherlock Holmes$\wedge$she was always the woman.

In this example, the replacement symbols suggested by the program turn out to be the correct ones, and as a result the output string given by the program is identical to the original, correctly spelt string.

Chapter 7

Conclusion

In this dissertation I have shown that a language model that has been automatically inferred from a Natural Language corpus may be successfully applied to the word prediction problem. In addition to word prediction, the other applications of entropic chunking, random text generation and proof reading were presented.

The implemented prediction algorithm was shown to be able to generalise about the inferred language to a greater extent than the IBM predictor. The Q-Predictor didn't have to fall back as often as the IBM predictor when making predictions, and ranked its predictions in a more optimal order. For simple sample texts, the Q-Predictor easily out-performed the interpolated IBM predictor. On a larger sample of Natural Language text, the Q-Predictor was found to provide a 10% reduction in perplexity over the non-interpolated IBM predictor.

These results are particularly encouraging, and suggest that further work in this area may prove beneficial.

7.1 Limitations and Disadvantages

The main disadvantage of the Q-Predictor is that it is painfully slow. This has meant that I have been restricted to using small data sets, when evidence suggests that large data sets would produce better results.

When using smaller data sets, the distribution of frequency vectors in the data space is quite sparse. The clustering algorithms employed are unable to cope with this sparseness of data, and don't perform particularly well.

The Q-Predictor doesn't incorporate many of the features of the Upwrite Predictor. Of particular note is the fact that it performs only a single level of upwriting, and doesn't perform chunking of the symbols. This has meant that the prediction algorithm presented has not been fully exploited in the implementation.

7.2 Recommendations for Future Work

The Q-Predictor has been demonstrated to be superior to the non-interpolated IBM predictor when trained on various different corpora. But the Q-Predictor is a simplified version of the more generic Upwrite predictor, which infers a proper hierarchical model for the text. It is desirable, therefore, to design and implement a full version of the Upwrite predictor.

The prediction algorithms presented in this thesis rely on upwriting symbols in a text to their higher-level representations. Clustering the distribution vectors was proposed as a technique for deriving the necessary syntactic categories, but was found to be slow, and didn't produce the quality of clustering that was anticipated. Therefore, other techniques of finding word equivalence classes should be investigated.

In this dissertation I have been able to test the prediction algorithms on only relatively small corpora. Ideally, the algorithms should be tested using large training texts that give a good coverage of the language, and IBM have suggested that a corpus of at least 1 billion words is required to do this [34]. A future research possibility is to train the Upwrite predictor using such a corpus, and this would require a tremendous amount of computational power.

One final possibility for future work is to apply the developed algorithms to many of the applications presented in Section 2.3, and to test the extent to which inferred language models provide improvements over traditional techniques.

Bibliography

[1] Michael D. Alder. *Principles of pattern classification: Statistical, neural net and syntactic methods of getting robots to see and hear.* Lecture Notes, available for anonymous FTP from ciips.ee.uwa.edu.au in the directory pub/syntactic/book, February 1994.

[2] Mike Alder. *Stochastic grammatical inference.* Master's thesis, University of Western Australia, Nedlands, W.A. 6009, December 1988.

[3] D. Angluin and C.H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 6:237–369, 1983.

[4] A. Averbuch, L. Bahl, R. Bakis, P. Brown, A. Cole, G. Daggett, S. Das, K. Davies, S. Degennaro, P. de Souza, E. Epstein, D. Fraleigh, F. Jelinek, S. Katz, B. Lewis, R. Mercer, A. Nadas, D. Nahamoo, M. Picheny, G. Schichman, and P. Spinelli. *An IBM PC based large-vocabulary isolated-utterance speech recognizer.* Technical report, IBM T.J. Watson Research Center, January 1986.

[5] A. Averbuch, L. Bahl, R. Bakis, P. Brown, G. Daggett, S. Das, K. Davies, S. Degennaro, P. de Souza, E. Epstein, D. Fraleigh, F. Jelinek, B. Lewis, R. Mercer, J. Moorhead, A. Nadas, D. Nahamoo, M. Picheny, G. Schichman, P. Spinelli, D. Van Compernolle, and H. Wilkens. *Experiments with the Tangora 20,000 word speech recognizer.* Technical report, IBM T.J. Watson Research Center.

[6] L. Bahl, R. Bakis, J. Bellegarda, P. Brown, D. Burshtein, S. Das, P. de Souza, P. Gopalakrishnan, F. Jelinek, D. Kanevsky, R. Mercer, A. Nadas, D. Nahamoo, and M. Picheny. *Large vocabulary natural language continuous speech recognition.* Technical report, IBM T.J. Watson Research Center.

[7] L. Bahl, R. Bakis, P. de Souza, and R. Mercer. *Polling: A quick way to obtain a short list of candidate words in speech recognition.* Technical report, IBM T.J. Watson Research Center, September 1987.

[8] L. Bahl, P. Brown, P. de Souza, and R. Mercer. *A tree-based statistical language model for natural language speech recognition.* Technical report, IBM T.J. Watson Research Center, September 1987.

[9] L. Bahl, P. Brown, P. de Souza, R. Mercer, and M. Picheny. *A method for the construction of acoustic Markov models for words.* Technical report, IBM T.J. Watson Research Center, September 1987.

[10] L. Bahl, F. Jelinek, and R. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2), March 1983.

[11] P. Brown, J. Cocke, S. Della Pietra, V. Della Pietra, F. Jelinek, J. Lafferty, R. Mercer, and P. Roossin. *A statistical approach to machine translation.* Technical report, IBM T.J. Watson Research Center, October 1988.

[12] P. Brown, J. Cocke, S. Della Pietra, V. Della Pietra, F. Jelinek, R. Mercer, and P. Roossin. *A statistical approach to French/English translation.* Technical report, IBM T.J. Watson Research Center.

[13] P. Brown, F. Jelenik, and R. Mercer. *Basic methods of probabilistic context free grammars.* Technical report, IBM T.J. Watson Research Center.

[14] Noam Chomsky. *Aspects of the theory of syntax.* Special Technical Report of the Research Laboratory of Electronics 11, The Massachusetts Institute of Technology, 1965.

[15] Noam Chomsky. *Syntactic Structures.* Mouton, 1975.

[16] Daniel I. A. Cohen. *Introduction to Computer Theory.* John Wiley & Sons, Inc., 1991.

[17] Jordan R. Cohen. Application of an auditory model to speech recognition. *Journal of the Acoustical Society of America*, 85(6), June 1989.

[18] Richard Timon Daly. *Applications of the Mathematical Theory of Linguistics.* Mouton & Co., 1974.

[19] Pierre A. Devijver. Statistical pattern recognition. In K.S. Fu, editor, *Applications of Pattern Recognition*, chapter 2, pages 15–36. CRC Press, Inc., 1982.

[20] Jeffrey L. Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7, 1991.

[21] Ute Essen and Volker Steinbiss. Cooccurrence smoothing for stochastic language modeling. In *ICASSP 1990 International Conference on Acoustics, Speech and Signal Processing*, pages 161–164, 1990.

[22] King-Sun Fu and Taylor L. Booth. Grammatical inference: Introduction and survey - part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(3), May 1986.

[23] King-Sun Fu and Taylor L. Booth. Grammatical inference: Introduction and survey - part II. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(3), May 1986.

[24] K.S. Fu. Syntactic pattern recognition. In K.S. Fu, editor, *Applications of Pattern Recognition*, chapter 3, pages 37–63. CRC Press, Inc., 1982.

[25] E.M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[26] I.J. Good. A.M. Turing's statistical work in World War II. In J.L. Britton, editor, *Collected works of A.M. Turing : Pure mathematics*, pages 393–396. Elsevier Science Publishers B.V., 1992.

[27] Z.S. Harris. Distributional structure. *Word*, 10:140–162, 1954.

[28] John A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, 1975.

[29] Douglas R. Hofstadter and Daniel C. Dennett. *The Mind's I.* The Harvester Press Limited, 1981.

[30] David J. Hutches. *Data Structures and Algorithms for the Efficient Representation and Retrieval of Incremental Lexical Information.* PhD thesis, University of California, San Diego, 1993.

[31] Ajay N. Jain and Alex H. Waibel. Robust connectionist parsing of spoken language. In *ICASSP 1990 International Conference on Acoustics, Speech and Signal Processing*, pages 593–596, 1990.

[32] Frederick Jelinek. *Markov source modeling of text generation.* Technical report, IBM T.J. Watson Research Center.

[33] Frederick Jelinek. *Principles of lexical language modeling for speech recognition.* Technical report, IBM T.J. Watson Research Center.

[34] Frederick Jelinek. *Self-organized language modeling for speech recognition.* Technical report, IBM T.J. Watson Research Center.

[35] Frederick Jelinek. The development of an experimental discrete dictation recognizer. *Proceedings of the IEEE*, 73(11), November 1985.

[36] Frederick Jelinek. *Computation of the probability of initial substring generation by stochastic context free grammars.* Technical report, IBM T.J. Watson Research Center, April 1990.

[37] G.J.F Jones, H. Lloyd-Thomas, and J.H. Wright. Adaptive statistical and grammar models of language for application to speech recognition. In *IEE Colloquim on Grammatical Inference : Theory, Applications and Alternatives*, pages 25/1–8, April 1993.

[38] Julian Kupiec. Robust part-of-speech tagging using a hidden Markov model. *Computer Speech and Language*, 6(3), July 1992.

[39] M. Kutas. Reading senseless sentences. *Science*, 207:203–205, 1980.

[40] S.M. Lucas. New directions in grammatical inference. In *IEE Colloquim on Grammatical Inference : Theory, Applications and Alternatives*, pages 1/1–7, April 1993.

[41] Patricia McKenzie and Michael Alder. *Initialising the EM algorithm for use in gaussian mixture modelling.* Available on the World Wide Web on the URL http://ciips.ee.uwa.edu.au/pub/syntactic/papers.

[42] Laurent Miclet. Grammatical inference. In Horst Bunke and Alberto Sanfeliu, editors, *Syntactic and Structural Pattern Recognition*, chapter 9, pages 237–290. World Scientific Publishing Co. Pte. Ltd., 1990.

[43] G. Miller and N. Chomsky. Finitary models of language users. In R.D. Luce, R.R. Bush, and E. Galanter, editors, *Handbook of Mathematical Psychology II*, pages 421–464. Wiley, 1963.

[44] Marvin Minsky. *The Society of Mind.* William Heinemann Ltd., 1987.

[45] A. Radford. *Transformational Grammar, 2nd Ed.* Cambridge University Press, 1988.

[46] Allan Ramsay. Inference in language processing. In *IEE Colloquim on Grammatical Inference : Theory, Applications and Alternatives*, pages 5/1–3, April 1993.

[47] Martin Redington, Nick Chater, and Stephen Finch. *The potential contribution of distributional information to early syntactic category acquisition.*

[48] Hinrich Schütze. *Part-of-speech induction from scratch.*

[49] Hinrich Schütze and Yoram Singer. *Part-of-speech tagging using a variable memory Markov model.*

[50] C.E. Shannon. Prediction and entropy of printed English. *Bell Systems Technical Journal*, 30:50–64, 1951.

[51] Claude E. Shannon and Warren Weaver. *The Mathematical theory of Communication.* University of Illinois Press, 1949.

[52] John B. Smith. Computer criticism. In Jr. Walter A. Sedelow and Sally Yeates Sedelow, editors, *Computers in Language Research 2*, pages 25–59. Walter de Gruyter & Co., 1983.

[53] Raoul N. Smith. *Probabalistic Performance Models of Language.* Mouton & Co., 1973.

[54] V. Steinbiss, A. Noll, A. Paeseler, H. Ney, H. Bergmann, C. Dugast, H. H. Hamer, H. Piotrowski, H. Tomaschewski, and A. Zielinski. A 10,000 word continuous-speech recognition system. In *ICASSP 1990 International Conference on Acoustics, Speech and Signal Processing*, pages 57–60, 1990.

[55] Michael G. Thomason. Introduction and overview. In Horst Bunke and Alberto Sanfeliu, editors, *Syntactic and Structural Pattern Recognition*, chapter 1, pages 3–25. World Scientific Publishing Co. Pte. Ltd., 1990.

[56] Wen-Hsaing Tsai. Combining statistical and structural methods. In Horst Bunke and Alberto Sanfeliu, editors, *Syntactic and Structural Pattern Recognition*, chapter 12, pages 349–366. World Scientific Publishing Co. Pte. Ltd., 1990.

[57] J.P. Ueberla. Analysis of a simple bipos language model - attempt at a strategy to improve language models for speech recognition. In *IEE Colloquim on Grammatical Inference : Theory, Applications and Alternatives*, pages 10/1–8, April 1993.

[58] Julie A. van Dyke. *A syntactic predictor to enhance communication for disabled users.* Technical Report 92-03, University of Delaware, August 1991.

[59] Jr. Walter A. Sedelow and Sally Yeates Sedelow. Science and human language. In Jr. Walter A. Sedelow and Sally Yeates Sedelow, editors, *Computers in Language Research 2*, pages 1–23. Walter de Gruyter & Co., 1983.

[60] Satosi Watanabe. *Knowing and Guessing.* John Wiley & Sons, Inc., 1969.

[61] Warren Weaver. Translation. In William N. Locke and A. Donald Booth, editors, *Machine translation of languages : fourteen essays.* MIT Press, 1955.

[62] Yorick A. Wilks. Machine translation and the artificial intelligence paradigm of language processes. In Jr. Walter A. Sedelow and Sally Yeates Sedelow, editors, *Computers in Language Research 2*, pages 61–111. Walter de Gruyter & Co., 1983.

[63] Ave Wrigley. Parse tree n-grams for spoken language modeling. In *IEE Colloquim on Grammatical Inference : Theory, Applications and Alternatives*, pages 26/1–6, April 1993.

Appendix A

Implementing the Q-Predictor

Rather than implementing the Q-Predictor as a single monolithic program, I chose to write a suite of smaller programs, each of which performs a single step in the process. Each program in the suite must have knowledge about the language model, and so they share a global data pool which is stored on disk. For reasons of standardisation and portability, all the programs were implemented in *ANSI C* running under the *UNIX* operating system.

The programs are invoked in the traditional UNIX style. Optional parameters may be given on the command line or within a separate options file. Some of these options are used by all programs in the suite, while other options are specific to the current program. A full description of all program options and instructions for each program in the suite are given in Appendix C.

To allow recovery after a crash, and to prevent multiple programs from modifying the same data concurrently, I designed and implemented functions for creating and maintaining a database on disk. This database is used to store the results of the statistical analysis, while an accompanying log file records all major program operations.

In addition to the database files, programs may create their own data files. All files are created with the same filename, a three-character filename extension being used to distinguish between them.

Many of the programs require similar resources and functions, and so a large library of ten separate files was developed. Functions in this library are accessible by all the programs in the suite, and provide operations such as program initialisation, database access, binary searches and vector routines.

A.1 Statistical Analysis

The program *extract_ngram* was written to perform Statistical Analysis on the training corpus. To support symbols of any type[1] while minimising the storage requirements of each n-gram, it was decided to store symbols as strings of characters. Each unique symbol is entered into a a dictionary which assigns it with a 16 bit identifier, imposing an upper limit of 65,535 symbols in the alphabet.[2] A string of n symbols is thus represented by an array of n 16 bit numbers, which reduces storage requirements when the symbols are English words[3]

The dictionary is sorted alphabetically, which allows a binary search to be performed on the symbols. To retrieve a symbol when given its identifier, a second data structure, sorted by symbol identifiers, was required.

The n-gram data is stored in a file which is accessed via a hashed index which contains pointers to blocks of data on disk. A hash number is generated from the n-gram array itself, and this number is used to access the n-gram data on disk. Because different n-grams may generate the same hash number, a collision list is maintained for each entry in the index.

The n-gram data file consists of doubly-linked blocks of data, each 32 bytes in length.[4] The data for a particular n-gram is stored within several linked blocks. The head block contains the n-gram array itself, along with a count of the number of times it has occurred and the class it belongs to. This is followed by a list of symbol identifiers

[1] In this case, the symbols may be either letters or words.

[2] This is the highest number that can be stored in 16 bits.

[3] This is because the average length of an English word is around five bytes, while the length of a symbol identifier is two bytes.

[4] This value is defined in the program header file, and may be easily changed.

and their corresponding counts, which is a compact representation of the count vector, as only symbols with non-zero counts are stored in the file.

Statistical analysis involves creating the n-gram data. To do this, the program reads a symbol from the input file. If the symbol has been encountered before, its identifier is returned. Otherwise, it is assigned a unique identifier and added to the dictionary. The current n-gram is then created from the previous $(n-1)$ symbols and the new one.

The previous $(n-1)$ symbols are hashed to find the n-gram on disk. If the n-gram already exists on the disk, its count vector is updated using the symbol read from the input file. If it doesn't exist on disk, a new n-gram entry is created, and the count vector of this new n-gram is initialised using the symbol.

This process iterates until the end of the input file is reached. At this stage, the statistical analysis is complete and the program successfully exits. All data is written to the disk so that other programs may access it.

On completion, the program may optionally display the n-gram data in human-readable form, which allows one to inspect the gathered statistics. As an example, the *extract_ngram* program was executed using the Sherlock Holmes corpus as the training text. The data corresponding to the trigram $\langle ', I \rangle$ is displayed below. Each valid sequent to the trigram is listed, along with the number of times it occurred in the text.

1 : (')
8 : (answered)
1 : (said)
9 : (asked)
5 : (cried)
1 : (soon)
1 : (shouted)
1 : (read)
1 : (asks)
1 : (yelled)
1 : (screamed)
1 : (roared)
1 : (kissed)
1 : (panted)

The trigram $\langle ', I \rangle$ occurs when one character is relating an experience to Sherlock Holmes. This explains the relatively low count values in the example, and the presence of words such as 'soon' and 'kissed'.

A.2 Clustering

Implementation of the two clustering algorithms required functions for converting the n-gram statistics into frequency vectors, and for performing common vector operations. The two clustering programs implemented were *dog_rabbit* and *kmeans*, and they are essentially similar.

To allow the user to specify the number of clusters to be found, the maximum number of iterations, the stopping threshold, and other parameters specific to the clustering algorithm, library functions were modified to provide extra command line arguments and entries in the options file.

The number of clusters to be found is used to allocate the required number of centroid vectors, which are initialised to random locations. Each iteration of either algorithm involves updating the position of these centroids. Appropriate functions were created to perform the update operation.

Once the algorithms have performed the required number of iterations, they output the centroids of any clusters found to a data file, for other programs in the suite to use. Optionally, the centroid and data vectors may be written to files in a format suitable for the *xgobi* program, which is able to display the vectors on the computer screen.

The other elements of the cluster vector not displayed are equal to zero. In the original grammar, the members of the fruit class are equi-probable. A finite training text provides only an approximation to the language, and so the cluster found above is only an approximation to the real class.

A.3 Upwriting

The upwrite process is achieved by using the *assign* program to modify the data found by the statistical analysis. The assign program works by reading the centroid vectors from the appropriate data file, and comparing each centroid with the frequency vector of each n-gram. This process is used to determine the class of each n-gram, if any. If the frequency vector of an n-gram is within some user-defined radius of a particular centroid, then the n-gram becomes a member of that particular class. The membership information is written to the header block of the n-gram in the data file.

The result of this operation is to modify the n-gram data file such that each entry in the file has associated with it either the number of the cluster it belongs to, or the number zero indicating that the n-gram belongs to no cluster.

The *upwrite* program uses the cluster numbers assigned to the n-grams to upwrite the symbols in the input file to their class representations. This is done by reading strings of $(n-1)$ symbols from the input text, finding the corresponding n-gram from the data file, and finding the cluster that the n-gram distribution has been assigned to. If no cluster exists, the upwrite program falls back and repeats the process for the $(n-1)$-gram. If the program falls all the way back to the unigram, the sequent symbol is not upwritten, and is displayed unchanged to the output file. Otherwise, a symbol representing the assigned cluster is written to the output file.

A.4 Predicting

The Q-Predictor was implemented as the *predict* program. Ideally, the predictor would work for any order of language model. For practical reasons, however, I had to restrict the predict program to a 2^{rd} order model. This restriction is not too limiting considering that I am mostly concerned with comparing the Q-Predictor with the IBM trigram predictor.

The Q-Predictor works by reading a symbol from the testing corpus, and using this symbol to form an n-gram. This n-gram is upwritten using the cluster distribution of previous n-grams. Each symbol in the n-gram is upwritten in much the same way as in the upwrite program, with the following exceptions. Firstly, if a symbol is new and the upwrite process falls back to the unigram, then the symbol is upwritten to its prediction. This process prevents the predictor from falling back to the unigram distribution whenever a new symbol is encountered. The second exception is that if the symbol to be upwritten is assigned a zero probability in the distribution of the upwritten class, then it is added to the class.[5] By doing this, all the information contained in the original text is retained. If the predictor were adaptive, neither of these modifications would be necessary.

Having upwritten the n-gram, the program then loops over all possible non-zero combinations of downwritten n-grams, and uses each of these to predict the sequent symbol. In each case, the prediction is simply the probability distribution of the n-gram. The weighted sum of these prediction vectors is taken to formulate the final prediction.

In some cases, the prediction vector may be empty. When this occurs, the program performs a fallback, and predicts using the $(n-1)$-gram. This fallback process may be repeated until the unigram distribution is used for the prediction. The program was modified to allow the user to specify whether fallback or interpolation[6] should be used.

When the prediction process has been completed, relevant information is written to the log file. This information allows the user to gauge the performance of the predictor. For example, the output shown below was generated by the Q-Predictor using the Sherlock Holmes text as the training and testing corpus.

The program also outputs its predictions to another file. This file may be brief or verbose, depending on the requirements of the user. A brief version shows on each line the predicted symbol, the actual symbol, the resulting surprise and the number of guesses that would be required to correctly guess the symbol that occurred.

[5]The symbol is added to the upwritten distribution by assuming its probability is equivalent to that it would have if the predictor was adaptive and the given symbol occurred once.

[6]The IBM predictor may also use fallback or interpolation. Interpolation takes the weighted sum of the trigram, bigram and unigram predictions as the final prediction, with the weights supplied by the user.

Appendix B

Description of Source Code

In this Appendix, I give a brief description of all of the program code I have written during the course of my project. The program code itself is included on the floppy disk attached to the inside cover of this dissertation, and is also available on the CIIPS network in the directory `~hutch/code`.[1]

Each directory on the disk contains the source code for the appropriate program, along with a Makefile for compiling the program, and a document that contains a description of the program and its usage.

This Appendix is designed to reflect the directory structure contained on the floppy disk. That is, each Section in this Chapter corresponds to a directory on the disk, while the Subsections correspond to the files themselves. For a more detailed description of the source code than is given in this Chapter, refer to the comments within the files given on the disk.

B.1 Shannon

This directory contains ten separate source code files, each of which provides a particular set of functions that are in some way related. These files are intended to be used by any program that needs to access the language model database.

[1] Please contact me via e-mail if you have trouble gaining access to the source code. My e-mail address is `hutch@ciips.ee.uwa.edu.au`.

Also included in this directory is the standard header file *shannon.h*. This header file must be included by any program that uses functions from any of the source code files in the directory. It defines standard constants, macros and data-types that are used throughout the code.

Command.c

When using any of the language model programs, the user may specify program arguments and options in a file. This source code provides functions to parse the options file specified by the user, and sets the values of internal program variables accordingly. The function to parse the options file is usually called as part of the program initialisation phase.

Dict.c

A dictionary manager is provided by this code. The dictionary is used to store the symbols extracted from the training corpus. Functions are provided to read the dictionary file from the disk, write the dictionary to the disk, return a symbol from the dictionary when given its identifier, enter a new symbol into the dictionary, and perform searches on the dictionary. These functions define a standard interface to the dictionary, accessible by any program using the database.

Dog.c

The clustering algorithms produce centroid vectors which need to be stored to disk, as other programs in the suite may need to access these centroids. This code provides the necessary functions for reading and writing cluster centroids to and from the disk.

Error.c

All of the programs in the suite may need to generate messages to the user. This source code contains functions for generating error, information, debugging and warning messages. All of these messages are written to the specified file if it exists, or to the computer screen otherwise. Additionally, the information, debugging and warning messages may be suppressed by the user via the appropriate program options.

Exit.c

This code contains a function that hardens the database from memory to disk, creating necessary files for data recovery if the program terminates unsuccessfully. Other functions are provided for detecting and acting on signals, so that the program can recover from otherwise disastrous situations such as bus errors and segmentation faults.

Hash.c

A manager for the hashed index into the n-gram data is contained in this source code. Functions are provided for reading the hash index from the disk, writing the hash index to the disk, generating the hash address of the current n-gram, searching the hash index, retrieving a hash entry given a hash number, and adding a new entry to the hash index. These functions provide a generic interface to the data, and handle hash address collisions automatically and transparently.

Init.c

Upon program initialisation and termination, many house-keeping chores must be performed. Most if not all of these are handled by this code. An initialise function is provided which opens all files, initialises variables, allocates memory, reads program arguments from the user, performs a database recovery if required, and fires up the signal handler. The cleanup function does just the opposite – it closes files, hardens important data to disk, and de-allocates memory.

Ngram.c

An n-gram manager is a necessary part of the database, and is contained in this source code. Interfacing functions to the n-gram data are supplied, allowing searches for n-grams to be performed, new n-grams to be entered into the database, and general reading, writing and allocation of data. Additional functions allow the data to be displayed to an appropriate file in human-readable form.

Parse.c

Many of the programs in the suite need to read symbols from an input file. This code provides the necessary functionality. A function in the code reads the next symbol from the input file, where the type of symbol to be read has been previously specified by the user.

Vector.c

The clustering and upwriting algorithms perform vector operations, and this code contains functions that allow these operations to be performed in a standard and consistent manner. Functions are provided to convert n-gram data to its vector representation, allocate and free vectors, calculate the distance between two vectors, copy one vector to another, add one vector to another, normalise the specified vector, and display vectors in human-readable form.

B.2 Extract_Ngram

This directory contains the source code for the program that performs statistical analysis on the specified input texts.

Extract_Ngram.c

This source code defines a function that performs statistical analysis by reading a symbol from the input file, updating the current n-gram using the read symbol, then forming the new n-gram. This process is repeated until the end of the input file is reached.

An additional function is used to collect bigram and unigram statistics when the IBM mode is activated. This is done by replacing symbols in the n-gram with the wildcard symbol, and updating the statistics of the resulting n-gram.

B.3 Chunk

This directory contains the source code for the program that uses the database generated by the statistical analysis to chunk the input text into higher level symbols based on the entropy of the predictor.

Chunk.c

This code contains a function that performs an iteration over the input text. During the iteration, an n-gram is formed from the input text. This n-gram is read from the database, and is assumed to exist.[2] A function in the file calculates the entropy of the distribution vector belonging to the current n-gram. This entropy is compared with the threshold entropy specified by the user, and a separator symbol is written to the output file if it is found to exceed this threshold.

B.4 Cluster

This directory contains the source code for both the *dog_rabbit* and the *kmeans* clustering algorithms. Both of these programs use vector representations of the n-gram

[2]When chunking a text, the statistical analysis should be performed on the text being chunked in addition to other training texts. Therefore all n-grams in the text being chunked will exist in the database.

distributions to find the specified number of cluster centroids.

Dog_Rabbit.c

This code contains a function that performs the clustering algorithm, by iteratively updating the positions of the cluster centroids in accordance with the algorithm. Only a small amount of the entire code is specific to the algorithm, so that different clustering algorithms may be implemented easily.

Kmeans.c

This source code file is essentially identical to *dog_rabbit.c*, with the only notable difference being the part of the code that updates the positions of the centroids.

B.5 Upwrite

This directory contains the source code for two separate programs. The first uses the cluster centroids found by either of the clustering algorithms to update the database to include information about the clusters. The second may be used to upwrite the input text to its higher-level representation.

Assign.c

This code contains a single major function which loops over all of the n-gram data in the database, assigning each n-gram with the cluster that it is a member of, if such a cluster is found to exist.

Upwrite.c

This code contains a function which reads n-grams from the input file and uses them to upwrite the next symbol in the file. The upwritten text is written to the appropriate output file.

B.6 Predict

This directory contains the source code for the Q-Predictor and the IBM predictor.

Predict.c

The Q-Predictor algorithm is performed by a function in this code. The function is quite large and complicated, but essentially performs the algorithm as described in Chapter 4. Several support functions are provided, and perform such activites as guessing the most likely sequent symbol, calculating the received surprise, and returning either the distribution vector of the current n-gram.

Ibm.c

This code is similar to that in *predict.c*, with the difference that the function performs the IBM prediction algorithm instead.

B.7 Convert

This directory contains the source code for a program that is able to convert text given in SGML[3] format to standard ASCII format. The Sherlock Holmes corpus was initially in SGML format, and so this program was required to turn it into a form suitable for analysis.

Convert.c

Functions in this source code file return the next word from the input file in ASCII format. Two arrays are given in the file that define how SGML keywords and SGML

[3] A language used to specify the structure of a text. It is not too dissimilar to LaTeX, but is intended to be used as a standard platform-independent text file format rather than a typesetting language.

tokens map to their equivalent ASCII representations. These arrays do not provide exhaustive coverage of the SGML command set, but are sufficient for the Sherlock Holmes text.

B.8 Hmm

This directory contains the source code for two different programs, each of which is able to generate text using an internal Hidden Markov Model.

Hmm1.c

This code contains functions for generating a text using a Hidden Markov Model defined by the user. A function is provided which outputs the next character based on the current state of the model. The number of states is supplied by the user on program invocation.

Hmm2.c

This source file defines a function which generates the language given in Section 5.1. The code itself contains an array in which the symbols in each state of the model are stored. The function that generates the next symbol is similar to that in *hmm1.c*.

B.9 Gen

This directory contains programs that are able to generate random texts using the inferred language model.

Qgen.c

This code is very similar to *predict.c*, the major difference being that rather than predicting the next symbol as the most likely one, the prediction is made at random in such a way that the distribution of the sequent symbol is satisfied. When these 'predictions' are written to the output file, the random text with the same statistical properties as the training text results.

Gen.c

This code contains a function that finds the distribution of the next symbol, and outputs a symbol at random that satisfies this distribution. In essence, the function mimics a generative Markov model.

B.10 Check

This directory contains programs which extract n-gram statistics from a text on the letter level, and use the extracted data to proof-read the specified document.

A separate statistical analysis program was created for this purpose, as the application was intended to be used as part of a larger system. The programs were therefore written to be as optimal as possible, disregarding the database methodology employed previously.

This directory also contains the header file *spell.h*, which defines standard constants and data-types. This header file is included by all three of the source code files discussed below.

Spell.c

This source code contains commonly used functions that may be shared by both programs. Functions are provided for generating error messages, returning the next character from the input file, returning the next symbol from the input file, inserting n-gram data into the dictionary, finding a specified n-gram, reading and writing the n-gram data, and so on.

Ngram.c

This code contains a function that reads symbols from the input file, forms the appropriate n-gram, and updates the database with this information.

Check.c

This source code contains a function that gets the next n-gram from the input file, and calculates the surprise received when the sequent symbol is read. The sequent symbol is changed to another symbol if the received surprise exceeds a certain threshold. The symbol chosen to replace the sequent symbol in such a situation is chosen to be the most likely substitute.

Appendix C

User's Manual

In this Appendix, I give a brief description on how to use each of the programs that I have written for this project. The programs are divided into those that are part of the *shannon suite*, and those that are not. Programs within the shannon suite all access the language model database, while programs not in the suite provide other functionality.

Note that documentation is provided on the floppy disk for all of the programs, and may be identified by the `.doc` filename extension.

C.1 Invocation of Programs in the Shannon Suite

The programs in the shannon suite are *extract_ngram*, *chunk*, *dog_rabbit*, *kmeans*, *assign*, *upwrite*, *predict*, *ibm*, *qgen*, and *gen* All of the programs in the suite were designed to share the same database, and use the same library of functions. For this reason, each of the programs is invoked in the same way.

The programs output their data under the database name, using a filename extension for uniqueness. See the manual pages in the directories on the floppy disk for information about which data files are produced by which programs.

The programs in the shannon suite may be invoked in two different ways. The first method of invocation is in the standard UNIX style, whereby the user may specify various options on the command line. The second method of program invocation is to direct the program to read the options from a separate options file. This second method of invocation is provided to prevent the user from having to type many arguments each time he or she wishes to executes a program.

When an options file has been created, programs may be invoked using `program -O filename`, where `filename` is the name of the options file itself. If an option file isn't being used, the program arguments must be explicitly given on the command line. All of the program options available are summarised in Table C.1. The leftmost column of this table contains the option as it appears within the options file, while the centre column contains the corresponding command line switch.

Example options files are given on the floppy disk in the directories contained in the *test* directory. The option files end with the extension *.com* by convention. Included within these directories are the three corpora used for testing in Chapter 5.

C.2 Convert

The *convert* program filters a file in SGML format to give its standard ASCII representation. The program is invoked as shown below.

```
convert [-d] [-p] [-o outfile] [infile ...]
```

The `-d` switch specifies that additional information is to be displayed to the user during execution. The `-p` switch activates 'pretty-printing', which inserts line-breaks into the output text so that it fits neatly on an 80 column display. The `-o` switch optionally specifies the name of the file that is to be output by the program. Zero or more input files may be specified by giving their filenames.

If no input files are given on invocation, the program will attempt to get input from `stdin`. This allows the user to re-direct or pipe information to the program, or

Option	Switch	Purpose
ORDER	-n	The order of the language model
SYMBOL	-S	The type of symbols to extract
VERBOSE	-v	Include information messages during execution
SHOWGRAM	-s	Show the n-gram data in a human-readable format
DEBUG	-d	Include information useful when debugging
TIME	-t	Timestamp each message written to the log file
REMAIN	-r	Display the percentage of execution remaining
XGOBI	-x	Generate an *xgobi* compatible file
GNUPLOT	-g	Generate a *gnuplot* compatible file
NICE	-N	Minimise memory usage by using disk files as VM
NUMDOG	-D	The number of cluster centroids
ITERATE	-z	The maximum number of clustering iterations
RADIUS	-R	The cluster radius
PRUNE	-P	The minimum dog fatigue permitted
INHIBIT	-H	The dog inhibition factor
WORK	-w	The amount added to the dog's fatigue
MINMAX	-m	The minimum average movement permitted
IBM	-I	Use trigram, bigram and unigram statistics
INTERPOLATE	-p	Interpolate predictions rather than falling back
UNIGRAM	-u	Do not include the unigram distribution
ALPHA	-a	The trigram interpolation weight α
BETA	-b	The bigram interpolation weight β
GAMMA	-c	The unigram interpolation weight γ
THRESH	-T	The threshold entropy used when chunking the text
GEN	-G	The number of symbols to randomly generate
INFILE	-i	The input file(s)
OUTFILE	-o	The name of the database

Table C.1: Option file keywords and corresponding command line switches to type text in from the keyboard. Similarly, if the output file isn't specified, the program output will be sent to `stdout`, which by default will display the output on the terminal.

C.3 Hmm1

The *hmm1* program generates valid strings in a language defined by a Hidden Markov Model. The program is invoked as shown below.

```
hmm1 <states> <size> <len>
```

The number of states in the model is specified using the `<states>` parameter, while the number of symbols contained within each state is given by the `<size>` parameter. The number of symbols to be generated is specified by the `<len>` parameter. The generated symbols are written to `stdout`, where they may be re-directed to a file if necessary.

The symbols generated by the program are single letters. The program automatically assigns letters to states to satisfy the requirements of the user. As there is no practical limit imposed on the number of states or symbols that may be specified, the same symbol may have to be assigned to more than one state. This occurs when the number of states multiplied by the number of symbols per state exceeds 64, which is the number of distinct symbols recognised by the program.

C.4 Hmm2

The *hmm2* program generates the language defined by the model in Figure 5.1. It is invoked as shown below.

```
hmm2 <len>
```

The sole command line switch `len` is used to specify the number of symbols to generate. The generated symbols are written to `stdout`, and may be re-directed to a file if required.

C.5 Ngram

The *ngram* program infers an $(n-1)^{th}$-order Markov model for the language which contains the strings given in the input files. This model is referred to as a *dictionary*, and is used by the *check* program to proof-read texts in the same language. The *ngram* program is invoked as below.

```
ngram -n order -o dict [-i file]
```

The switch `-n` specifies the order of the language model, while the `-o` switch gives the name of the dictionary file to be generated. An input file is optionally specified via the `-i` switch, with the input being read from `stdin` when no input file was specified.

C.6 Check

The *check* program uses the dictionary database generated by the *ngram* program to proof-read a source of text. It is invoked as follows.

```
check -d dict -t threshold [-i file] [-o file] [-r] [-v]
```

The `-d` option specifies the dictionary file generated by the *ngram* program. The surprise threshold that is used to determine when an error has occurred is specified using the `-t` switch, while the `-i` and `-o` switches optionally give the input and output filenames. The input file is the one to be proof-read, and the output file will contain the 'corrected' version of the input file. The `-r` switch specifies that corrections to the text are to also update the current n-gram, which tends to give better results. Finally, the `-v` switch displays additional information about the corrections being made to the text.

If the input file isn't specified, input is taken from `stdin`. Likewise, input goes to `stdout` by default.