Probabilistic
Part of Speech Tagger

Aisling Fleming
B.A. (Mod.) CSLL
Final Year Project May 2002
Supervisor: Dr. Martin Emms
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_____________________________  May 3, 2002

Aisling Fleming
Abstract

For most natural language processing applications, a part of speech tagger is a fundamental component. A probabilistic part of speech tagger, implemented in C and based on the concepts of the Hidden Markov Model and the Viterbi Algorithm is presented here. The methodologies used in its implementation enable robust and accurate tagging. Accuracy exceeds 95%. Implementation strategies and optimisations resulting in high-speed, accurate operations are described.
I wish to express my thanks to the following people:

To my family, for their support throughout my college years.
To my flatmates, and friends for providing an enjoyable escape.
To Timothy (the tagger himself) for finally doing what he was told!
And especially to Martin, my supervisor, for his constant help, guidance
and encouragement.
Contents

Chapter 1.  Introduction ........................................................................................................1
  1.1 Aims.........................................................................................................................1
  1.2 Project Overview......................................................................................................3

Chapter 2.  The Art of Part of Speech Tagging ..................................................4
  2.1 What is Part of Speech Tagging? ........................................................................4
  2.2 General Algorithms and Methods ....................................................................5
    2.2.1 Supervised vs. Unsupervised ......................................................................5
    2.2.2 Stochastic vs. Rule Based ...........................................................................6
  2.3 My Approach .........................................................................................................7
    2.3.1 System Overview .........................................................................................7
    2.3.2 Corpus Based ...............................................................................................8
    2.3.3 Information Extracted from Corpus .........................................................9
      2.3.3.1 Lexical Probabilities ...........................................................................9
      2.3.3.2 Contextual Probabilities ...................................................................10
    2.3.4 The Underlying Model ..............................................................................11
    2.3.5 Viterbi Algorithm .......................................................................................13
    2.3.6 Disambiguation of the First Word ............................................................15

Chapter 3.  Linguistic Problems in Analysis ..................................................17
  3.1 The Token vs. The Word ....................................................................................17
  3.2 Tokenisation .........................................................................................................18
  3.3 Segmentation .........................................................................................................20
  3.4 Capitalisation .........................................................................................................21

Chapter 4.  Unknown Words .......................................................................................24
Contents

6.1 Observing the Tagging Process..............................57
6.2 Empirical Evaluation of the Tagger..........................58
   6.2.1 Automatic Evaluation....................................59
       6.2.1.1 Discussion of Automatic Evaluation..........61
   6.2.2 Manual Evaluation........................................62
       6.2.2.1 Discussion of Manual Evaluation.............62
       6.2.2.2 Problems with Portmanteau Tags.............64
6.3 Timing the Tagger.............................................65
6.4 Memory De-allocation........................................66
6.5 Observing the Databases.....................................67

Chapter 7  Conclusion....................................................68
7.1 Future Work.......................................................68
   7.1.1 Better disambiguation of capitalised words.........68
   7.1.2 Pre-processing of MWUs.................................69
   7.1.3 N-grams for further disambiguation................70
7.2 Summing Up.......................................................70

Bibliography..........................................................72

Appendix A  The Tagset.................................................74
Appendix B  Texts Used...............................................77
Appendix C  Code.......................................................93
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>A sentence containing ambiguous words</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Various approaches to automatic POS tagging</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Brief overview of the system</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>A simplified illustration of a HMM</td>
<td>13</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>The most common tags for Hapax Legomena</td>
<td>28</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Complete system architecture</td>
<td>39</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Menus to initiate the tagging process</td>
<td>40</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>The fate of a token!</td>
<td>44</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Tokenising a sentence</td>
<td>45</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>The function to apply the suffix probabilities</td>
<td>48</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>The basic Lattice</td>
<td>49</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Tagger output (showing lattice) for a sentence</td>
<td>51</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>Dividing the Portmanteau tags</td>
<td>56</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Observing the tagging process</td>
<td>58</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Results of tagging Text 1 in Appendix B</td>
<td>61</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Machine Specifications</td>
<td>65</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>A timed tagging of a large file</td>
<td>66</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Verification of memory de-allocation</td>
<td>66</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>Observing the Databases</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1.

Introduction

Consciousness...does not appear to itself chopped up in bits...A “river” or a “stream” are the metaphors by which it is most naturally described.

(William James)

Part of speech tagging is generally considered to be one of the basic and indispensable tasks in natural language processing. Taggers are used for parsing, for recognition in message extraction systems, for text-based information retrieval, for speech recognition, for generating intonation in speech production systems and as a component in many other applications.

Part of speech tagging involves assigning to a word it’s disambiguated part of speech in the sentential context in which the word is used. Many words are ambiguous in their part of speech. Even the word ‘tag’ can feature as a noun or a verb. However when the word appears in the context of other words, the ambiguity is often reduced: in “a tag is a part of speech label”, the word ‘tag’ can only be a noun.

Many sequences of tags are theoretically possible for a given sequence of words. Narrowing all possibilities down to the most probable sequence can improve dramatically both the speed and accuracy of later phases of processing.

1.1 Aims

The fundamental aim of this project was to build a probabilistic part of speech tagger that could quickly and efficiently handle large amounts of unprocessed text. Once that was done, a number of modifications and improvements could have been implemented. These included:
• using statistics from a number of different corpora (the BNC being the corpus that the tagger was built on) in order to drive disambiguation, with results converted to a fixed more parser-centric representation. Statistics from the Penn Treebank, and Suzanne Corpus could be collected and each made available to the disambiguation engine.

• Implementing an evaluation module to compare pre-tagged text with a re-tagged result, in order to evaluate design changes.

• Design a web-interface.

• The basic tagger returns just the best tagging. Adjustments could be made to allow the user to specify how many possible tags should be returned.

• The basic tagger considers only 1 preceding tag. Greater depth of context could be considered, and approaches to interpolating depth-0, depth-1 and depth-2.

• There is considerable scope for experimentation in processing unknowns; that is words that never occurred in the training corpus. This is important to do well, as a 20% rate of unknowns is not uncommon, even relative to large lexical resources.

• There is also scope of experimentation with tokenisation strategies, that is deciding where one word end, and another begins, and with sentence segmentation strategies.

• There is scope for experimentation with basic statistical model used (e.g. prob of word given tag vs prob of tag given word)

• There is a technique known as Baum-Welch parameter re-estimation, which automatically adjusts the probabilities given untagged text. A module implementing this could also be attempted

There is major scope for expansion and improvement on the tagger. Time limitations meant that in addition to implementing the tagger, only a small fraction of these modifications could be attempted. In the time allowed I aimed to deal with two of these, namely segmentation and tokenisation issues and unknown word handling.
1.2 Overview

In the course of this paper, I will outline previous methodologies and approaches to part of speech tagging (chap 2), as well as giving a theoretical (chap 2) and practical (chap 5) outline of the methods implemented by me. I will explain why these were chosen and how they were implemented. I will also outline various considerations and problems with different areas of part of speech tagging (chaps 3 & 4), and how I handled them in my project (chap 5). Finally, I will deal with performance issues of the tagger, how these were evaluated and how well the tagger performed (chap 6).
Chapter 2.
The Art of Part of Speech Tagging

You see, it’s like a Portmanteau – there are two meanings packed into one word.

(Through the Looking Glass (Lewis Carroll))

This chapter serves as an introduction to the concept of part of speech tagging and outlines the various methodologies adopted when implementing taggers. In the latter half of the chapter the approach adopted in this particular tagger is outlined. The algorithms and ideas behind the system are explained here. The actual implementation of the tagger, driven by these methodologies is given in chapter 5.

2.1 What is part of speech tagging?

Part of speech tagging strives to provide lexical descriptions for words in different situations and contexts. A crucial feature of lexical description is ambiguity. A single word form may relate to various mutually exclusive portions of linguistic information. The English word form ‘rust’ for instance is either a verb or a noun. In the verb sense, it relates to a finite or a non-finite form, and the finite reading can be interpreted as present, imperative or indicative. In highly inflectional languages, such as German, inflection introduces a need for greater disambiguation of verb forms.

The following diagram (Figure 2.1) shows how a simple sentence can contain a number of potentially ambiguous words. Under each word some of its possible parts of speech are shown, in order of decreasing likelihood. The correct tag is in bold font.
The can will rust.

Auxiliary modal verb modal verb noun
noun noun verb
verb verb

Figure 2.1 A sentence containing ambiguous words.

2.2 General Algorithms and Methods

Several different approaches have been used for building text taggers. An overview of the main approaches is given in Figure 2.2.

Figure 2.2 Various approaches to automatic POS tagging [VanG 95]
(This is a simplified description – in reality many tagging systems use aspects of some or all approaches for the purpose of tagging.)

2.2.1 Supervised vs. Unsupervised

Supervised taggers rely on pre-tagged corpora to help in the process of disambiguation. Various things that can be extracted from the corpus include; the lexicon, word/tag frequencies, tag sequence probabilities or a set of rules.
Unsupervised, on the other hand, use sophisticated computational methods to automatically induce word groupings (thus devising its own tagset) from raw, untagged texts, and based on this tagset, is able to calculate the probabilistic information needed by stochastic taggers, or to induce the rules needed by rule based systems (outlined in 2.2.2).

There are pros and cons to using either of these two methods: unsupervised taggers are extremely portable and can be applied to any corpus of raw text. The advantage of this is that reliable, pre-tagged text is not easily accessible for some languages or genres of writing. On the other hand, the word clustering (tagsets) devised by automatic taggers, tend to be very coarse, i.e. one loses the fine distinctions found in the carefully designed tag sets used in the supervised methods.

2.2.2 Stochastic vs. Rule Based.

The term 'stochastic tagging' can refer to any number of different approaches to the problem of part of speech tagging. Any model that somehow incorporates frequency or probability, i.e. statistics, may be properly labelled stochastic. If based on a corpus, then such a tagger would have to extrapolate various probabilities from observations in the corpus and use them in its disambiguation process.

In contrast, typical rule based approaches use contextual and morphological information to assign tags to unknown or ambiguous words. These rules are often known as context frame rules. As an example, a context frame rule might say something like: If an ambiguous/unknown word X is preceded by a determiner and followed by a noun, tag it as an adjective.

\[ \text{det} - X - n = X/\text{adj} \]
These rules can be either automatically induced by the tagger or encoded by the designer. Eric Brill designed the best-known rule-base part of speech tagger as it was the first one to be able to achieve an accuracy level comparable to that of stochastic taggers. [Brill 95a] Rule based taggers most commonly require supervised training; but, very recently there has been a great deal of interest in automatic induction of rules. One approach to automatic rule induction is to run an untagged text through a tagger and see how it performs. A human then goes through the output of this first phase and corrects any erroneously tagged words. The properly tagged text is then submitted to the tagger, which learns correction rules by comparing the two sets of data. Several iterations of this process are sometimes necessary.

Stochastic taggers have a number of advantages over rule based ones as they obviate the need for laborious manual rule construction and possibly capture useful information that may not have been noticed by the human engineer. However, these probabilistically driven ones have the disadvantage that linguistic information is only captured indirectly, in large tables of statistics. Also in their favour is the fact that automatic rule-based taggers have much smaller storage requirements and are more portable.

The fact remains though, that human language is random and difficult to capture in a finite number of prescribed rules. The probabilistic method of tagging means that such prescriptions are not necessary, and seems to be a more competent and robust method for tagging of unprocessed, random text.

2.3 My Approach

In the following sections I will aim to give a scientific reasoning for the methods and algorithms used in the basic tagging process of this project.

2.3.1 System Overview
The input to the tagging system is unrestricted English text and the unit over which the algorithm functions is the sentence. It tags a sentence purely through observations about different linguistic phenomena, which it observed in the corpus. A brief outline of this procedure can be seen in Figure 2.3

![Figure 2.3. Brief overview of the system](image)

**2.3.2 Corpus Based**

In a pure stochastic, corpus-based approach, as adopted here it is difficult to become aware of what cues to linguistic structures are actually being captured and where the failings are in these approximations to the true underlying phenomena. Information about words and their less usual contexts are hidden deeply in the thousands of contextual probabilities.

While this may seem like a shortcoming of the method, it is also the case that corpus-based methods are often able to succeed while ignoring the true complexities of language, due to the fact that complex linguistic phenomena can often be indirectly observed through simple epiphenomena.

In using the corpus as the only source of information, the tagger could infer probabilities of all possibilities. Every contextual possibility featured in the corpus is taken into account, not just the most frequent and this leads to a better overall picture of the language in question, as human language does not always conform to the rules set out for its construction. Even though less frequent constructions have lower
probabilities, they are still present to contribute to the tagging process, and so are not totally ruled out from the beginning.

### 2.3.3 Information extracted from the corpus

For execution of the tagging process, two different types of probabilities are extracted from the corpus: *lexical probabilities*, i.e. the probability of a particular tag conditional on the particular word, and *contextual probabilities*, which describe the probability of a particular tag depending on the surrounding tags. In this tagger the latter conditioning is based on bigram probabilities, i.e. only on the previous tag. These probabilities are explained in more depth below:

#### 2.3.3.1 Lexical Probabilities

Following [Tzou 99], the term genotype is used to capture the set of parts of speech a word can be tagged with. For example the word ‘catch’ has a genotype of [NN1 VVI VVB] (see Appendix 1 for a full explanation of these acronyms). A genotype is extracted for each word, based on observations in the corpus. This allows probabilities to be estimated on the genotypes rather than on the words. Genotypes play an important part in smoothing probabilities. They allow abstraction away from the word in question and further calculations to be based on the various tags of the genotype instead. By paying attention to tags only and thus ignoring the words themselves, this approach makes it easier to handle less frequent words and new words that have not been seen in the corpus. The formula used to calculate these probabilities is as follows:

$$P(t_i \mid w_j) = \frac{C(w_j \& t_i)}{C(w_j)}$$
This reads: the probability of a word $w_i$ having the tag $t_i$ is equal to the sum (denoted by $C$) of all times this occurred in the corpus, divided by the amount of times that particular word appeared in the corpus.

### 2.3.3.2 Contextual Probabilities

For the purpose of tagging, bigram probabilities were also considered. These are vital to the disambiguation process as they capture information about the probabilities of different tag sequences occurring. For example, in example (the can!), the word ‘can’ could have been a noun or a verb, depending on context. Lexical probabilities would have indicated that the word is a verb, as it occurs more often as a verb than a noun in the corpus. However if the bigrams ‘determiner followed by verb’ and ‘determiner followed by noun’ are considered, it is clear that the word ‘can’ is in fact a noun. The formula used to calculate these probabilities is given below:

$$P(t_2 \mid t_1) = \frac{C(t_1 \& t_2)}{C(t_1)}$$

This formula is used to calculate the probabilities of transitions for sequences of words. Put simply, it involves counting ($C$) the number of times that a certain tag $t_1$ occurred before another certain tag $t_2$, and then dividing this number by the amount of times that the former tag ($t_1$) occurred in the corpus.

For instance in determining the probability of a noun following a determiner. These categories are plugged into the formula as follows:

$$P(\text{noun} \mid \text{det}) = \frac{C(\text{det} \& \text{noun})}{C(\text{det})}$$

Again, this reads: the probability of a noun occurring given the occurrence of a determiner is equal to the number of times that a determiner and a noun occurred together, divided by the number of times that a determiner occurred.
The number of bigram probabilities for a corpus is equal to the number of tags in the tagset squared. Thus if we include the 61 tags in the tagset and the 30 portmanteau tags, a total of 8281 different bigrams can possibly occur. If this is compared to the amount of two-word combinations that could possibly occur in a 100 million word corpus as is used, the value of the contextual probability figures can be clearly seen, as they allow for syntactic generalisations to be observed. Put simply, the frequency of the sequence ‘determiner followed by noun’ is much higher than the frequency of the sequence ‘the followed by can’ in the corpus and since the tag sequence probabilities are based on much more occurrences that the word sequence one, they therefore have more value.

### 2.3.4 The Underlying Model

Following Charniak [Cha 96], the problem of part of speech tagging can be formally defined as:

$$\arg\max_{t_{1:n}} P(t_{1:n} \mid w_{1:n}) = \arg\max_{t_{1:n}} \frac{P(w_{1:n}, t_{1:n})}{P(w_{1:n})} = \arg\max_{t_{1:n}} P(w_{1:n}, t_{1:n})$$

To put this into English, the most probable set of tags $t_{1:n}$ must be found given a list of words $w_{1:n}$.

The tagger uses a first order Markov model for the purpose of part of speech tagging. This means that the contextual probabilities used will be bigrams, and so their scope will extend as far back as the last previous token. The states of the model represent tags and outputs represent words. Each state in the HMM corresponds to the part of speech of the word produced on the arc leading in to it. Transition probabilities depend on the states (as discussed in 2.2.2) and output probabilities depend on the most recent category. The sequence of tags returned is a Markov chain.
A Markov chain is a collection of random variables having the property that, given the present, the future is conditionally independent of the past. The aim of the calculations performed in the model is to compute the most probable sequence of tags $T = t_1, t_2, \ldots, t_n$ given a sequence of words $(W)$.

The two most important features of a Markov chain are that it has a limited horizon and is time invariant. The assumption that the chain has a limited horizon means that a word’s tag only depends on the previous tag, and the fact that this dependency does not change over time indicates that it is time invariant.

These properties cannot be taken as being fully conclusive for looking at what dependencies a word has, as for some long-distance dependencies, such as WH-extraction, the limited horizon property does not hold (as highlighted by Chomsky in 1958[Jur 00]). Despite this factor, first order Markov modelling is still seen to be very effective for the purpose of tagging.

The model implemented here is that of a Hidden Markov Model (HMM). Markov Models can only represent observable situations, and in part of speech tagging, although the words themselves can be seen, the tags themselves cannot. For this reason, a HMM is used, as it allows observed events (the words in the input sentence) and hidden events (the tags) to be built in to the model. Each hidden tag state produces a word in the sentence.

HMMs are both probabilistic and non-deterministic. Knowing an observation (word) sequence does not imply that a unique state (tag) sequence can be inferred. The most that can be inferred from such a model is the most likely state sequence, given an observation sequence.

The first step in the tagging process involves building such a model (henceforth to be referred to as the lattice). Each word can have a number of different tags, each of which
is seen to be a state in the lattice, the arcs are taken to be transitions from one word to the next. This can be seen in Figure 2.4

![Figure 2.4 A simplified illustration of a HMM (in which only some of the arcs are labelled for reasons of transparency)](image)

The next step in the basic probabilistic disambiguation process is to use the lexical and transitional probabilities to determine the optimal path through the search space ranging from one unambiguous tag to the next. Typically there are an exponential number of possible paths through such a lattices. This means that it is necessary to implement some kind of search algorithm to allow the various contextual and lexical calculations to be put to use in the disambiguation process and the most probable path to be calculated. The algorithm chosen to fulfil this purpose was the Viterbi algorithm.

### 2.3.5 Viterbi Algorithm

The most common algorithm for implementing an n-gram approach is known as the Viterbi Algorithm; an efficient dynamic search algorithm that avoids the polynomial expansion of a breadth first search by “trimming” the search tree at each level using the best N Maximum Likelihood Estimates (where N represents the number of tags of the
following word). It proceeds linearly in time proportional to the amount of words in the sentence.

The basic idea behind the algorithm is to compute the most likely tag sequence, starting with an unambiguous tag and working up to the answer, one state at a time. At each state only the most likely sequence, and its probability, up to and including that state.

As input the algorithm takes the lattice, which includes the sequence of observed words (\(W = (w_1, w_2, w_3, \ldots, w_n)\)) and returns the most probable tag sequence (\(T = (t_1, t_2, t_3, \ldots, t_n)\)) and the probability of that sequence. The algorithm is described in the following pseudocode:

Suppose the string of words \(W = (w_1, w_2, w_3, \ldots, w_n)\) has length \(n\).

for \(i = 1\) to \(n\)
  for all tags \(t_j\) of word \(w_i\)
    bestpath\((i,j)\) = 0
  for all tags \(t_k\) of previous word \(w_{i-1}\)
    current = bestpath\((i-1,k)\) * \(P(t_j | t_k)\) * \(P(w_i | t_j)\)
    if (current > bestpath\((i,j)\))
      then
        bestpath\((i,j)\) = current
        backpointer\((i,j)\) = \(k\)

Then backtrace from the most probable \(t_j\) for word \(w_n\), through the lattice, storing the path of states encountered, and finally return this path.

Within the product, for each tag \(t_i\), we compute the product \(P(t_k) * P(w_i | t_j) * P(t_j | t_k)\). The first of these terms is the probability of the previous tag currently being considered in the calculation. \(P(w_i | t_j)\), is the probability of the present tag, given the word and it tends to make the tagger prefer tags that are common for the word in question. The final
term, $P(t_j \mid t_k)$, is the probability of a tag $t_j$ being preceded by $t_k$. It tends to prefer tags that are likely to come after the tag for the previous word.

The tagger returns the tag sequence $t_{1,n}$ maximising $P(t_{1,n} \mid w_{1,n})$, where $w_{1,n}$ is a sequence of $n$ words and $t_{1,n}$ are the corresponding $n$ tags. To put this into words, for a sentence of length $n$ the tagger tries to find the tag sequence $t_{1,n}$ that has the highest probability given the words of the sentence, $w_{1,n}$.

Since statistical methods only take neighbouring tags into account within a limited window (in this case the word and its most previous word), sometimes the decision cannot cover all the linguistic contexts necessary for part of speech disambiguation. For instance the correct resolution of a preposition vs. subordinating conjunction ambiguity in a small window is often impossible because both morphological categories can have identical local contexts (both can be followed by a noun phrase). The idea of multiple paths through the lattice offers a solution to this problem. While in assigning a probability to a word, the bigram probability is used. The probability of the previous tag, is not just its probability as a tag, but rather the probability of the entire path up to and including that tag.

### 2.3.6 Disambiguation of the First Word

Difficulties were observed in tagging the first word of a sentence. In the tagging process as implemented here, a ‘.’ is inserted before each sentence, to act as a beginning of sentence marker. Using this additional tag, even if it stems from rudimentary processing of a punctuation mark slightly improves tagging results, especially for the first word in the sentence. The motivation behind inserting this extra token at the start of every sentence processed is that, without it, the first word only has the lexical probabilities (i.e. probability of tag given word (see 2.2.1)) to base it’s calculations on. Whereas by inserting the ‘.’ before this word, the transition probability of the tag
occurring after a punctuation symbol can be used for more context-driven disambiguation.

In the corpus itself the words and punctuation are taken as a sequence of words and not divided by sentence marker mark-up, and so the probability calculations included the probabilities of transitions from ‘end of sentence marker’ to ‘first word of next sentence’.

An example of the usefulness of this can be seen in the following sentence:

`.laughing is fun!`

The word `laughing` can be either a noun (as it is here) an adjective or a progressive verb. If only the lexical probabilities are considered then the word is tagged as an adjective, but if the two bigrams ‘punctuation followed by noun’ and ‘punctuation followed by verb’ are observed then `laughing` turns out to be favoured as being a noun.

Of course, the next bigram calculation of `is` following `laughing` would also aid in this disambiguation, but this stochastic method for tagging is driven by the availability of bigrams on both sides of a tag, and it is better for the first word in the sentence to also be able to avail of this.

So it was for these reasons that the placing of a full stop at the beginning of every sentence taken as input to the tagger came about.
Chapter 3.

Linguistic Problems in Analysis

*I could never make out what those damned dots meant.*

*(Winston Churchill)*

There are more linguistic issues to be considered in part of speech tagging, other than the tag for the word itself. The tagger must figure out what a word actually is, this problem then extends to determining what a sentence is, as the unit over which disambiguation operates is the sentence. It must also figure out what part different punctuation symbols play both in the word and in the sentence. The problem of discerning the motivation behind the capitalization of a word is also taken in to consideration, and what measures were taken to ensure that the first word in a sentence could undergo the same disambiguation process as the rest of the sentence. The procedures for text cleaning and normalisation as outlined in this chapter are of primary importance as they prepare the text for the tagging process.

3.1 The Token vs. The Word

The definition of the word has been, for a long time, a major problem for linguistic theory because, however the term word is defined, there are some items in a language which speakers of those languages call ‘words’ but which are not covered by the definition. In Paiute, a Native American language, wū-to-kuchum-punku-rūgani-yugwi-va-ntū-mū is a ‘word’ meaning ‘they who are going to sit and cut up with a knife a black bull (the ‘–‘ here separates the elements of the word in order to aid clarity, but do not actually feature in it). [Bau 83].

For the purpose of English, the task of determining a word from a string of characters is for the most part, fairly trivial. Our orthographic system typically marks word
boundaries in one of two ways; either using white space to separate the words or some punctuation symbol. Thus in the case of English, there is no problem with the most basic question in text analysis – figuring out what the various words of the input are. However, even this basic question becomes a nontrivial problem in a language with orthography like Chinese, where word boundaries are conventionally not marked and one must therefore segment text into words. This problem must be solved before other analysis of the text can proceed.

The purpose of the tagger is to competently deal with tagging of a full text ‘warts and all’, and thus must take more that the words themselves into consideration. For this reason, the unit of language to be tagged at each transition of the process will no longer be a ‘word’, but rather a ‘token’. The motivation behind this distinction will now be explained.

Punctuation cannot always be considered as a word boundary marker, nor does a punctuation symbol always constitute a token in itself. Sometimes a token can extend over multiple words, conjoined by punctuation symbols e.g. ice-cream. On the other hand sometimes a word must be split, and punctuation symbols tied to a particular part of the word e.g. ’Aisling’s’ is split into ‘Aisling’ and ‘s’.

### 3.2 Tokenisation

Tokenisation of text, involves splitting up a stream of characters into tokens (words, numbers, punctuation acronyms etc.). This task is not as simple as it might seem at first glance. Problems occur in the following areas:

- **Full stops**: Full stops are tokens on their own or they can occur as part of tokens, e.g. in the case of abbreviations, dates, ordinals, enumerations.
- **Hyphens**: can be token internal or can be used for punctuation. In the case of punctuation, surrounding words are usually delimited by spaces.
• **Acronyms**: these can occur in a variety of different ways: capital letters only e.g. IBM, letters separated with a full stop or even as a mixture of uppercase and lowercase letters with or without full stops e.g. in Ges.m.b.H. (Gesellschaft mit beschränkter Haftung, (private limited liability company)). Thus acronyms must be distinguished on the one hand from ordinary words in capital letter, and on the other hand from abbreviations. A clear distinction between abbreviations and acronyms is extremely difficult to achieve.

• **Enclitics**: these are word endings which should be regarded as being an actual word in themselves, although words containing them are conventionally represented as a single word. Examples of these include:

  - ‘d in I’d
  - ‘re in they’re
  - n’t in shan’t
  - n’t in wasn’t

Some words can even contain three separate words as in wouldn’t’ve

Observations can be made, and generalisations constructed as to what the nature of the word in a language is, but the fact remains, that language is neither regular nor defined. Constructions can be built at random and while they would be understood by a reader through the pragmatics of the situation, the tagger is forced to deal with them based solely on their syntax. They need not appear in the conventional fashion. The tagger must be built to try to copy with such irregularities. An example of such an unconventional, yet common construction is the following:

  “I’m so sleepy. z.z.z.z.z.z.”

The ‘z’s signify snores, but how should the tagger handle them? Is each ‘z’ a word in itself, should it be joined to it’s ‘.’ or is each ‘z.’ a sentence in itself? The tagger could read this stream of characters to be a single word, a number of words or in fact a number of sentences. This leads on to the next question of what constitutes a sentence.
3.3 Segmentation

Mikheev (in [Mik 99a]) explains how segmenting texts into sentences and sentence boundary disambiguation is an important aspect in developing many text processing applications: syntactic parsing, information extraction, machine translation, question extraction, text alignment, document summarisation to name but a few. For the most part, sentence splitting is a simple matter: a full stop, explanation mark or question mark usually signal a sentence boundary. However, numbers, abbreviations and acronyms must be taken into consideration as they can feature in the last position in a sentence, in which case their full stop acts both as part of the token and as the end of sentence marker. If we know that the word preceding a full stop is not one of these three things, then almost certainly the full stop indicates a sentence boundary. This is exacerbated by the fact that abbreviations and acronyms do not form a closed set i.e. one cannot list all possible abbreviations or acronyms. Say a finite list of all abbreviations could be enumerated, it can be seen that abbreviations can also coincide with regular words, i.e. ‘in’ can denote an abbreviation for ‘inches’ or ‘no’ can denote an abbreviation for ‘number’.

It was observed [Mik 99a] that the Wall Street Journal Corpus is rich with abbreviations and only 83% of sentences there end with a regular word followed by a full stop.

A clever way of dealing with such problems was presented in the same paper. It presented a Document-Centered Approach to Proper Name and Abbreviation Handling. This approach basically derives the disambiguation of an individual token is derived from the entire document rather than from it’s immediate local context. This process involves looking at all potential abbreviations (each word of length four or less, followed by a full stop). If one of these words is used elsewhere in the document then it can be concluded that this is in fact not an abbreviation. To decide whether a potential abbreviation is in fact one, the document is searched to see if it does in fact appear in a context followed by a full stop, and then by either a lowercase word, a number or a
comma. This method relies on the assumption that there is a consistency of writing and vocabulary within the same document. When neither unigrams nor bigrams can help to decide whether the token is an abbreviation or not, the system decides in favour of the more frequent category for the potential abbreviation from the current document. If all other methods fail then the word is simply taken to be a non-abbreviation.

But in general, such constructions are problematic. In the British National Corpus (the corpus used in this particular tagger), there is no mark-up used to indicate that a word is an abbreviation, and so observations on different types of abbreviations and acronyms are almost impossible. Ideally, they should be tagged as they are for the full form of the word and the full stop should feature as a part of the word, but this is extremely difficult to achieve.

3.4 Capitalisation

The unconventional and erratic nature of human language comes to the fore in the classification of rules for capitalisation and punctuation. It seems that while there are well laid rules as to what is and is not allowed – these rules are there to be broken, or at least to be stretched to the limit! For every list of rules there is a far longer list of exceptions.

Capitalised words present a case for ambiguity – they can stand for proper names as in “Brown replied, …”, or they can be just capitalised common words as in “Brown snakes are …”. Another quirk in capitalisation rules is the fact that adjectives derived from location such as Irish, American, etc. are always capitalised, whether they are acting as an adjective (An Irish girl) or as a proper noun (She is Irish.). It can be observed of language, that there are many circumstances where there are no hard and fast rules as to whether a word should be capitalised or not in, and even if such a rule exists, it is often times not adhered to. An example of such a literary mistake can be seen in Appendix B
Text 2 in which the initial letters of the name ‘Queen Mother’ are capitalised in one paragraph and then lower case in the following one.

In general the disambiguation of capitalised words doesn’t seem to be too difficult: if a word is capitalised in an unambiguous position, e.g. not after a full stop, or other punctuation which might require the following word to be capitalised (such as quotes or brackets), it is a proper name or part of a multi-word proper name. However, when a capitalised word is used in a position where a word would usually be capitalised, it is extremely difficult to decide whether it is acting as a proper name or not.

Since words are always capitalised at the beginning of the sentence it is not sufficient to look up only the capitalised form of the word in the lexicon. This is necessary to prevent the word *Old* in the sentence *Old cars drive slowly* from being tagged as a proper noun as in *Old Blue Eyes*.

A word that begins with a capital letter can be capitalised for any number of reasons. It can be acting as a proper noun or can simply be a capitalised version of a common word. The former denotes proper names – names of organisations, locations, people, etc., but there are also other positions in a text where capitalisation is expected. Such positions include the first word in a sentence, words in all-capitalised titles, a word following a colon or an open quote, the first capitalised word in a list entry.

Mikheev (in [Mik 99a]) tells of another approach to the disambiguation of capitalised words. He shows that a Document-Centered Approach similar to that, which can be used to handle abbreviations (see section 3.3) can be successfully employed. Also the sequence strategy outlined in the same paper deals with multi-word capitalised sequences e.g. names of organisation etc. His approach returned an error rate of less than 0.3%. In general this approach works better on longer documents rather than shorter as more word bigrams can be collected to aid in disambiguation.
In general it can be seen that language has its nuances and that it is sometimes difficult for the tagger to determine the reason for a word’s capitalisation. If a word appears at the start of the sentence and the word does not feature in the corpus, then the first letter could be put into lower case and the corpus rechecked, as most words that are not proper nouns are much more likely to feature in the corpus in this format, than at the start of sentences.
Chapter 4.

Unknown Words

_Apart from the known and the unknown, what else is there?_

_(The Homecoming (Harold Pinter))_

One problem remains outstanding in regards to the tagging approach discussed so far: How should unknown words be dealt with? The problem is outlined in the following chapter, as are various methods to deal with it. In the latter part of the chapter an outline of the method used in this particular tagger, to deal with unknown words, is given.

4.1 The Problem of Unknown Words

How can one calculate the probability that a given word occurs with a given tag if that word is unknown to the tagger? Granted, the rules in rule-based taggers are equipped to address this issue, as are automatic taggers, but what happens in the stochastic models?

Words unknown to the lexicon present a problem to the corpus driven part of speech tagger, as it relies on morpho-syntactic information for its analysis. The part of speech tagging depends heavily on a lexicon of words to supply the required $P(w_i \mid t_i)$. If a word is not available in the dictionary then this probability needs to be provided in some other way.

Sparseness of training data is a common problem in statistic-based approached to part of speech tagging. In case of estimation of lexical associations on the basis of word forms, a fairly large amount of training data is required, because all inflected forms of a word will be considered as different words.
The fact of the matter is, that no corpus could ever possibly contain all words likely to be found in real input. Psycholinguists have argued that humans not only know about morphological structure, but also that they are able to use this knowledge to both actively parse and produce morphologically complex words. Thus, if people are able to freely generate new words while making use of the morphology of their language, then it is fairly hopeless to expect any corpus to contain all the words one is likely to encounter.

Take for example the verb ‘to floop’ (a favourite verb of mine which means to lazily meander through the day doing nothing (originally coined by Phoebe from the Friends TV series)). This can be used in a variety of different ways, and could be used to from a construction in any of the open classes or words (discussed in 4.2.1)

e.g. Verb I flooped about the house all day.
Noun He is such a floop.
Adjective Her floopy attitude will be her downfall.
Adverb I did the work, if somewhat floopily.

For an unknown word it would appear that no lexical information could be extracted from the corpus. It can be observed that the loss of lexical information greatly lowers the accuracy of the tagger. Thus some means of improving the tagger so as to handle such words must be found. Various methods of doing this are discussed below.

### 4.2 Approaches and their Motivations

The simplest approaches to the tagging of unknown words involve either assigning all possible tags of a tagset to an unknown word, or assigning the most probable tag, which a proper singular noun for capitalised words and common singular noun to it otherwise. The appealing feature of these approaches is their simplicity. Not surprisingly, their
performance is quite poor: if a word is assigned all possible tags, the search space for
the disambiguation of a single tag increases and makes it fragile; if every unknown
word is classified as a noun, there will be no difficulties for the disambiguation, but the
accuracy will not be reliable enough.

Although to assign capitalised unknown words as proper nouns seems a good heuristic
“…some capitalised words are proper nouns and some are not” was argued in [Chu 88]
where a more elaborate heuristic was proposed.

Dermates and Kokkinakis(cited in [Mik 97]) proposed a simple probabilistic approach
to unknown word guessing: the probability that an unknown word has a particular pos-
tag is estimated from the probability distribution of words which occur only once in a
corpus (so called hapax legomena (see 4.3.1 for further explanation). This method
alone when used in isolation for disambiguation has a reported performance of 66% for
English (but 82% for German). [Mik 97]

More advanced tag-guessing methods use word features like prefixes and suffixes in
order to determine all possible tags. Such methods can achieve better performance,
reaching up to 85% of tagging accuracy on unknown for English

Fortunately, a small set of words accounts for the vast majority of unknown word
occurrences. The reason for this can be seen in the fact that the parts of speech for
words used in language fall into two broad classes: closed class, and open class. These
are outlined in the following section.

4.2.1 Open and Closed Class Words

Parts of speech can be divided into these two main super categories: closed class and
open class. [Jur 00] Closed classes have a fixed membership and no new or unknown
words are likely to appear in these classes. The words tend to be very short and occur
very frequently in language. For example, there is a fixed set of personal pronouns and no new ones are likely to be coined. Other word forms contained in this set include prepositions, conjunctions, and auxiliary verbs. By contrast, the set of words that compose the open class is constantly expanding and changing, and include the word forms, nouns, verbs, adjectives and adverbs. The word ‘floop’ as mentioned earlier is an example of such a word.

Other methods for unknown word guessing have been studied including the rule-based method and the decision tree-based method (see [Mik 97] for further elaboration)

Words not found in the corpus are generally both open class and regularly inflected (thus having a similar morphological structure as known words). This makes the algorithm employed (outlined below) a very effective means to guess the most probable set of tags for an unknown word.

### 4.3 Approach Taken to Handling Unknowns

In the assigning of a genotype (set of tags) to a word that did not occur in the corpus the tagger employs lexical and orthographic observations and contextual rules.

In tagging a known word, a genotype of most probable tags was extracted from the corpus and inserted as a column into the lattice. For an unknown word, a similar column must be built.

The statistical approach adopted here (from [Man 99]) is used to discern what tags and probabilities should be inserted into the column of the lattice. It takes the following factors into consideration: hapax legomena, capitalisation and suffixes. These are outlined below, followed by an explanation of the equation:
4.3.1 Hapax Legomena

The set of Hapax Legomena contains words which appeared only once in the entire corpus. The phrase *Hapax Legomena* is Greek for ‘read only once’. The idea of using these to aid in the process of the disambiguation of unknown words stems from the idea that the probability distribution of words that occurred only once in a corpus should be similar to the distribution of unknown words. All closed class and usual words, like prepositions or determiners, will have a far higher frequent distribution in the corpus, than say nouns or verbs (as these can be invented ad hoc). Thus the probability $P(w_i|t_i)$ for an unknown word is determined by the average of the distribution of that tag over all singleton words in the corpus.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Meaning</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN1</td>
<td>Noun</td>
<td>21.4</td>
</tr>
<tr>
<td>NP0</td>
<td>Proper Noun</td>
<td>16.5</td>
</tr>
<tr>
<td>AJ0</td>
<td>Adjective</td>
<td>13.6</td>
</tr>
<tr>
<td>NN2</td>
<td>Plural Noun</td>
<td>12.6</td>
</tr>
</tbody>
</table>

Figure 4.1 The most common tags for Hapax Legomena.
(as feature in the BNC)

4.3.2 Capitalisation

Capitalisation plays an important role in the disambiguation of unknown words. In general, if the word is capitalised, its genotype will be very different to that of a capitalised word. Capitalised words have a much greater probability of being proper nouns, and are extremely unlikely to be finite verbs (as mentioned in 5.3.2, only finite verbs cannot appear at the head of a sentence, and so if encountered are much more
likely to be non-capitalised). This factor plays an important role in determining which of the Hapax Legomena tags are likely for the word in question. It narrows down the possibilities, as the set of Hapax Legomena stretches over all words. Using the word’s capitalisation as a factor in determining its probability allows for some word-specific colouring on the previous tags and probabilities found for that word.

### 4.3.3 Suffix Information

A scientific reasoning for including suffix information is that within the same language, generally unknown words obey general morphological regularities. The process of forming a new word by attaching an affix to a word or a morpheme is called derivational word formation. This process is very productive and if one form of a word is known, then corresponding forms can be coined. An example if this is the relatively newly coined verb ‘to download’. The process of producing new words from this proceeds as follows:

- Verb -> Noun e.g. The downloads are small.
- Verb -> Adjective e.g. The downloadable files are small.

The suffix is a strong predictor for word classes, e.g. words in the BNC ending in *able* are adjectives (e.g. wearable, fashionable) and nouns (cable, variable). Observations about the morphology of the token allow the tagger to colour the tag probabilities given by the hapax legomena and capitalisation calculations so that they are specific to the word in question.

---

1 The term suffix as used here means “final sequence of characters in a word” which is not necessarily a linguistically meaningful suffix.
4.3.3.1 Probable Tags given Suffix

To aid in the processing of unknowns by the tagger functions that perform morphological analysis were implemented so as to provide more information about the given word. A hash table of all suffixes that appeared in the corpus and their corresponding tags and probabilities, was used for this purpose. The method of calculation of these suffixes and their probabilities is outlined below.

In this tagging method, the largest suffix considered was of length 6 and the smallest was of length 1. The probability distribution for a particular suffix is generated from all words in the corpus that share the same suffix of a particular length. The algorithm for the extraction of information for the suffix is similar to that of 2.3.3.1 used to extract information for the word.

Basically, it extracts the largest suffix from a word (maximum length 6) and performs the following calculation based on that suffix:

\[
P(t_i & s_i) = \frac{C(t_i & s_i)}{C(s_i)}
\]

To put this into words: a count (C) is performed of all the occurrences of a tag \(t_i\) with a word that has the suffix \(s_i\). This is then divided by the sum of all occurrences of words having that suffix.

It then recursively performs this on the next smallest suffix, until the smallest suffix (of length 1) has been calculated. This calculation is performed for every unique suffix that features in the corpus. If the word is smaller than length 6 then the entire word is taken for calculation and undergoes the same recursive process.

The hash table of pre-calculated tags and probabilities for suffixes resulting from these calculations meant that the tagger’s job of extracting useful morphological information
for an unknown word was very easy. The tags and probabilities for the suffix of the unknown word (again of maximum length 6) could be easily discerned.

This involved first checking if the largest suffix of the unknown word featured in the hash table. If so then the tag information stored in the database was extracted and used for disambiguation. On the other hand, if this suffix did not feature, then the next smallest suffix is taken. This process continues until a suffix is found which features in the hash table, or until the suffix has been reduced to length 1.

Take for example the word ‘stochastic’. Its suffix of length 6 is extracted – ‘hastic’. Then the suffix ‘hastic’ is searched for in the suffix hash table. If this is not found, then ‘astic’ is tried and so on until a suffix features in the table. If it doesn’t feature, then the word is taken to be a proper noun.

4.3.4 Algorithm Used

These probabilities are all incorporated into the following equation [Jur 00]:

\[ P(w_i | t_i) = P(\text{hapax legomena} | t_i) \cdot P(\text{capitalised word} | t_i) \cdot P(w_i \text{ suffix} | t_i) \]

This reads: the probability of \( t_i \) being the tag for the unknown word \( w_i \) is the probability of that tag being assigned to a hapax legomena word, times the probability of that tag occurring for a capitalised word, times the probability of that tag occurring with a suffix of that word.

The beauty of this approach is that it brings many factors to bear on the final probability distribution of the tags of the genotype selected for an unknown word. It uses general information about the usual tags for infrequent words as well as morphological
information based the word itself as well as other cues, namely the occurrence of a capital letter at the start of the word.

This calculation assumes that the features are mutually independent. Even though this is not strictly true for all words e.g. a capitalised word is more likely to occur as a hapax legomena than a non-capitalised one, so the features ‘hapax legomena’ and ‘capitalised word’ in the above equation are not really independent.

For this reason, an adaptation was made on the original algorithm. For the purpose of this project the corpus observations on hapax legomena and capitalisation were done simultaneously. Two finite lists were compiled, one with all the tags that applied to capitalised hapax legomena tokens, and one with all the tags that applied to non-capitalised hapax legomena. The probabilities then could be observed over each of the entire lists. These two lists were then stored so that when an unknown word was encountered, a simple observation on the state of capitalisation of its initial letter was the only work needed to apply a genotype to that word. This meant that during runtime, the only calculation necessary was the colouring of the particular tags of the genotype with the probabilities of the suffix (as observed from the corpus and stored in the database). The tags in the hapax legomena list that did not occur with the suffix were not discounted, but the ones that occurred with the suffix now had much higher probabilities, and so were more likely to be chosen.

The tags and probabilities obtained through this process, when taken with the bigram probabilities extracted when the tags for the unknown word feature as a column in the lattice, allow for an extremely accurate calculation of the most probably tag for a given word.

It was concluded in evaluation experiments that the morphologically driven guessing rules do improve accuracy (chap 6). This sort of analysis would probably be of even greater benefit to taggers of a language like German, in which one can join any amount of words together to form one new word. In such a word the part of speech would be
easily determinable from observing the suffix, as the past of speech for the newly compounded word, depends only on the final word particle as can be seen from the example:

Fussballweltmeisterschaftsqualifikationsspiel

This can be translated as ‘football-world-competition-qualification-game’, or ‘world cup qualifier’ as it is more commonly called. Since, in German, ad hoc construction of new words is much more likely than in English and their part of speech only depends on the final syllable, the methodology of this approach could be of use when implementing taggers in other languages also.
Chapter 5.

Implementation Outline

*You shall know a word by the company it keeps!*

*(Firth, 1957)*

This chapter outlines the particulars of my tagger; the resources used in its implementation and how the implementation was carried out.

5.1 Resources Used

Various resources were required in order to build the probabilistic part of speech tagger. These included a corpus and its tagset, a database to hold various probabilities from the corpus, and a language in which to implement it. These are all elaborated on below.

5.1.1 The British National Corpus

The British National Corpus (BNC) is [BNC 95]:

- A sample corpus: composed of text samples no more than 45,000 words in length.
- A synchronic corpus: includes imaginative and informative texts.
- A general corpus: not restricted to any particular subject, field, register or genre.
- A monolingual British English corpus: texts are the product of native speakers of British English.
- A mixed corpus: it contains both spoken and written language.
The 100-million-words of the BNC were tagged automatically, using the CLAWS4 automatic tagger. With a corpus of this magnitude, post-editing and correction of tagging errors produced by the automatic tagger was not undertaken, and so the errors (about 1.7% of all words) remain in the corpus. In addition, the corpus contains ambiguous taggings, shown in the form of ambiguity tags (also called "portmanteau tags"). An example of such a tag is VVD-VVN, which indicates that the automatic tagger was unable to decide, with sufficient likelihood of success, which was the correct category, VVD (past tense verb) or VVN (past participle), and so left two possibilities for the user to disambiguate. Approximately 4.7% of the tags in the Basic Tagset tagging of the BNC are portmanteau tags.

The BNC is encoded using a complex mark-up scheme known as the Corpus Document Interchange Format (CDIF). This scheme follows the reference concrete syntax of SGML, in which all elements are delimited using tags. These delimiters don’t just provide mark-up for the words and their relevant tag, but for the entire structure of material contained in the corpus e.g. paragraph markers, markers surrounding a quote and even, in the case of the spoken transcriptions, paralinguistic phenomena like non-verbal but vocalised sounds (e.g. cough), and speaker’s accent to name but a few.

### 5.1.1.1 The C5 tagset

The BNC is tagged using the C5 tagset (referred to as the “BNC Basic Tagset” in some literature). Each C5 tag represents a grammatical class of words, and consists of a partially mnemonic sequence of three characters: e.g. NN1 for "singular common noun". Different tagsets vary in size, granularity, complexity and level of annotation. These are outlined below with particular reference to the C5 tagset.

#### Size

The C5 tagset consists of 61 tags, 4 of which are used to tag various punctuation symbols. In addition to this there are 30 portmanteau tags. These are not defined in
the tagset, but rather, built from existing tags. These are generally two tags in length and appear as the two original tags, joined by a hyphen.

**Granularity**

Tagsets vary in granularity of the information conveyed. The BNC has different levels of granularity for various parts of speech. For example, it distinguishes between different types of adjectives (comparative, superlative etc.), but does not provide similar distinctions for adverbs.

**Complexity**

Tagsets differ with respect to the complexity of tags, i.e. a tag could convey simplex or complex information. In the majority of tagsets different classes of information are mingled together into a single tag. Part of speech tags can either represent syntactic category or both syntactic category and inflection. The former is sufficient for languages with little inflection such as English, but the latter is preferable for highly inflected languages like German. A typical example of this is that verbs are usually annotated according to their syntactic category (main, auxiliary, modal) and their inflectional properties, such as finite, infinite, past participle.

Apart from syntactic category and inflection, the tagset also contains tags for punctuation, symbols, interjections etc.

### 5.1.2 The Berkeley DB

Berkeley DB is an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications. It provides a simple function-call API for data access and management.[Sle 01])

Smart storage techniques, allow for much faster manipulation than flat files. Searches and sort algorithms on flat files can be very time-consuming.
The Berkeley DB system offers several different indexing algorithms (known also as access methods) including Btree, Hash, Queue and Renco. The most popular and widely used of these is Hash, and this is the one used in the implementation of the tagger.

Storage and retrieval for the Berkeley DB access methods are based on key/data pairs. Both key and data items are represented by the DBT data structure that stores a pointer to memory and a length. DBT is an acronym for database thang and was names this simply because no one could think of a reasonable name that wasn’t already being used somewhere else! [Sle 01]. When accessing the database, a key is passed in and the data structure associated with that key is returned.

5.1.2.1 Why is it necessary to use the Berkeley DB?

Using the Berkeley DB to store the bare information needed for disambiguation by the tagger meant that the complex mark-up scheme of the BNC (a scheme known as the Corpus Document Interchange Format (CDIF)) could be abstracted away from

The corpus-based approach adopted in designing the tagger allowed a wealth of linguistic knowledge to be accessed and weighted in order to return the most likely tag sequence. Brill [Brill 92] showed that, although corpus-based approaches have been successful in many different areas of natural language processing, it is often the case that these methods make it difficult to capture the linguistic information they are modelling directly. This can make it difficult to analyse, understand and improve the ability of these approaches to model underlying linguistic behaviour.

The Berkeley DB hash tables accessed by the tagger offered a wealth of easily accessible, pre-calculated statistics (see 5.2.4 for a complete explanation). This ensured that the various knowledge required from the corpus for the purpose of tagging was available and the Berkeley DB C API allowed these to be extracted with minimum hassle.
5.1.3 The C Programming Language

Kerninghan and Ritchie [Ker 88] say that C is a ‘pleasant, expressive and versatile language for a wide variety of programs’. It is relatively low-level functional language. The version used in the tagger, ANSI C was formally defined in 1988 and is now supported my most compilers. The memory management functions supplied with C do not provide garbage collection. This means that memory allocation and deallocation have to be done manually when necessary.

A powerful, yet initially confusing feature of the C programming language is the pointer. Pointers enable programs to simulate call-by-reference and to create and manipulate dynamic data structures. Such techniques are used extensively throughout the program.
5.2 System Outline

Following is an outline of the tagger implementation following the methodology discussed in section 2.4. A visual outline can be seen in Figure 5.1. Running through the description at each stage is an example of how the tagger handles the sentence ‘Englebert’s 4th dog died.’ Hopefully this will make the process at each stage more transparent.

The tagger is menu driven so as to allow ease of access to all functions. Input can be taken in from the terminal in the form of a single sentence or from a file. File handling is outlined below. An example of the initial menu is given in Figure 5.2.

Would you like to use input from standard input or from a file?
Standard input & output - type 's'
Input from & output to files - type 'f'
Input from particular file - 'p'
Please enter s, f or p: s
Please enter your sentence:
Englebert's 4th dog died.

What kind of analysis would you like?:
Simple - 1
Showing lattice - 2
Full explanation - 3
Please enter 1, 2 or 3:

Figure 5.2 Menus to initiate the tagging process.
5.2.1 File Handling

The tagger was designed so that it could be used inside other application that could make use of its quick and accurate language processing functions. For this reason it was important that it should be able to accept large amounts of unprocessed text, process it and return it in a clear and uniform fashion. Some taggers demand that text be in a certain format in a file so as to allow ease of sentence extraction, but this tagger is robustly built so as to accept input in its raw, unprocessed form and then handles this text accordingly.

If the user specifies that input be extracted from a file, then the tagger waits for the particular filename to be entered. If this file can be found, it is opened and sequential sentence extraction begins. Each sentence is extracted, tagged and the resulting string of tagged text is printed to file (the output file having the same name as the input file but followed by the extension .out).

If the filename cannot be found then the user is prompted for another filename to be entered.

5.2.2 Segmentation

Segmentation, as outlined in chapter 3, is necessary when input is to be extracted from a file. The segmentation process extracts characters from a file sequentially, and stores them in an array, until an end of sentence marker is reached. The end of sentence markers recognised by the tagger are the full stop, the exclamation mark and the question mark.

The segmentation function of the tagger is built so as to recognise when a full stop occurs in a digit, and in this case, extraction does not stop, but rather continues on until a true end of sentence marker (or the end of file marker) is encountered.
Extraction does not stop if a new line marker or extensive white space is encountered so that the unit employed in the tagging process is a true sentence, bounded by a marker.

5.2.3 Tokenisation

The tokeniser accepts a pointer to an array of characters as input. This array is either the output from the segmenter, if input is from a file, or simply the string entered by the user if input is taken directly from the terminal. This string is then passed through the tokeniser, whose duty is to convert this sequence of characters into a sequence of tokens. There are many possibilities to be considered, as outlined in 3.2, as white space is not the sole token separator.

First of all, the string is passed through a pretokeniser, which divides the input string into a string of tokens, each separated by a single white space. It places a ‘.’ at the beginning of the sequence (for reasons outlined in section 3.4), and arranges the rest of the input into a neatly ordered string. It takes the following possibilities into consideration:

- If single or multiple white space markers (space, new line, tab) are encountered together, replace them with a single white space character.
- If a number is encountered, keep all its digits together, including any decimal points or commas that may occur inside it. But if the number ends with a full stop then this must be kept apart as it is more likely to be a end of sentence marker. An example of this is ‘I was born in 1981.’
- If an ordinal number is encountered e.g. 3rd then the number, and the following suffix must be kept together, so as to form a single token.
- If a punctuation symbol is encountered, then it must be separated from the word it lies beside. Generally no white space features between a comma
and the word that precedes it, or a quotation mark and the word that follows it. These punctuation symbols must be separated from the word by a single white space. But this is not the case for all punctuation. There are some exceptions which must be dealt with:

- when an apostrophe occurs in the middle of a word then this word must be split into two tokens, with the apostrophe being tied to the start of the second token. e.g. *I’d* is split into *I* ‘*d*

- if a hyphen is encountered and it features inside a word, thus having no white space at either side of it, then the word should be kept together (e.g. *ice-cream*) but if white space does occur around the hyphen, then the hyphen must be bound by white space at both sides.

Once the string has been converted from a string of characters to a string of tokens (see section 2.2) for an explanation of the difference between the two concepts, the tokens can be extracted and each stored as an element in a dynamic array of strings. Figure 5.3 shows the different procedures that could take place when building a column for a token.
Figure 5.3. The fate of a token!
The dynamic nature of the array stems from the fact that the size of the sentence cannot be predetermined and so the size of the array can also not be predetermined. Only the amount of space required for that particular array of strings need be allocated. The actual number of tokens in the sentence is determined in the tokenisation stage. This avoids the problem of a sentence being too long for the array allocated and means that large memory spaces need not be allocated, if only small ones are required.

The character stream ‘Englebert's 4th dog died.’ would have been tokenised in the following way:

.Englebert’s4thdogdied.

and memory would have been allocated for a dynamic array of character pointers, of length 7.

| . | Englebert | ‘s | 4th | dog | died | . |

Figure 5.4 Tokenising a sentence.
(in reality, a pointer to the first letter of a string is stored, but for transparency here, the array will hold the token strings)

5.2.4 Building the Lattice

The lattice is a fully dynamic array of arrays. Neither its width, nor the heights of any of its columns are predefined. It grows to accommodate a sentence of any length, each word having as many or as few tags as is necessary.
The output from the tokeniser, an array of strings (in C a string is stored as a pointer to the first character of an array, so the array is fundamentally a pointer to the first element of an array of pointers, with each of these pointers pointing to the first element of an array of characters.), forms the horizontal axis of the lattice.

The lattice is stored as a structure. This structure holds the width of the lattice, and an array of pointers to columns. Each token in the sentence has its own column in the lattice.

Each column is stored as a structure having a height and an array of pointers to part of speech structures. These part of speech structures are the states in the HMM and there is one of these stored in the column for each possible tag for a word.

A part of speech structure has four elements: a pointer to the word, a pointer to one of the tags for that word, the corresponding current probability for that tag, and a back pointer to the most probable previous tag (which is another part of speech structure in the preceding column).

To build a column of the lattice for a token, the Prob_tags_given_word database (as discussed in 2.3.3.1) is accessed. The word is the key and the data returned is a string of all probable tags and their probabilities, in the form

```
```

(These are the tags and basic probabilities for the word ’died’.)

The column height is set to be the number of tags that feature in the string. This string is then split and each tag and its corresponding probability is sequentially stored in a part of speech structure and entered into the lattice. The back pointers are not set at this point.
If an unknown token is encountered, the process outlined in chapter 4 begins. Because the hapax legomena information is not dependent on the word in question but rather is the same for all unknown words, and because there are only two alternatives for the capitalisations factor in the calculation, two arrays could be predefined, one holding all the tags and their probabilities for capitalised and one for non-capitalised hapax legomena. These were extracted from the corpus and presented in the same format as output from the database. These tags (30 for capitalised words and 24 for non-capitalised words) can then be used to build a column in the array for the unknown word in the exact same way as a normal column is built.

After this has been built, the tags can be coloured with word specific probabilities, based on the suffix of the unknown word. This brings the calculations into line with the algorithm for unknown word calculation as outlined in 4.3.3.

Using the largest suffix of the word that can be found in the database of suffix probabilities (see 4.2 for an explanation of this procedure), a string of all tags that that suffix occurred with is returned. This is of the form:

```%
%%NP0[1]
```

(This is the probability returned for the largest suffix of the word ‘Englebert’, which was ‘ebert’, as ‘lebert’ did not feature in the corpus.)

These tags are counted, and memory is allocated for another dynamic array of structures to hold them (string splitting and token counting takes into account that they are in a different format to strings returned from regular word lookup). They then appear in the same format as a column of the lattice.

A function (given in Figure 5.5) then compares the current column in the lattice, and this newly built suffix array. It compares each tag and if they are the same then it multiplies the probability of that tag in the lattice by the probability of that tag occurring with the suffix. This then means that the tags in the column are then
coloured with word-sensitive information. Thus the tags that occurred with the suffix then have a higher probability than the tags that did not. The other tags are not discounted (as suffix information should not be fully determinant of the probable tag), but rather are not now as likely as they were before.

```c
int apply_affixes(int no_of_tags, int no_affixtags)
{
    int i, j;

    for(i=0; i<no_of_tags; i++){
        for(j=0; j<no_affixtags; j++){
            if(strcmp(posinfo_ptr[i].tag, affix_ptr[j].tag) == 0)
                posinfo_ptr[i].prob = (posinfo_ptr[i].prob * affix_ptr[j].prob * 100);
        }
    }

    return 1;
}
```

Figure 5.5 The function to apply the suffix probabilities

For the purpose of this calculation, it can be observed that if the probabilities of the tags from the hapax legomena list were simply multiplied by the suffix probabilities (which are all less than 0) for those particular tags, then these tags would be less probable than the tags that were not observed with the suffix. This would defeat the purpose of applying the suffix colourings. Different methods of combating this problem were tested. It was observed that if the probabilities of the suffix tags were multiplied by 100, so that instead of all adding up to 1, they added up to 100, then the affix probabilities did not fully determine the tag chosen, but did make the relevant tags far more likely (the same outcome could have been achieved by reducing the probabilities of the tags that did not apply to the suffix, by 100).

This process of column building continues until a column has been built for every token in the sentence.
5.2.5 Calculating the Most Probable Path

After the lattice has been built, computing the maximal path through the lattice using the Viterbi algorithm then disambiguates the sentence. This involves setting the back pointers for each of the tags so that paths are set up through the lattice. The equations behind this procedure are given in chapter 2.

The function proceeds as follows: starting with the first tag of the second column (as the ambiguity of the tag in the first column has been obviated) the calculation: Probability of tag in previous column * the probability of the tag in the previous column being followed by the present tag in present column * the probability of the present tag, is carried out and the back pointer is set to point to this previous tag. This calculation must be carried out for all tags in the previous column, and the one with the highest probability of being the tag preceding the present tag is stored. This same calculation is then carried out for each of the part of speech structures stored in
the present column. After this is complete for the column, each structure has it’s back pointer pointing to the most probable previous tag for that tag.

The same calculation is then carried out for each of the remaining columns, with the probabilities stored being not just the probabilities of the most probable tags for the present bigram, but rather the probability of the entire path up to and including that tag, as indicated by the back pointers.

Thus the whole lattice is swept through, with all back pointers being set along the way. Once the probabilities for the final column have been calculated, the best path through it can then be found. The most probable tag for the final word is found (however its probability is the probability of the most likely path, which finishes at that tag) and the tag is stored in the final position of a new array, which is the same length as the lattice array. The back pointer of that tag is then followed, back to the most previous column and this tag is then stored in the second last place. This procedure continues, discovering the most probable path of tags, until the first column of the lattice is encountered. This is then returned as the most probable path.

An example of the lattice and the most probable path returned can be seen in the tagger output in Figure 5.7. In making this transparent and allowing the lattice to be printed to the screen, one knows not just the most probable part of speech for each word, but also the second most probable etc. One can also see how great the difference is between the first choice and the second etc. So while the tagger returns the best overall tag sequence, one is able to view all alternatives at each step of the way.
<table>
<thead>
<tr>
<th>word</th>
<th>tags &amp; probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>PUN (1.000000e+02)</td>
</tr>
</tbody>
</table>

Englebert
- NP0 (3.043788e+07)
- NN1-NP0 (0.000000e+00)
- NN1 (3.326052e+02)
- NN2 (2.841722e+02)
- VVD (2.685248e+00)
- CRD (4.018250e+01)
- AJ0-VVG (0.000000e+00)
- AJC (9.387579e-01)
- AJS (8.412513e-02)
- CJS (1.435131e+01)
- AJ0-VVR (0.000000e+00)

's
- POS (1.506298e+10)
- VBZ (8.879993e+08)
- VDZ (1.198022e+04)

4th
- ORD (4.559978e+09)

dog
- NN1 (6.380771e+12)
- NN1-VVB (0.000000e+00)

died
- VVN (5.809973e+13)
- VVD (4.051675e+15)

.    | PUN (5.071958e+18) |

The most probable tag sequence is:
- word : Englebert    tag : NP0    (Proper noun )
- word : 's           tag : POS    (Possessive marker )
- word : 4th          tag : ORD    (Ordinal numeral )
- word : dog          tag : NN1    (Singular common noun )
- word : died         tag : VVD    (Verb - past tense form)
- word : .            tag : PUN    (Punctuation: separator)

No. of unknown words encountered : 1
Would you like a different type of analysis? (y for yes , n for no):

Figure 5.7 Tagger output (showing lattice) for a sentence.

The output is either printed to the terminal screen or to file and is displayed in the following format; where each sentence is marked with <sent> at the beginning and
end and the most probable tag for each word appears after it, also enclosed in angle brackets.

\(<\text{sent}>\text{ No}<\text{ITJ}>!<\text{PUN}>\text{<sent>}\)

Each sentence is placed on a new line, to aid transparency.

## 5.3 Other considerations

When building the tagger, many decisions as to how various aspects of the system should be implemented had to be made. Outlined in this section are a few of such considerations and the reasons why the paths taken were pursued.

### 5.3.1 Storing the Lattice

In designing the lattice, there was a choice of storage methods. The one opted for, in which the entire lattice is stored for the duration of the tagging process, or storing just the current column and its previous column. All that is really needed for disambiguation is the most previous column, as the probabilities for the tags in that preceding column, are actually the probabilities for the most probable path to that point. An array of tags could have been set up to hold the most probable tags sequence up to the preceding column and as each column is calculated, the previous one could have been broken down.

While this method would have worked well within the taggers current functionality, and would have saved on space requirements it seemed like a better option to allow the lattice and the tagging sequence to be as transparent as possible. If this were a pure performance oriented situation, where memory was at a premium or greater speed was necessary then this would have been a better option, but there are also arguments in favour of storing the entire lattice
If the disambiguation process were to be expanded to greater depths, say, to take trigrams or n-grams into consideration, then the transparency of the lattice in its current form would make this much easier to implement. Also the memory requirements for storing the lattice were well handled, as each lattice was built, purely for the purpose of tagging a single sentence and once this procedure was complete, all memory used was promptly freed up and reclaimed, for use for the next sentence.

5.3.2 Retaining Capitalisation of First Word

Automatic de-capitalisation of the first word of each sentence was considered, as in general it is much more probable for a word to appear in its non-capitalised form, than it would be for it to appear capitalised. The pros and cons of this approach were considered and finally the option of maintaining the capital letter was decided upon. Some motivations for this are given below:

- In languages, such as English, where only auxiliary verbs can invert (go to head of the sentence), there is a very high likelihood that all of these auxiliary verbs, which form a closed set in English, will already have been encountered in a capitalised form.
- It can be observed that when nouns appear at the head of a sentence, they are generally proper nouns, in which case they should be capitalised anyway, and de-capitalising them would mean that they would loose the chance of being found in the corpus. Usually normal singular nouns do not appear in the first position in a sentence, but rather appear after a determiner. (with the exception of some genre e.g. newspapers (e.g. “Child born with 13 toes.”))
- Closed class words, which form a small subset of all distinct words in the corpus, will usually have been encountered in the corpus in their capitalised form already.
5.3.3 De-capitalisation

In addition to adopting the approach of retaining the capital letter on leading words for lookup, the option was also given for later de-capitalisation. This was done by providing the condition that if a word could not be found in the initial dictionary lookup, it was then de-capitalised and the process was repeated. It is much more probable for a common word to appear in its non-capitalised form (as mentioned in 5.3.2), and if it is found then the corresponding tags for such the common word are used. If on the other hand, neither the capitalised nor the non-capitalised form were found, then the word remained capitalised and was treated as an unknown word.

5.3.4 Value placed on suffix information.

Various weightings could have been placed on the word-specific, suffix information. Taggings were carried out with the probabilistic value of the suffix being multiplied by 10 (making the suffix information less determining) and by 100. Tests were run where the suffix was the determining factor, that is, the hapax legomena and capitalisation information of only the tags that featured with the suffix in the corpus, were considered. This was seen to be too restrictive, since the largest (thus less frequent in the database, as smaller suffixes have a wider scope of words to base their probabilities on) suffix, sometimes does not feature with the correct tag, but a smaller suffix of that suffix might.

An example of this is: if a suffix of an unknown word of length six, is taken, and this suffix appeared only once in the corpus, then the tag associated with it in the corpus will be the only one considered. This was observed to be too restrictive for the task in hand.
Tests were also conducted on the effect of varying the length of the suffix considered. It was found that if the largest suffix considered was reduced from length six to length three, then the overall result did not change very much. A reason for this is that the word with the suffix of length six and its corresponding tags is also considered for the calculations of its smaller suffixes of that word. The amount of tags for a suffix increases as the suffix gets smaller. More of the tags were coloured (as the smaller the suffix, the more tags it occurs with) when only suffixes of length three or smaller were considered. In the end the longest suffix length considered was six. This is found to work extremely well in helping to discern the most probable tag for an unknown word.

5.3.5 Dividing the Portmanteau tags.

It has been shown that there are many approaches and methods adopted in fulfilling the task of part of speech tagging. In the initial introduction it was stated that the purpose of part of speech tagging is to assign each word its most probable tag. There are certain circumstances however, where some taggers are willing to relax the ‘one tag per word’ requirement in order to decrease the probability that an incorrect tag will be assigned to a word. In the automated tagging of the BNC, the CLAWS tagger opted for this approach and returned a so-called Portmanteau tag when the probabilities of different tags were too close for it to securely choose between them.

In my approach I decided that such tags were unsatisfactory and the single most probable previous tag should be returned. However, as these portmanteau tags act in just the same manner as true tags in the corpus, a way of eradicating them, while still using the information they hold had to be found.

Seeing as they indicate that the CLAWS tagger was wavering between the two tags and that the probabilities for these tags were too close to choose between them, the
tagger simply takes the portmanteau probability and splits it evenly between the two tags that feature in it. This is illustrated in figure 5.8:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag : NN1</td>
<td>0.5</td>
</tr>
<tr>
<td>Tag : NP0</td>
<td>0.2</td>
</tr>
<tr>
<td>Tag : NN1-NP0</td>
<td>0.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag : NN1</td>
<td>0.65</td>
</tr>
<tr>
<td>Tag : NP0</td>
<td>0.35</td>
</tr>
<tr>
<td>Tag : NN1-NP0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.8 Dividing the Portmanteau tags

The tagger output in Figure 5.7 shows the portmanteau tags having been reduced to 0 and the probabilities distributed between the relevant tags.
Chapter 6.

Scoring and Evaluation

*The end must justify the means.*

*(Mathew Proir)*

This chapter outlines the functions that allowed the tagger to be observed during the tagging process. It also shows how the tagger’s performance and speed were both automatically and manually evaluated.

### 6.1. Observing the Tagging Process

A useful functionality of the tagger is that if the option is chosen (from one of the initial menus), the user is able to observe the behaviour of the tagger during tagging. One can see calculations made and choices followed, at each stage in the process. This is practical as it allows the user to see what influences various probabilistic factors had on the tagging process.

For example, say the tagger is trying to decide between two tags for a certain word. The user can then observe the Viterbi calculations for both, and can see whether it was the bigram, the lexical or the most probable previous tag’s probability that caused the tagger to favour one tag over another.

This function is also useful for the developer, as they can enter sentences that the tagger tagged incorrectly and observe what caused the tagger to choose an incorrect path. They can then take this into consideration and try to alter the calculation weightings accordingly.
An extract from the output generated by the function that lets the user see the path taken through the lattice and that give valid reasons why each choice was made in the tagging process is shown in figure 6.1:

```plaintext
the original tag(s) for 'ice-cream' are
  tag:NN1                prob:9.580420e+01
  tag:AJ0-NN1            prob:0.000000e+00

Calculating path to tag NN1 from tag AT0
  Probability of NN1 being the tag for the word 'ice-cream': 9.580420e+01
  Probability of NN1 given that previous tag was AT0: 9.836640e+05
  Probability of an (the previous word): 5.269227e+33
  Overall probability of ice-cream having tag NN1: 4.965674e+41
  *AT0 is the most probable previous tag for 'ice-cream'
  when 'ice-cream' has 'NN1' as it's tag.

Calculating path to tag NN1 from tag CJC
  Probability of NN1 being the tag for the word 'ice-cream': 9.580420e+01
  Probability of NN1 given that previous tag was CJC: 9.836480e+05
  Probability of an (the previous word): 4.291530e+28
  Overall probability of ice-cream having tag NN1: 4.044235e+36
  The most probable previous tag not changed

Calculating path to tag NN1 from tag UNC
  Probability of NN1 being the tag for the word 'ice-cream': 9.580420e+01
  Probability of NN1 given that previous tag was UNC: 9.836320e+05
  Probability of an (the previous word): 4.291530e+28
  Overall probability of ice-cream having tag NN1: 4.044169e+36
  The most probable previous tag not changed
```

Figure 6.1 Observing the tagging process.

### 6.2 Empirical evaluation of the Tagger

The most important feature in the evaluation of a part-of-speech tagger is accuracy. The quality of the output depends on comparability of conditions such as:

- The size of the tagset used: a smaller tagset ensures more accurate tagging, but does not offer as much information or disambiguation between lemmas as a larger one would.
• The type of corpus used for training and testing affects the quality of a
tagger’s output in different ways. Accuracy differs when the genre or type of
the corpus data differs from the material to be tagged.
• Complete or incomplete vocabulary: Accuracy is higher when all the input
words feature in the corpus, than it would be if the system has an incomplete
vocabulary and the system has to disambiguate the unknown word using other
statistical methods.
• Type of vocabulary: the tagging of subject specific texts e.g. medical or legal
texts would demand a training corpus that contained examples of such texts
(otherwise the number of unknowns would be unnaturally high). Also the
high instance of idiomatic expressions in literary texts often leads to
inaccuracy.

In order to evaluate the performance of the tagger, its performance had to be tested on a
number of different texts. Two methods of evaluation were chosen: pre-tagged texts
were obtained and these were manually scored for correctness, and post correction was
manually performed on various different unseen texts.
During the process the number of unknown words encountered was also recorded, so as
to observe whether results were poorer for texts containing many unknown words. The
findings are documented below and the texts used for performance testing can be seen
in Appendix B.

6.2.1 Automatic Evaluation

Evaluation of the accuracy of the tagger can be done by comparing a manually pre-
tagged text, tagged using the same tagset, to the output that the tagger would return, if
required to tag the same text.
To perform this, medical texts, which were tagged correctly using the C5 tagset, were used. An example of such a text can be seen in Appendix B, Text 1. The text and the tag appeared in the following format:

```
ELE1alpha:NP0 is:VBZ expressed:VBN in:PRP the:AT0 three:CRD prostate-derived:AJ0 cell:NN1 lines:NN2
```

where each word is followed directly by its appropriate tag. All punctuation was removed and the each new sentence was placed on a new line.

The evaluation proceeded in the following way:
- the file containing the pre-tagged text was opened and the first line was extracted.
- An array of part of speech structures was set up to hold each word and its corresponding correct tag and an array of strings was set up to hold the tokens themselves. This second array appears then in the same format as the tokeniser output from a normal tagging.
- From this second array, which holds the words of the text themselves, the usual tagging procedure continues: a lattice is built, calculations are performed, a best path is returned and output is printed to file in the format that the tagger itself uses.
- This best path, which is an array of tag structures, each having a word and a tag, is then compared to the original correct tag sequence (as stored in the first ‘correct tags’ array built.
- Calculations are performed and results are returned. An example of such results can be seen in figure 6.2
6.2.1.1 Discussion of Automatic Evaluation

The automatic tagging procedure was performed on biomedical texts having an unnaturally high number of unknown words. The percentage of unknown words featured was almost 17%. The genre of the texts was extremely scientific, whereas the BNC is composed of texts from a huge range of different genre, with a high proportion of transcribed spoken conversations, newspapers, novels, essays, letters etc.

In a post observation on the errors made by the tagger it can be seen that no errors were made on closed class tokens. Errors made were approximately 50:50 unknown words: uncommon, but known words, appearing here as compound noun phrases. An example of this is the compound noun phrase ‘Mad bHLH-zip proteins’, with its original tagging being:

Mad:NP0 bHLH-zip:NP0 proteins:NN2

The tagger tagged ‘Mad’ and ‘bHLH-zip’ as adjectives that qualify a plural noun, but both words are in fact proper nouns in that context. The tagger can be seen to only have made mistakes on these constructions specific to medical-texts.
This would indicate that, had the text been written in the same genre, as that of the tagger or vice versa, had the corpus been made up of tagged medical texts, then the tagger would have performed much better.

Despite these factors, the tagger achieved an accuracy rate of over 95.5% for these texts.

6.2.2 Manual Evaluation

The tagger is built to robustly handle any kind of text given to it as best it can. It was therefore tested with a range of different text types, to see how it coped.

A count of the words was made during the tagging process and the output was manually checked to see if the correct tags had been assigned to the tokens in the text.

Three of such texts can be seen in Appendix B (texts 2, 3 and 4) and the taggings produced from them as texts 2(a), 3(a) and 4(a). These texts are from three very different genres, the first is a newspaper report, the second, a children’s poem with made-up words and the third, an informal piece, with extracts of conversation.

6.2.2.1 Discussion of Manual Evaluation

Tagging the text in Text 2 resulted in 97.5% accuracy. There were no unknown words in this text and out of the 13 tokens tagged wrong, most were part compound noun phrases, which when broken down into their constituent words, produced incorrect taggings. An example of this is the compound proper noun ‘The Royal Chapel of All Saints’ which was tagged as:

the<AT0> Royal<AJ0> Chapel<NN1> of<PRF> All<DT0> Saints<NN2>
Also, in this journalistic piece (taken from The Irish Times) there were inconsistencies in capitalisation e.g. ‘the Queen Mother’ is capitalised some of the time, and later on is not. Such errors led to a less than accurate tagging.

Text 3, a children’s poem, can also be seen in Appendix B. It contains an unusually high number of unknown words, but they are all constructed having regular morphological suffixes. The tagger is seen to correctly tag 100% of these unknown words.

Apart from one incorrectly assigned tag, the three other errors (as shown in bold print in the tagging output 3(a)) were made in tagging tokens with irregular punctuation.

In printing text 4 to file, the function that extracts the largest suffix for an unknown word was made visible. Every time the suffix database was accessed, the input was printed. This allows the user to see the largest suffix considered, and watch it shrink until one is found.

In the output text 4(a), these unknown words are highlighted and the suffix breakdown for each unknown word is given before the sentence it features in is printed.

In observing these unknown words, we can see that only one error is made, that being in the final unknown word. This shows that the unknown word algorithm is adept at dealing with any unknown words that can be thrown at it!

These random texts underwent no pre-processing. In observing the output it is easy to see how the tagger handled other aspects of tagging and segmenting, not captured by the automatic evaluation.

Sentence segmentation was, in general, well captured, with the exception of one type of sentence structure – a quote. An example of a segmentation error occurred in the following piece of text:
Mr Ahern, in a message of condolence, said: "The queen mother......

Here the quotation marks after ‘sympathies.’ are actually part of the first sentence, but since tagging does not take new line markers into consideration (a method adopted as it generally makes tagging more robust and accurate), and since they occurred after the end of sentence marker, they were taken to be part of the second sentence.

This is a difficult problem to solve without resorting to a more rule based approach, which would take away from the simplicity and robustness of the probabilistic approach employed. Also, as mentioned previous, sometimes such punctuation is used at random, and would not follow the prescribed rules.

In general the tagger is seen to effectively handle segmentation. This is a virtue of the tagger, as, in the literature, it is explained that text is generally processed before inputting it to the tagger, so that it has one sentence per line.

### 6.2.2.2 Problems with Portmanteau Tags

Despite including a function to eradicate all portmanteau tags from being considered as part of the best path, it can be seen that if they are the only option, then they will still be returned. If the CLAWS tagger repeatedly assigned only a portmanteau tag to a given token, then this will be the only tag to feature in the column, and thus the probability cannot be split between the two relevant tags.

An example of this is in the poem The Jabberwocky (see Text 5). This contains a large number of made-up words, used only in the context of this poem. This text is obviously part of the BNC and the tagger could not split the portmanteau tags assigned to some of the unknown words. The reason for this is that it is built in such a way that it can only split the tags contained in a portmanteau tag, if one or both of these tags also feature in the column. It is usually the case that one or both of the tags in the portmanteau will also feature for that word as single tags elsewhere in the corpus. This is not so for this
text, as these words only appeared once in the corpus, and so the tags could not be disambiguated into their constituent parts. This is a shortfall of the BNC. It can be seen that manually tagged corpora can often give more conclusive results.

The portmanteau tags that could not be resolved are shown in bold print in Text 5(a) in Appendix B.

6.3 Timing the tagger

All timings reported are were recorded on a machine with the following specifications:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>Sun (Sun Microsystems)</td>
</tr>
<tr>
<td>System Model</td>
<td>Blade 100</td>
</tr>
<tr>
<td>Main Memory</td>
<td>128 MB</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>328 MB</td>
</tr>
<tr>
<td>CPU Type</td>
<td>sparcv9+vis</td>
</tr>
<tr>
<td>OS Name</td>
<td>SunOS</td>
</tr>
<tr>
<td>OS Version</td>
<td>5.8</td>
</tr>
<tr>
<td>Processor</td>
<td>&quot;Sun UltraSPARC-IIe&quot;</td>
</tr>
<tr>
<td>Speed of Processor</td>
<td>502 MHz</td>
</tr>
</tbody>
</table>

Figure 6.3 Machine Specifications.

The tagger currently tags between 1000 and 1200 tokens per second (including file I/O). The speed mainly depends on the percentage of unknown words and the average ambiguity rate.

An example of a timed run is given in Figure 6.4. This is a timed tagging of the first 3 chapters of Alice in Wonderland.
Scoring and Evaluation

Statistics
Time taken: 11 seconds for 10997 words of text
No. of unknown words encountered : 143
No. of sentences encountered (no. of lattices built) : 463
Speed of tagger : 999 words/sec

Figure 6.4 A timed tagging of a large file.

6.4 Memory De-allocation

Testing had to be carried out to check if the tagger was able to perform the various memory allocation and deallocation functions correctly, to ensure that minimal memory was required to perform even the largest tagging. Tests were run, using a large file of 463 sentences (11,000 words), and in the building of these 463 lattices, the memory requirements increased for the first sentence, but remained static thereafter. The functions to perform the memory de-allocation were removed (for testing purposes) and the memory requirements were seen to rise steadily.

The stages in the test process:
1) before the file was run
2) with all the free statements included - the 656K RES doubled (for first lattice) but then remained constant.
3) without the free statements - the RES was constantly increasing and ended up being 8 times its original size.

Figure 6.5. Verification of memory de-allocation.
6.5 Observing the Databases

The process of observing the tagger, and discerning where errors were made, it was often advantageous to see what tags and probabilities existed in the corpus for a given word or suffix or to see probabilities for bigram transitions. For this reason I implemented another module to allow the user to see these quickly and easily. An example of a run of this module can be seen in Figure 6.6.

```
Which database would you like to access?
For 'all tags given a certain word' type 1
For 'bigram probabilities' type 2
For 'all tags given a certain word (based on it's suffix only)' type 3
Please type 1,2 or 3 - 1

Please enter the word : can

The tags for that word are :
.VM0[0.457838409973].NN1[0.3751617076326].VVB[0.135246383629307].AJ0-[NN1[0.00352816652946019].NN1-VVB[0.0270492767258615].VVI[0.00117605550982006]

Which database would you like to access?
For 'all tags given a certain word' type 1
For 'bigram probabilities' type 2
For 'all tags given a certain word (based on it's suffix only)' type 3
Please type 1,2 or 3 - 2

Please enter the first tag of the bigram) : NN1
Please enter the second tag of the bigram) : NP0
The probability of that bigram is : 0.550029

Which database would you like to access?
For 'all tags given a certain word' type 1
For 'bigram probabilities' type 2
For 'all tags given a certain word (based on it's suffix only)' type 3
Please type 1,2 or 3 –3
Please enter the word : pling
suffix was : pling
The tags for that largest suffix of that word are :
%%VVG[0.265625]%%NN1-VVG[0.140625]%%AJ0-NN1[0.1328125]%%AJ0-VVG[0.125]%%NN1[0.1171875]%%AJ0[0.109375]%%NP0[0.078125]%%NN1-NP0[0.03125]
```

Figure 6.6 Observing the Databases
Chapter 7.

Conclusion

“Language, Timothy.”

(Ronnie Corbett)

(from the TV series ‘Sorry!’)

In this chapter I will outline some improvements that I think would enhance the performance of the tagger. I will also sum up with an overview of the last 6 chapters, finishing with a conclusion.

7.1 Potential for Future Work

This tagger has great potential for future work. A number of modifications and improvements were outlined in the Aims in section 1.1. Some things that I would work on include: greater disambiguation of capitalised words, pre-processing of multi word units (e.g. ‘ad hoc’ as a single token), and using trigrams to aid disambiguation. These are outlined below:

7.1.1 Greater disambiguation of capitalised words.

Accounting for the immediate context of a capitalised word could enhance the lexicon lookup strategy adopted here. Mikheev ([Mik 97] showed that capitalised words in their ambiguous positions are not easily disambiguated by their surrounding past of speech context as attempted by taggers in general. For instance, many surnames are at the same time nouns or plural nouns in English and thus in both variants can be followed by a past tense verb. The capitalised words in the phrases ‘Sails rose.’ or ‘Flowers blossomed.’ Can easily be interpreted in either way and only knowledge of semantics disallows the plural noun interpretation of ‘Stars can read.’
Another challenge, beyond the scope of the tagger at present is to decide if the first capitalised word belongs to the group of the following proper nouns or is merely an external modifier and therefore not a proper noun. For instance, *One World* is a single proper noun phrase, but in *One Microsoft program...* the word “One” is an external modifier and can be safely de-capitalised. The document-cantered approach mentioned earlier would be a relatively straightforward addition to make to the tagging procedure. It would be implemented as part of the pre-tokenisation stage.

### 7.1.2 Pre-processing of MWUs

The problem of orthographic words being composed of more than one token was outlined in chapter 3. The opposite problem, the many-to-one correspondence between strings of words, acting together as one word and orthographic words is posed by certain complex units (at different levels) which in spite of spanning over several words, should be analysed (i.e. tokenised) as a single one.

These units have been globally termed multi-word units, MWUs in the literature. These are fixed phrases and can occur as the following types: conjunctions, adverbs, prepositions, and proper nouns e.g. ‘New York’, ‘in terms of’.

A solution given to this problem has been to store these fixed phrases in the corpus as one word (without blanks within) with a special character (tilde, ~) joining the different words forming the fixed phrase. But, obviously fixed phrases do not appear in texts in such a way, so it is necessary to pre-process the texts to be tagged, in order to detect the possible fixed phrases and transform them into the form in which they are stored in in the lexicon.

A hand made list of such fixed phrases (with blanks in between) and their corresponding tags could be easily extracted from the corpus (as the BNC tags them as single tokens) and stored in the modified format i.e. without blanks and with their constituent words joined by the special character.
The text to be tagged need only then be scanned through, and compared with the list of MWUs. If any of these set phrases occur in the text, their constituent words could be joined. This would allow the MWUs to be correctly handled by the tokeniser and treated as if they were single tokens thereafter for the purpose of tagging.

The only problem with this approach is that sometimes a phrase that has all the constituents of a MWU should in fact not be joined, but rather, the constituent parts tagged individually. Also, sometimes the CLAWS automatic tagger of the BNC did not handle such constructions correctly.

But this area in general has a wealth of possibilities that could be explored.

### 7.1.3 N-grams for further disambiguation.

The system could potentially be expanded to use trigrams or even greater still for n-grams for further disambiguation. There are pros and cons to adopting this approach. The number of parameters seems quite large to reliably estimate from corpus frequencies. Given that our tagset has 61 normal tags and 30 portmanteau tags (these have to be taken into consideration here as they feature in the original corpus), 753,571 contextual parameters would have to be estimated from the corpus. On the other hand, the use of trigrams in addition to bigrams would enhance the probability of being able to infer a token’s tag due to its context.

### 7.2 Summing up

In the course of this paper I have attempted to give an insight into the process I underwent in implementing a fully functional part of speech tagger. The aims and an overview of the project were given in chapter 1. A scientific overview of the various methodologies adopted in part of speech tagging and the one finally implemented by
me, were outlined in chapter 2. Chapters 3 and 4 explain various problems and considerations in part of speech tagging and how they were handled for the purpose of this project. Chapter 5 proceeds to explain the implementation of the tagger. This explanation abstracts away from the actual technical coding details of the implementation. For an insight into these, please see the extensive comments on the code itself (Appendix C). Chapter 6 then tells how the tagger performed in a variety of tests, both manual and automatic. This chapter then outlines further modules of the project that could potentially be implemented, and finally, sums up the project.

The aims of the project were reached. As mentioned in section 1.1, the goal was to implement the basic tagger and to work on two other issues from a given list. In the end, I was able to implement three of them: having aimed for dealing with tokenisation and segmentation, as well as unknown words, I was also able to build an evaluation model to test the performance of the tagger.

Average part of speech tagging accuracy is between 95% and 98%, depending on genre. The accuracy is slightly better for texts containing mostly known tokens than it is for ones having a high percentage of unknowns. A more decisive factor in determining the accuracy of the output is the genre of the text being handled. If the style of the text is not strictly specific to one discipline (i.e. not legal or medical etc.) then the tagger performs much better. It is robustly built to handle issues of ambiguous punctuation and segmentation. The level of accuracy achieved is on a par with state-of-the-art tagger results found in literature.
Bibliography


Bibliography


Web resources used

Tagset Information: http://www.hcu.ox.ac.uk/BNC/what/gramtag.html
Alice in Wonderland: http://www.cs.indiana.edu/metastuff/wonder/wonderdir.html
Queen Mother Text: http://www.ireland.com/newspaper/world/2002/0401/1451261988FRMUM2.html
Eletelephony: http://www.cswnet.com/~erin/child.htm#Eletele
Skateboarding text: http://www.observer.co.uk/life/story/0,6903,615634,00.html
Appendix 1.

The BNC Tagset

Each tag consists of three characters. Generally, the first two characters indicate the general part of speech, and the third character is used to indicate a subcategory. When the most general, unmarked category of a part of speech is indicated, in general the third character is 0. (For example, AJ0 is the tag for the most general class of adjectives.)

AJ0 Adjective (general or positive) (e.g. good, old, beautiful)

AJC Comparative adjective (e.g. better, older)

AJS Superlative adjective (e.g. best, oldest)

AT0 Article (e.g. the, a, an, no) [N.B. no is included among articles, which are defined here as determiner words which typically begin a noun phrase, but which cannot occur as the head of a noun phrase.]

AV0 General adverb: an adverb not subclassified as AVP or AVQ (see below) (e.g. often, well, longer (adv.), furthest. [Note that adverbs, unlike adjectives, are not tagged as positive, comparative, or superlative. This is because of the relative rarity of comparative and superlative adverbs.]

AVP Adverb particle (e.g. up, off, out) [N.B. AVP is used for such "prepositional adverbs", whether or not they are used idiomatically in a phrasal verb: e.g. in 'Come out here' and 'I can't hold out any longer', the same AVP tag is used for out.]

AVQ Wh-adverb (e.g. when, where, how, why, wherever) [The same tag is used, whether the word occurs in interrogative or relative use.]

CJC Coordinating conjunction (e.g. and, or, but)

CJS Subordinating conjunction (e.g. although, when)

CJT The subordinating conjunction that [N.B. that is tagged CJT when it introduces not only a nominal clause, but also a relative clause, as in 'the day that follows Christmas'. Some theories treat that here as a relative pronoun, whereas others treat it as a conjunction. We have adopted the latter analysis.]

CRD Cardinal number (e.g. one, 3, fifty-five, 3609)
Appendix 1 – The Tagset

DPS Possessive determiner (e.g. your, their, his)

DT0 General determiner: i.e. a determiner which is not a DTQ. [Here a determiner is defined as a word which typically occurs either as the first word in a noun phrase, or as the head of a noun phrase. E.g. This is tagged DT0 both in 'This is my house' and in 'This house is mine'.]

DTQ Wh-determiner (e.g. which, what, whose, whichever) [The category of determiner here is defined as for DT0 above. These words are tagged as wh-determiners whether they occur in interrogative use or in relative use.]

EX0 Existential there, i.e. there occurring in the there is ... or there are ... construction

ITJ Interjection or other isolate (e.g. oh, yes, mhm, wow)

NN0 Common noun, neutral for number (e.g. aircraft, data, committee) [N.B. Singular collective nouns such as committee and team are tagged NN0, on the grounds that they are capable of taking singular or plural agreement with the following verb: e.g. 'The committee disagrees/disagree'.]

NN1 Singular common noun (e.g. pencil, goose, time, revelation)

NN2 Plural common noun (e.g. pencils, geese, times, revelations)

NP0 Proper noun (e.g. London, Michael, Mars, IBM) [N.B. the distinction between singular and plural proper nouns is not indicated in the tagset, plural proper nouns being a comparative rarity.]

ORD Ordinal numeral (e.g. first, sixth, 77th, last). [N.B. The ORD tag is used whether these words are used in a nominal or in an adverbial role. Next and last, as "general ordinals", are also assigned to this category.]

PNI Indefinite pronoun (e.g. none, everything, one [as pronoun], nobody) [N.B. This tag applies to words which always function as [heads of] noun phrases. Words like some and these, which can also occur before a noun head in an article-like function, are tagged as determiners (see DT0 and AT0 above).]

PNP Personal pronoun (e.g. I, you, them, ours) [Note that possessive pronouns like ours and theirs are tagged as personal pronouns.]

PNQ Wh-pronoun (e.g. who, whoever, whom) [N.B. These words are tagged as wh-pronouns whether they occur in interrogative or in relative use.]

PNX Reflexive pronoun (e.g. myself, yourself, itself, ourselves)
Appendix 1 – The Tagset

POS The possessive or genitive marker 's or '(e.g. for 'Peter's or somebody else's', the sequence of tags is: NP0 POS CJC PNI AV0 POS)

PRF The preposition of. Because of its frequency and its almost exclusively postnominal function, of is assigned a special tag of its own.

PRP Preposition (except for of) (e.g. about, at, in, on, on behalf of, with)

PUL Punctuation: left bracket - i.e. ( or [

PUN Punctuation: general separating mark - i.e. . , ! , : ; - or ?

PUQ Punctuation: quotation mark - i.e. ' or "

PUR Punctuation: right bracket - i.e. ) or ]

TO0 Infinitive marker to

UNC Unclassified items which are not appropriately classified as items of the English lexicon. [Items tagged UNC include foreign (non-English) words, special typographical symbols, formulae, and (in spoken language) hesitation fillers such as er and erm.]

VBB The present tense forms of the verb BE, except for is, 's: i.e. am, are, 'm, 're and be [subjunctive or imperative]

VBD The past tense forms of the verb BE: was and were

VBG The -ing form of the verb BE: being

VBI The infinitive form of the verb BE: be

VBN The past participle form of the verb BE: been

VBZ The -s form of the verb BE: is, 's

VDB The finite base form of the verb BE: do

VDI The infinitive form of the verb DO: do

VDN The past participle form of the verb DO: done

VDZ The -s form of the verb DO: does, 's
Appendix 1 – The Tagset

VHB The finite base form of the verb HAVE: have, 've

VHD The past tense form of the verb HAVE: had, 'd

VHG The -ing form of the verb HAVE: having

VHI The infinitive form of the verb HAVE: have

VHN The past participle form of the verb HAVE: had

VHZ The -s form of the verb HAVE: has, 's

VM0 Modal auxiliary verb (e.g. will, would, can, could, 'll, 'd)

VVB The finite base form of lexical verbs (e.g. forget, send, live, return) [Including the imperative and present subjunctive]

VVD The past tense form of lexical verbs (e.g. forgot, sent, lived, returned)

VVG The -ing form of lexical verbs (e.g. forgetting, sending, living, returning)

VVI The infinitive form of lexical verbs (e.g. forget, send, live, return)

VVN The past participle form of lexical verbs (e.g. forgotten, sent, lived, returned)

VVZ The -s form of lexical verbs (e.g. forgets, sends, lives, returns)

XX0 The negative particle not or n't

ZZ0 Alphabetical symbols (e.g. A, a, B, b, c, d)

Total number of grammatical tags in the BNC Basic Tagset: 61
Appendix 2.
Texts Used

Text 1
(an extract from the original medical text used for automatic evaluation)

In PRP this DT0 context NN1 ARA70 NP0 previously AV0 called VVD VV\nRFG NP0 and CJC ELE1 NP0 has VHZ been VBN described VVN as PRP a AT0 putative AJ0 coactivator NN1 that CJT specifically AV0 enhances VVZ the AT0 activity NN1 of PRF the AT0 androgen NN1 receptor NN1 AR NN1 but CJC not XX0 that DT0 of PRF the AT0 glucocorticoid NN1 receptor NN1 GR NN1 the AT0 progesterone NN1 receptor NN1 or CJC the AT0 estrogen NN1 receptor NN1 ER NP0

ELE1 alpha NP0 is VBZ expressed VVN in PRP the AT0 three CRD prostate derived AJ0 cell NN1 lines NN2 examined VVD PC 3 NP0 DU 145 NP0 and CJC LNCaP NP0 and CJ C this DT0 expression NN1 is VBZ not XX0 altered VVN by PRP androgen NN1 treatment NN1

Both DT0 ELE1 alpha NP0 and CJC ELE1 beta NP0 interact VVB in PRP vitro NN1 with PRP the AT0 AR NN1 but CJ C also AV0 with PRP the AT0 GR NN1 and CJ C the AT0 ER NN1 in PRP a AT0 ligand independent AJ0 way NN1

Overexpression NN1 of PRF either DT0 ELE1 NP0 isoform NN1 in PRP DU 145 NP0 HeLa NP0 or CJC COS NP0 cells NN2 had VHD only AV0 minor AJ0 effects NN2 on PRP the AT0 transcriptional AJ0 activity NN1 of PRF the AT0 human AJ0 AR NN1

ELE1 alpha NP0 has VHZ no AT0 intrinsic AJ0 transcription NN1 activation NN1 domain NN1 or CJ C histone NN1 acetyltransferase NN1 activity NN1 but CJ C it PNP does VDZ interact VVI with PRP another DT0 histone NN1 acetyltransferase NN1 p CAF NP0 and CJ C the AT0 basal NN1 transcription NN1 factor NN1 TFIIB NP0
Text 1(a)
(the tagger’s retagging of the extract in Text 1)

In this context, ARA70 (previously called RFG and ELE1 has been described as a putative coactivator that specifically enhances the activity of the androgen receptor AR but not that of the glucocorticoid receptor GR, the progesterone receptor or the estrogen receptor. ELE1α is expressed in the three prostate-derived cell lines examined: PC-3, DU-145 and LN CaP and this expression is not altered by androgen treatment.

Both ELE1α and ELE1β interact in vitro with the AR but also with the GR and the ER in a ligand-independent way. Overexpression of either ELE1 isoform in DU-145, HeLa, or COS cells had only minor effects on transcriptional activity of the human AR. ELE1α has no intrinsic transcription activation domain but it does interact with another histone acetyltransferase p/CAF and the basal transcription factor TFIIB.

ELE1α has no intrinsic transcription activation domain but it does interact with another histone acetyltransferase p/CAF and the basal transcription factor TFIIB.
Text 2.
(a newspaper report about the recent death of the Queen Mother)

The queen mother will be sorely missed by her family and by the British nation and we offer them our sincerest sympathies."

Mr Ahern, in a message of condolence, said: "The queen mother had a personal grace and charm which endeared her not only to the citizens of Britain but also to many people here in Ireland."

Queen Elizabeth last night led members of the royal family in prayer and remembrance at the coffin of the queen mother.

Senior royals filed solemnly into the Royal Chapel of All Saints in Windsor Great Park, London, for a brief private service. Prince Charles, said to be "absolutely devastated" by his grandmother's death, and his sons, Prince William and Prince Harry, cut short a skiing holiday in the Swiss resort of Klosters to fly home.

In a break with protocol, the queen allowed Prince Charles, her heir, to travel on the same plane as his sons, something normally avoided by the royal family for security reasons.

The queen mother kept the royal family together through seven tempestuous decades - from the 1936 abdication crisis that propelled her shy husband onto the throne to Prince Charles's bitter divorce from Princess Diana in 1996.

Many schools in Britain will make special arrangements to enable pupils to watch the funeral on television, head-teachers said. Tributes poured in yesterday describing the queen mother as a figure of courage who had won a place in the hearts of millions around the world.
Appendix 2 – Texts Used

Text 2(a)
(tagging of text 2)

<sent>President<NP0> expected<VVD> to<T00> attend<VVI> funeral<NN1> of<PRF> Queen<NN1> Mother<NN1> The<AT0> President<NN1> , Mrs<NP0> McAleese<NP0> is<VBZ> expected<VVN> to<T00> attend<VVI> the<AT0> funeral<NN1> service<NN1> for<PRP> Queen<NP0> Elizabeth<NP0> Queen<NN1> Mother<NN1> on<PRP> April<NP0> 9th<ORD> , Government<NN0> sources<NN2> said<VVD> last<ORD> night<NN1> , though<CJS> a<AT0> final<AJ0> decision<NN1> has<VHZ> not<XX0> yet<AV0> been<VBN> made<VVN> .</PUN> <sent>

<sent>The<AT0> decision<NN1> will<VM0> be<VBI> seen<VVN> as<PRP> a<AT0> further<AV0> deepening<AJ0> of<PRF> relations<NN2> between<PRP> the<AT0> Republic<NN1> and<PRP> Britain<NP0> .</PUN> <sent>

<sent>In<PRP> December<NP0> 1999<CRD> , President<NN1> had<VHD> lunch<NN1> with<PRP> Queen<NP0> Elizabeth<NP0> II<CRD> in<PRP> Buckingham<NP0> Palace<NP0> .</PUN> <sent>

<sent>Both<DT0> the<AT0> President<NN1> and<PRP> Taoiseach<NN1> , Mr<NP0> Ahern<NN1-NP0> sent<VVN> condolences<NN2> in<PRP> February<NP0> to<PRP> the<AT0> queen<NN1> mother<NN1> following<AJ0> death<NN1> of<PRF> her<DPS> daughter<NN1> , Princess<NP0> Margaret<NP0> .</PUN> <sent>

<sent>Queen<NN1> Mother<NN1> will<VM0> be<VBI> buried<VVN> at<PRP> Windsor<NP0> Castle<NP0> beside<PRP> her<DPS> husband<NN1> , King<NP0> George<NP0> VI<CRD> , who<PNQ> died<VVD> 50<CRD> years<NN2> ago<AV0> , ceremony<NN1> on<PRP> April<NP0> 9th<ORD> , Buckingham<NP0> Palace<NP0> announced<VVD> .</PUN> <sent>

<sent>The<AT0> queen<NN1> mother<NN1> , whose<DTQ> life<NN1> covered<NN1> a<AT0> century<NN1> of<PRF> tumultuous<AJ0> change<NN1> for<PRP> the<AT0> world<NN1> and<PRP> royal<AJ0> House<NN1> of<PRF> Windsor<NP0> , died<VVD> in<PRP> her<DPS> sleep<NN1> on<PRP> Saturday<NP0> with<PRP> her<DPS> daughter<NN1> , Queen<NP0> Elizabeth<NP0> , at<PRP> her<DPS> bedside<NN1> .</PUN> <sent>

<sent>Mrs<NP0> McAleese<NP0> said<VVD> : "<PUQ> She<PNP> had<VHD> many<DT0> admirers<NN2> in<PRP> Ireland<NP0> who<PNQ> will<VM0> mourn<VVI> her<DPS> passing<NN1> ."</PUN> <sent>
The queen mother will be sorely missed by her family and we offer our sincerest sympathies. 

"Mr Ahern, in a message of condolence, said: "The queen mother had a personal grace and charm which endeared her not only to the citizens of Britain but also to many people here in Ireland."

"Queen Elizabeth last night led members of the royal family in prayer and remembrance at the coffin of the queen mother.

"Senior royals filed solemnly into the Royal Chapel of All Saints in Windsor Great Park, London, for a brief private service. Prince Charles, said to be "absolutely devastated" by his grandmother's death, and his sons, Prince William and Prince Harry, cut short a skiing holiday in the Swiss resort of Klosters to fly home."

"In a break with protocol, the queen allowed Prince Charles, her heir, to travel with his sons on the same plane as something avoided by the royal family for security reasons."

"The queen mother kept the royal family together through seven tempestuous decades - from 1936 abdication crisis that propelled her shy husband onto the throne to Prince Charles' bitter divorce from Princess Diana in 1996."

"Mr Ahern, in a message of condolence, said: "The queen mother had a personal grace and charm which endeared her not only to the citizens of Britain but also to many people here in Ireland."

"Queen Elizabeth last night led members of the royal family in prayer and remembrance at the coffin of the queen mother.

"Senior royals filed solemnly into the Royal Chapel of All Saints in Windsor Great Park, London, for a brief private service. Prince Charles, said to be "absolutely devastated" by his grandmother's death, and his sons, Prince William and Prince Harry, cut short a skiing holiday in the Swiss resort of Klosters to fly home."

"In a break with protocol, the queen allowed Prince Charles, her heir, to travel on the same plane as something avoided by the royal family for security reasons."
Many schools in Britain will make special arrangements to enable pupils to watch the funeral on television, head-teachers said.

Tributes poured in yesterday describing the queen mother as a figure of courage who had won a place in the hearts of millions around the world.
Text 3
a children’s poem containing a high percentage of unknown words

Eletelephony
Once there was an elephant,
Who tried to use the telephant --
No! no! I mean an elephone
Who tried to use the telephone --
(Dear me! I am not certain quite
That even now I've got it right.)
Howe'er it was, he got his trunk
Entangled in the telephunk;
The more he tried to get it free,
The louder buzzed the telephee --
(I fear I'd better drop the song
Of elephop and telephong!).
Text 3(a)
The tagger’s analysis of the above text with errors marked – note this text has a large proportion of made-up words, but because these have regular suffixes, every one of them has been tagged correctly. Tokens, unknown to the tagger are underlined and actual errors made by the tagger are shown in bold print.

<sent>Eletelephony<NP0> Once<CJS> there<EX0> was<VBD> an<AT0> elephant<NN1> ,<PUN> Who<PNQ> tried<VVD> to<TO0> use<VVI> the<AT0> telephant<NN1> --<NN0> No<ITJ> !<PUN> <sent>

<sent>no<AV0> !<PUN> <sent>

<sent>I<PNP> mean<VVB> an<AT0> elephone<NN1> Who<PNQ> tried<VVD> to<TO0> use<VVI> the<AT0> telephone<NN1> --<NN0> (<PUL> Dear<AJ0> me<PNP> !<PUN> <sent>

<sent>I<PNP> am<VBB> not<XX0> certain<AJ0> quite<AV0> That<DT0> even<AV0> now<AV0> I<PNP> 've<VHB> got<VVN> it<PNP> right<AJ0> .<PUN> <sent>

<sent>)<PUR> Howe<NP0> 'er<NN1> it<PNP> was<VBD> ,<PUN> he<PNP> got<VVD> his<DPS> trunk<NN1> Entangled<VVD> in<PRP> the<AT0> telehunk<NN1> ;<PUN> The<AT0> more<AV0> he<PNP> tried<VVD> to<TO0> get<VVI> it<PNP> free<AJ0> ,<PUN> The<AT0> louder<AV0> buzzed<VVD> the<AT0> telephere<NN1> --<NN0> (<PUL> I<PNP> fear<VVB> I<PNP> 'd<VM0> better<AV0> drop<VVI> the<AT0> song<NN1> Of<PRF> elephop<NN1> and<CJC> telephong<NN1> !<PUN> <sent>

<sent>)<PUR> .<PUN> <sent>
Appendix 2 – Texts Used

Text 4
(informal writing)

It's 4.30 on a Wednesday afternoon, and Ben Bethell, Sam Ashton and Ntokozo Nkosi are cotching, which means they're sprawled across the sofas in Ben's front room, working their way solidly through a box of HobNobs, some Jaffa Cakes and a bowl of satsumas. They're also trying to explain the differences between skateboarders and townies, the two tribes that dominate their school, Stoke Newington, in north London.

'There's a skateboarding craze at the moment,' Ben says. 'Well, to some people it's a craze; to Sam it's a religion. A lot of the Year 7s get quite excited by it,' - he says this with the natural superiority of a Year 8 - 'they think, "Ooh, how exciting, rivalry between two groups."'

Sam says: 'There's this thing called a ruckus...'

'Which is tumbling about, jokey wrestling. '

'If it was serious, you'd call it a gang war.'

Sam, who is tall and effortlessly cool, is the only one who actually does any skateboarding, but is probably the most anxious to dissociate himself from the skateboarding crew. 'I do like baggy trousers [a crucial item of skateboarding uniform] and I have done since Year 6, when I was into hardcore bands like Limp Bizkit. But then everyone started liking hardcore, so now I've decided to go to more underground music, to make sure people won't find out about it. I listen mostly to modern punk rock, which is not as aggressive as the old punk rock.'

He names three of the obscure bands that he favours: No FX, Bad Religion and The Get Up Kids. 'So, yeah, there is this big thing between townies and skateboarders. But we're not really involved.' He sighs: 'Then people call you a no-styler.'
Appendix 2 – Texts Used

Text 4(a)
The tagger’s analysis of the above text with breakdown of suffixes considered for each unknown word (output from the tagger).

suffix was :tokozo
suffix was :okozo
suffix was :kozo
suffix was :ozo

suffix was :Nkosi
suffix was :kosi

suffix was :tching
suffix was :obNobs
suffix was :bNobs
suffix was :Nobs

It's 4.30 on a Wednesday, and Ben Bethell, Sam Ashton and Ntokozo Nkosi are cotching, which means they're sprawled across the sofas in Ben's front room, working their way solidly through a box of HobNobs, some Jaffa Cakes and a bowl of satsumas.

They're also trying to explain the differences between skateboarders and townies, the two tribes that dominate their school, Stoke Newington, in north London.

There's a skateboarding craze at the moment, Ben says this with the natural superiority...
of a Year 8 - ' they think, ' Ooh, how exciting, rivalry between two groups. 

Sam says: There's this thing called a ruckus. Which is tumbling, jokey wrestling. If it was serious, you'd call it a gang war. Sam, who is tall and effortlessly cool, is the only one who actually does any skateboarding, but is probably the most anxious to dissociate himself from the skateboarding crew. I do like baggy trousers, crucial item of skateboarding uniform, and I have done since Year 6, when I was into hardcore bands like Limp Bizkit. But then everyone started liking hardcore, so I decided to go to more underground music, to make sure people won't find out about it.

suffix was :Bizkit
suffix was :izkit
suffix was :zkit
suffix was :kit

I do like baggy trousers, crucial item of skateboarding uniform, and I have done since Year 6, when I was into hardcore bands like Limp Bizkit. But then everyone started liking hardcore, so I decided to go to more underground music, to make sure people won't find out about it.
I listen mostly to modern punk rock, which is not as aggressive as the old punk rock.

He names three of the obscure bands that he favours: NoFX, Bad Religion and The Get Up Kids.

So, yeah, there is this big thing between townies and skateboarders.

But we're not really involved.

suffix was: styler

He sighs: Then people call you a no-styler.

Time taken: less than 1 sec to tag 385 words of text
No. of unknown words encountered: 6
No. of sentences encountered (no. of lattices built): 18
Text 5
This text has an unusually high number of unknown words, but obviously features in the BNC. Many words, which were invented by the author, and which do not feature in any other text are contained. These were tagged as portmanteau tags in the BNC and have no other occurrences to aid in disambiguation. The tagger had to also resort to portmanteau tags in it’s tagging of this text.

Jabberwocky       Lewis Carroll (Charles Lutwidge Dodgson)

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"

He took his vorpal sword in hand:
Long time the manxome foe he sought--
So rested he by the Tumtum tree,
And stood awhile in thought.

And, as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!

One two! One two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.

"And hast thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!"
He chortled in his joy.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
Appendix 2 – Texts Used

Text 5(a)

(this is the tagging output for the above text. The tagger was unable to resolve the portmanteau tags (these are shown in bold)

"<sent>Jabberwocky<NP0> Lewis<NP0> Carroll<NP0> (<PUL> Charles<NP0> Lutwidge<NP0> Dodgson<NP0> )<PUR> 'Twas<NN2-VVB> brillig<NN1-VVB> ,<PUN> and<CJC> the<AT0> slithy<AJ0-NN1> toves<NN2-VVZ> Did<VD> gyre<NN1> and<CJC> gimble<NN1-VVB> in<PRP> the<AT0> wabe<NN1> ;<PUN> All<DT0> mimsy<NN1> were<VBD> the<AT0> borogoves<NN2> ,<PUN> And<CJC> the<AT0> mome<NN1> raths<NN2> outgrabe<NN1-VVB> .<PUN> <sent>

"<sent>Beware<VVB> the<AT0> Jabberwock<NN1-NP0> ,<PUN> my<DPS> son<NN1> !<PUN> <sent>

The<AT0> jaws<NN2> that<DT0> bite<NN1> ,<PUN> the<AT0> claws<NN2> that<CJT> catch<VVB> !<PUN> <sent>

Beware<VVB> the<AT0> Jubjub<NN1-NP0> bird<NN1> ,<PUN> and<CJC> shun<VVI> The<AT0> frumious<NN1-NP0> Bandersnatch<NN1-NP0> !<PUN> <sent>

"<sent>He<PNP> took<VVD> his<DPS> vorpal<NN1-NP0> sword<NN1> in<PRP> hand<NN1> :<PUN> Long<AJ0> time<NN1> the<AT0> manxome<NN1-NP0> foe<NN1-NP0> he<PNP> sought--<AV0> So<AV0> rested<VVD> he<PNP> by<PRP> the<AT0> Tumtum<NN1-NP0> tree<NN1> ,<PUN> And<CJC> stood<VVD> awhile<AV0> in<PRP> thought<NN1> .<PUN> <sent>

And<CJC> ,<PUN> as<CJS> in<PRP> uffish<NP0> thought<VVD> he<PNP> stood<VVD> ,<PUN> The<AT0> Jabberwock<NN1-NP0> ,<PUN> with<PRP> eyes<NN2> of<PRF> flame<NN1> ,<PUN> Came<VVD> whiffling<AJ0> through<PRP> the<AT0> tulgey<NN1-NP0> wood<NN1> ,<PUN> And<CJC> burbled<VVD> as<AV0> it<PNP> came<VVD> !<PUN> <sent>

One<CRD> two<CRD> !<PUN> <sent>

One<CRD> two<CRD> !<PUN> <sent>

And<CJC> through<PRP> and<CJC> through<PRP> The<AT0> vorpal<NP0> blade<NN1-NP0> went<VVD> snicker-snack<NP0> !<PUN> <sent>

He<PNP> left<VVD> it<PNP> dead<AJ0> ,<PUN> and<CJC> with<PRP> its<DPS> head<NN1> He<PNP> went<VVD> galumphing<NN1-NP0> back<AVP> .<PUN> <sent>

"<PUQ> And<CJC> hast<VVB> thou<PNP> slain<VVN> the<AT0> Jabberwock<NN1-NP0> ?<PUN> <sent>
Come to my arms, my beamish boy!

O frabjous day!

Callooh!

Callay!

"He chortled in his joy.

Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.
Appendix C.

Code

This contains the code necessary for implementing the tagger. The functions decided on for implementation in the end are included. All code written along the way can be seen in the directory ../users/Public/CSLL/4thYrProjects/Aisling. The databases used are stored in the directory /users/Public/CSLL/4thYrProjects/Software/PerlTagger/DB2.4.

Running the various modules is done by entering the following commands:

tagger: gcc -g -static timothy.c opendir.c lookup.c segment.c viterbi.c lattice.c disamb.c -L/usr/local/depot/db-2.4.14/lib -ldb
view databases: gcc -g -static dbtest.c opendir.c lookup.c segment.c viterbi.c lattice.c disamb.c -L/usr/local/depot/db-2.4.14/lib -ldb
scoring module: gcc -g -static score.c opendir.c lookup.c segment.c viterbi.c scorelattice.c disamb.c -L/usr/local/depot/db-2.4.14/lib -ldb

opendir.c – includes functions to open the various databases.

#include "/usr/local/depot/db-2.4.14/include/db.h"
#include <string.h>

/**********************************************************/
/*  this is a wrapper function for the C API function  */
/*  to open a database. Using this means not having    */
/*  to enter all the invariable information into the    */
/*  function call brackets each time - these are       */
/*  all encapsulated into the function                */
/***********************************************************/
DB * mydb_open(char *file){
    DB *dbp;
    db_open(file, DB_HASH, DB_RDONLY, 0444, NULL, NULL, &dbp);
    return dbp;
}
/***********************************************************/
/* this is a similar wrapping function used to extract */
/* information out of the database in question. It */
/* takes in a key and returns the corresponding data. */
/**********************************************************/

int myDB_get(DB *dbp, char *search, char *found, int size) {
    int j;
    DBT key = {search, strlen(search)};
    DBT data;

    j = dbp->get(dbp, NULL, &key, &data, 0);

    if (j == DB_NOTFOUND) {
        return 0;
    }
    else if (data.size < size) {
        strncpy(found,(char *)(data.data),data.size);
        found[data.size] = '\0';
        return 1;
    }
    else {
        /* error regarding size of destination */
        return -1;
    }
}

segment.c – includes functions to divide up input

#include <stdio.h>
#include <string.h>
#include "segment.h"
#include "datastructures.h"

#define MAXLENGTH 500
#define MAXWORD 42
#define MAXWORDS 100

char **tokens_ptr;
int numtokens;
char newinput[MAXLENGTH];

/**********************************************************/
/* a function to split up the sentence.Places a '.' */
/* at the start and spaces the sentence properly */
/* in order to correctly tokenise the words - */
/* handle digits containing '.', hyphens in a word, */
/* word splitting if an '"' occurs in the word etc. */
/* pretokenise(char *input){
   int i=0;
   int j=2;
   char c;
   int n=0;
   int stop = 0;

   /*take out any white space occuring at the start of the sentence*/
   while(isspace(input[i])){
      i++;
   }

   /*Place a '.' at the beginning of each sentence so that the first*/
   /*word of the sentence can be disambiguated properly.*/
   /*In doing this it can get the probability of the first word of the new sentence*/
   /*occurring directly after the end of a sentence*/
   newinput[0] = '.';
   newinput[1] = ' ';

   /*this stops when an end of sentence marker is found (but not for the initial '.')+*/
   while(((c = input[i]) != NULL)&&(newinput[j-2]!='.')&&(newinput[j-2]!=='!')&&(newinput[j-2]!=='?')) || (stop==0)){
      stop = 1;
      if(isalpha(c)){
         newinput[j] = c;
         i++;
         j++;
      }else if(isdigit(c)){
         newinput[j] = c;
         i++;
         j++;
      }else if(isspace(c)){
         newinput[j] = c;
         i++;
         j++;
      }
   }

   while(isdigit(input[i])||(input[i]=='.')||(input[i]==';')&&(isalpha(input[i+1]))||(input[i]=='t'&&input[i+1]=='h')||(input[i]=='s'&&input[i+1]=='t')||(input[i]=='n'&&input[i+1]=='d')){
      newinput[j] = input[i];
      i++;
      j++;
   }

   else if(isisspace(c)){
      newinput[j] = c;
      i++;
      j++;
   }
   /*to ensure there is only one white space between each word*/
   while(isspace(input[i])){
      i++;
   }
   /*if character is a hyphen -then dont separate the words*/
   if(c =='-'){
      newinput[j] = c;
      i++;
   }
}*/
j++;  /*this if ties the ' to the letters that come after it and makes them into one token*/
} else if((isalnum(input[i-1]))&&(c == '\')&&(isalnum(input[i+1]))){
    newinput[j] = ' '; 
    j++; 
    newinput[j] = c; 
    i++;  
    j++; 
    while((isalnum(input[i]))){
        newinput[j] = input[i]; 
        i++;  
        j++; 
    }
}
else{  /*to separate out punctuation for the tokeniser*/
    newinput[j] = ' '; 
    j++; 
    newinput[j] = c; 
    j++; 
    newinput[j] = ' '; 
    i++;  
    j++; 
}
}
newinput[j++] = '\0';

numtokens = 0;
while(newinput[n] != '\0'){  /*to handle the first word in the sentence*/
    if((numtokens == 0)&&(isgraph(newinput[n]))){
        numtokens++; 
        n++; 
        /*if a new word is encountered*/
    } else if((numtokens > 0)&&(newinput[n] == ' ')&&(isgraph(newinput[n+1]))){
        numtokens++; 
        n++; 
        /*if in the middle of a word*/
    } else{
        n++; 
    }
}
return 1;
}

/***********************************************************/
/*    a function to actually make the array of tokens -    */
/*    includes dynamic allocation for the array of strings*/
/***********************************************************/
int tokenise(char *input){

}
Appendix 3 – Code

```c
char output[MAXWORDS][MAXWORD];
char *token;
int x;
int i=0;
int j=0;

pretokenise(input);

token = strtok(newinput, " ");
strcpy(output[i], token);
i++;

while(i != numtokens){
    token = strtok(NULL, " ");
    strcpy(output[i], token);
i++;
}

/*dynamic allocation of memory for storing the array of tokens*/
if ((tokens_ptr = (char **)calloc(numtokens, sizeof(token))) == NULL) {
    printf("ERROR: Calloc failed");
    exit();
}

/*split the tokens and store them in the tokens_ptr array*/
for(x=0; x<numtokens ; x++){
    tokens_ptr[x] = strdup(output[x]);
}

return 1;
}

lookup.c – includes functions to lookup and sort.

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "lookup.h"
#include <malloc.h>
#define FOUNDSIZE 500

/*these two arrays hold all the probabilities for hapax legomena and capitalisation - precalculated*/
/*there are 768 characters and 32 tags in this string*/
```
Appendix 3 – Code

```c
/*there are 709 characters and 30 tags in this string*/
char oneoffcapsprobs[] = "c.NP0[500].NN1-NP0[18.2863990413421].AJ0[8.50482052399368].NN1[7.16052072552971].UNC[6.56462770303393].NN2[6.21929298981426].AJ0-NN1[3.94683806307533].NN0[3.7387657298083].VVB[1.4510542589466].VVG[1.14167438313634].NN1-VVB[1.05779181872651].AV0[0.77998910616047].ZZ0[0.72335094504058].NN2-VVZ[0.480418323438096].VVD[0.47823955531347].VVI[0.42377035766229].VVI[0.23857508578986].CRD[0.234217549975489].VN[0.223237310441745].AJ0-VVG[0.192820959742763].IJT[0.133994226265047].AJC[0.11983223487118].VVD-VN[0.040307206274816].AJ0-VV0[0.020698295114113]");

DB *dbprob_tag2_given_tag1;
DB *dbprob_tags_given_word;
DB *dbaffix_probs;
char found[FOUNDSIZE];
int numunknowns;
extern double tgtp;

/***********************************************************/
/*this function initialises all the databases used */
/***********************************************************/
int initialise_dbs() {

dbprob_tag2_given_tag1 = mydb_open("/users/Public/CSLL/4thYrProjects/Software/PerlTagger/DB2.4/cum_catcat_2_prob2000");
dbprob_tags_given_word = mydb_open("/users/Public/CSLL/4thYrProjects/Software/PerlTagger/DB2.4/cum_form_2_catprobs2000");
dbaffix_probs = mydb_open("/users/Public/CSLL/4thYrProjects/Software/PerlTagger/DB2.4/affixtype_to_catprob speedy");
}

/***********************************************************/
```

99
/* this function takes in the 2 tags of the bigram, */
/* concatenates them into one string and performs */
/* the tag2_given_tag1 lookup function and converts */
/* the returned string to its corresponding double */
/* value */
/**********************************************************/

double prob_tag2_given_tag1(char *tag2, char *tag1){

    char tolookup[20];
    strcpy(tolookup,tag1);
    strcat(tolookup," ");
    strcat(tolookup,tag2);

    if(myDB_get(dbprob_tag2_given_tag1, tolookup, found, FOUNDSIZE)) {
        tgtp = atof(found);
        return tgtp;
    }else {
        return -1;
    }
}

/**********************************************************/
/* this function takes a word as input.this word then */
/* serves as the key for the hash table lookup.it then */
/* returns a string of all possible tags and their */
/* probabilities */
/**********************************************************/

/*a method to return a string of all possible tags and their probabilities for*/
/*a given word or token*/
char * prob_tags_given_word(char *word){
    char *alltags;
    int i=0;
    char uncword[50];
    found[0] = NULL;

    /*to handle digits*/
    if(isdigit(word[0]) != 0){
        i++;
        while((isdigit(word[i]) != 0)||((word[i]=='.')||(word[i]==t')&(word[i+1]==h')))||(word[i]==s')&(word[i+1]==t'))||(word[i]==n')||(word[i]==n')&(word[i+1]==d')){
            i++;
        }
    }
    if((word[i+1] == \'0\')&(word[i-1]==t')&(word[i]==h'))||(word[i-1]==s')&(word[i]==t')||(word[i-1]==n')||(word[i-1]==n')&(word[i]==d')}
    alltags=".ORD[100]";
    else
    alltags=".CRD[100]";
/*to handle brackets*/
} else if(strcmp(word, "(" ) == 0)
    alltags=".PUL[100]";
else if(strcmp(word, ")") == 0)
    alltags=".PUR[100]";
/*all unrecognised punctuation*/
else if(strcmp(word, ".") == 0)
    alltags=".PUN[100]";
else if(strcmp(word, ",") == 0)
    alltags=".PUN[100]";
else if(strcmp(word, ";") == 0)
    alltags=".PUN[100]";
else if(strcmp(word, ":") == 0)
    alltags=".PUN[100]";
else if(strcmp(word, "-") == 0)
    alltags=".PUN[100]";
else if(strcmp(word, "'") == 0)
    alltags=".PUQ[100]";
else if(strcmp(word, "\"") == 0)
    alltags=".PUQ[100]";
else if(strcmp(word, "*") == 0)
    alltags=".UNC[100]";
else if(strcmp(word, "/") == 0)
    alltags=".UNC[100]";
else if(strcmp(word, "£") == 0)
    alltags=".UNC[100]";
else if(strcmp(word, "$") == 0)
    alltags=".UNC[100]";
/*this goes to search in the DB for the tags for that word*/
else if(myDB_get(dbprob_tags_given_word, word, found, FOUNDSIZE))
    alltags=found;
else if((check_if_cap(word)==1)&&(strcpy(uncword,word))&&(tolower(uncword[0]))&&(myDB_get(dbprob_tags_given_word,uncword, found, FOUNDSIZE)))
    alltags=found;
/* the next two if statements handle all unknown words-they set alltags to be */
/* the string of tags for hapax legomena with probabilities for capitalisation */
/* considered. The string alltags is headed with a marker letter to show that */
/* the array still needs affix modification. This is done in all_tags_to_array */
else if(check_if_cap(word) == 1){
    alltags = oneoffcapsprobs;
    numunknowns++;
} else if(check_if_cap(word) == 0){
    alltags = oneoffnocapsprobs;
    numunknowns++;
} else{
    alltags = "UNC[100]";
    numunknowns++;
}
return alltags;

/**********************************************************/
/* a function to perform a database lookup when */
/* passed in an affix. A list of all relevant tags */
/* and their probabilities */
/**********************************************************/
char *prob_tags_given_affix(char *affix){
    if(myDB_get(dbaffix_probs, affix, found, FOUNDSIZE) == 1){
        return found;
    } else{
        return NULL;
    }
}

/**********************************************************/
/* a function to count the number of tags in a */
/* string extraced from a database */
/**********************************************************/
int count_no_of_tags(char *alltags){
    int nooftags = 0;
    int i=0;

    for(i=0; (alltags[i] != '\0'); i++){
        if(alltags[i] == '\['){
            nooftags++;
        } else{
            nooftags++;
        }
    }
    return nooftags;
}
int all_tags_to_array(char *word, char *alltags, int nooftags) {
    void *ptr;
    char thetag[10];
    char stringprob[30];
    char *tempword;
    char *temptag;
    char *affix_str;
    double *tempprob;
    double theprob;
    int j, k;
    int t = 0;
    int i = 0;
    char c;
    int is_unknown = 1;
    int no_affixtags;

    if ((posinfo_ptr = (Posinfo *)calloc(nooftags, sizeof(Posinfo))) == NULL) {
        fprintf(stderr, "ERROR: Calloc failed");
        exit(1);
    }
    if (isalpha(alltags[i])) {
        is_unknown = 0;
        i++;
    }
    while (alltags[i] != '\0') {

        if (alltags[i] == '.') {
            i++;
        }

        if (isupper(alltags[i])) {
            /* to handle the tag */
            for (j = 0; ((c = alltags[i]) != '\['); j++, i++) {
                thetag[j] = c;
            }
            thetag[j] = '\0';

            theprob = atof(stringprob);
            tempprob = (double *) calloc(1, sizeof(double));
            tempprob[0] = theprob;
            posinfo_ptr[j].prob = tempprob[0];
            posinfo_ptr[j].tag = thetag;
            posinfo_ptr[j].is_upper = 1;
        }

        if (islower(alltags[i])) {
            /* to handle the tag */
            for (j = 0; ((c = alltags[i]) != ' '); j++, i++) {
                thetag[j] = c;
            }
            thetag[j] = '\0';

            theprob = atof(stringprob);
            tempprob = (double *) calloc(1, sizeof(double));
            tempprob[0] = theprob;
            posinfo_ptr[j].prob = tempprob[0];
            posinfo_ptr[j].tag = thetag;
            posinfo_ptr[j].is_upper = 0;
        }
    }
}

if((alltags[i] == '[') && (isdigit(alltags[i+1]))){
    i++;
    /* to handle the digits */
    for(j=0; (c=alltags[i]) != ']'; i++, j++){
        stringprob[j] = c;
    }
    stringprob[j] = '\0';
    /* to convert the string of digits to a double */
    theprob = atof(stringprob);
}
temptag = strdup(thetag);
/* add to the array */
posinfo_ptr[t].word = word;
posinfo_ptr[t].tag = temptag;
posinfo_ptr[t].prob = theprob;
posinfo_ptr[t].previous = NULL;
t++;
i++;
}
/* if the alltags sequence was marked with a letter at the beginning to */
/* show that it was an unknown word - find its largest affix, look this */
/* up in the database, count the affix tags returned, place them in an */
/* array and apply each affix to its corresponding tag in the original array */
if(is_unknown == 0){
    affix_str = most_prob_affix(word);
    no_affixtags = count_no_of_tags(affix_str);
    all_affix_tags_to_array(word, affix_str, no_affixtags);
    apply_affixes(t, no_affixtags);
    free_affix_ptr(no_affixtags);
    if(posinfo_ptr[t-1].tag != NULL)
        return 1;
    else
        return -1;
}

/***********************************************************/
/* this function sets up a dynamic array, allocates */
/* memory to store all the affix structs and their */
/* corresponding probabilities for the word in */
/* question. This dynamic array of structs is called */
/* affix_ptr. */
/***********************************************************/
int all_affix_tags_to_array(char *word, char *affix_str, int nooftags) {
    char thetag[10];
    char stringprob[22];
    char *tempword;
    char *temptag;
    double *tempprob;
    double theprob;
    int j;
    int t = 0;
    int i = 0;
    char c;

    if ((affix_ptr = (Posinfo *)calloc(nooftags, sizeof(Posinfo))) == NULL) {
        fprintf(stderr, "ERROR: Calloc failed");
        exit(1);
    }

    while (affix_str[i] != '\0') {
        if ((affix_str[i] == '%' && (affix_str[i + 1] == '%')))
            i++; i++;
        else if (isupper(affix_str[i])) {
            /* to handle the tag */
            for (j = 0; ((c = affix_str[i]) != '['); j++, i++)
                thetag[j] = c;
            thetag[j] = '\0';

        }

        if ((affix_str[i] == '[') && (isdigit(affix_str[i + 1]))) {
            /* to handle the digits*/
            for (j = 0; (c = affix_str[i]) != ']'; i++, j++)
                stringprob[j] = c;
            stringprob[j] = '\0';

        } /* to convert the string of digits to a double */
        theprob = atof(stringprob);

        /* tempword = strdup(word); */
        tempword = strdup(word);
        temptag = strdup(thetag);

        /* add to the array */
        affix_ptr[t].word = tempword;
        affix_ptr[t].affix = temptag;
        affix_ptr[t].affixprob = theprob;
        t++;
    }
}
affix_ptr[t].tag = temptag;
affix_ptr[t].prob = theprob;
affix_ptr[t].previous = NULL;
t++; 
i++;
}
if(affix_ptr[t-1].tag != NULL) {
    return 1;
}
}

/**********************************************************/
/*     this function takes the unknown word in question,  */
/*     and extracts it's largest suffix (max length 6)    */
/*     if it can find this suffix in the suffix DB it      */
/*     returns it's relevant tags and probabilities.      */
/*     If it cant find this, it takes the next smallest   */
/*     affix and tries that instead. It does this until   */
/*     it either finds one, or is at the last letter.     */
/**********************************************************/

char *most_prob_affix(char *word) {
    int length, start, i, j, k;
    char suffix[7];
    char *suffix_probs;
    length = strlen(word);
    suffix_probs = NULL;
    for (i = 6; ((i >= 1) && (suffix_probs == NULL)); i--) {
        if (length >= i) {
            start = length - i;
            k = 0;
            for (i = start; (word[j] != NULL) && (k < 7); j++) {
                suffix[k] = word[j];
                k++;
            }
            suffix[k++] = '\0';
            suffix_probs = prob_tags_given_affix(suffix);
            if (suffix_probs != NULL)
                return suffix_probs;
        }
    }
    if (suffix_probs != NULL)
        return suffix_probs;
    }

/**************************************************************/
/* this function applies all of the affixes currently */
/* stored in the affix array to the array of the */
/* corresponding tags for an unknown word */
/**********************************************************/

int apply_affixes(int no_of_tags,int no_affixtags){
    int i,j;
    for(i=0;i<no_of_tags ;i++){
        for(j=0;j<no_affixtags;j++){
            if(strcmp(posinfo_ptr[i].tag,affix_ptr[j].tag)==0)
                posinfo_ptr[i].prob = (posinfo_ptr[i].prob * affix_ptr[j].prob * 100);
        }
    }
    return 1;
}

/**********************************************************/
/* function to free up the affix_ptr array */
/* this has to be done every time the array is made and applied to a word that is unknown. */
/**********************************************************/

void free_affix_ptr(int no_affixtags){
    int k;
    for(k=(no_affixtags-1);k>=0; k--){
        free(affix_ptr[k].tag);
        affix_ptr[k].tag = NULL;
    }
    free(affix_ptr);
    affix_ptr = NULL;
}

lattice.c – includes functions to creat, store and free up the lattice.

#include <stdio.h>
#include <malloc.h>
#include "lookup.h"

extern char **tokens_ptr;
extern int numtokens;
extern Posinfo *posinfo_ptr;
int nooftags;
Lattice thelattice;
Appendix 3 – Code

/***********************************************************/
/* a function to make the lattice itself - it allocates */
/* an array big enough to hold the arrays for all the    */
/* words. Then it finds out how many tags each of these  */
/* words has and allocates accordingly. This dynamic    */
/* lattice then holds all the initial probabilities.    */
/* The lattice itself is fundamentally a dynamic array  */
/* of dynamic arrays of structures.                     */
/***********************************************************/
int make_lattice(){

    Column prescolumn;
    char *dbresult;
    int i;
    int j=0;

    thelattice.lat = (Column *)calloc(numtokens,sizeof(Column));
    thelattice.width = numtokens;

    for(i=0; i< numtokens ; i++){
        prescolumn.word = tokens_ptr[i];
        prescolumn.height= count_no_of_tags(dbresult);
        nooftags = prescolumn.height;

        all_tags_to_array(tokens_ptr[i],dbresult,prescolumn.height);

        prescolumn.col = &posinfo_ptr[0];

        thelattice.lat[j] = prescolumn;
        divide_portmanteau(j);
        j++;
    }

    /*assign the array of posinfo pointers to be the present column*/
    prescolumn.col = &posinfo_ptr[0];

    /*put this new column structure into the lattice*/
    thelattice.lat[j] = prescolumn;
    divide_portmanteau(j);
    j++;

} else{
    thelattice.lat[j].col[0].word = tokens_ptr[i];
    thelattice.lat[j].col[0].tag = "UNC";
    thelattice.lat[j].col[0].prob = 0;
    thelattice.lat[j].col[0].previous = NULL;
}

/***********************************************************/
/* this function prints the lattice in the form of */

108
Appendix 3 – Code

```c
/* a table so that all possibilities and their numbers are easy to see. */
/*****************************************************************************/
int print_lattice(){
    int k,i,j;
    for(k=0 ;k < 70 ;k++){
        printf("-");
    }

    /* print("%s"
        printf("%s\n","word","tags & probs"); */
    printf("\n word\t
tags & probabilities\n");
    printf("\n ------		---------------------
");
    for(i=0; i < thelattice.width; i++){  
        /*print the word itself*/
        printf(" %s", thelattice.lat[i].col[0].word);
        if(strlen(thelattice.lat[i].col[0].word)<=7){
            printf("\t");
        }else{
            printf("\n\t");
        }
        printf("%10s ", thelattice.lat[i].col[0].tag);
        printf(" (%e) ", thelattice.lat[i].col[0].prob);
        /*recurse through the tags*/
        for(j=1; j < thelattice.lat[i].height; j++){  
            printf("%10s ", thelattice.lat[i].col[j].tag);
            printf(" (%e) ", thelattice.lat[i].col[j].prob);
            if(j % 3== 0){
                printf("\n\t");
            }
            j++;
        }
        printf("\n\n");
    }
    for(k=0; k<70; k++){
        printf("-");
    }
    printf("\n\n");
}

/*****************************************************************************/
/* a function to free up the lattice so that it can be used again by the next sentence. */
/* This also frees up all the other arrays of pointers */
/* This allows memory reuse and prevents the program */
/* from running out of memory when tagging a large file  */
/**********************************************************************/
void free_lattice(){

    int c,t,p;

    /*free up all the words - stored in tokens_ptr*/
    for(p=(numtokens-1); p>=0; p--){
        free(tokens_ptr[p]);
        tokens_ptr[p] = NULL;
    }
    free(tokens_ptr);
    tokens_ptr = NULL;

    /*free up the array that holds the pos structs*/
    /*this current one only holds the pos for the last*/
    /*column in the array*/
    for(p=(nooftags-1);p>=0; p--){
        free(posinfo_ptr[p].tag);
        posinfo_ptr[p].tag = NULL;
    }
    free(posinfo_ptr);
    posinfo_ptr = NULL;

    free(thelattice.lat[numtokens-1].col);
    thelattice.lat[numtokens-1].col = NULL;

    /*free up all the tags stored in each col of the array*/
    /*except for the last one that has already been done*/
    /*in the previous free statement*/
    for(c=(numtokens-2); c>=0 ; c--){
        for(t=(thelattice.lat[c].height-1) ; t>=0 ; t--){
            free(thelattice.lat[c].col[t].tag);
            thelattice.lat[c].col[t].tag = NULL;
        }
        free(thelattice.lat[c].col);
        thelattice.lat[c].col = NULL;
    }
    free(thelattice.lat);
    thelattice.lat = NULL;
}

viterbi.c - includes functions to perform the calculations.

#include <stdio.h>
#include "datastructures.h"

extern Lattice thelattice;
extern int numtokens;
Tagging bptagseq;
double tgtp;

/********************************************
/*       a function to sequentially call the function  */
/* bestpath_a_column which performs the viterbi process  */
/********************************************

int do_viterbi(){
  int i;
  for(i=1; i < thelattice.width; i++){
    bestpath_a_column(i);
  }
}

/********************************************
/*       a function to execute the viterbi algorithm  */
/*  on each tag in the present column of the array. It  */
/*  sets the backpointer of each POS to be the most   */
/*  likely previous tag in the path                */
/********************************************
/*takes as input the column being investigated*/
int bestpath_a_column(int i){
  int j,k;
  char *temptgtp;
  double tgwp, prevprob, bpprob, origprob = 0;
/*this for is for dealing with each individual posinfo in the current column*/
  for(j=0; j < thelattice.lat[i].height; j++){
    tgwp, tgtp, prevprob, bpprob, origprob = 0;
/*this was added because the value of 'thelattice.lat[i].col[j].prob' changes throughout*/
/*and the original tag_given_word_prob was needed for calculation*/
    origprob = thelattice.lat[i].col[j].prob;
/*this for is for dealing with the probs of each of the posinfos in the previous column*/
    for(k=0; k < thelattice.lat[i-1].height; k++){
      tgwp = tgtp = prevprob = bpprob = 0;
/*this was added because the value of 'thelattice.lat[i].col[j].prob' changes throughout*/
/*and the original tag_given_word_prob was needed for calculation*/
      origprob = thelattice.lat[i].col[j].prob;
/*this for is for dealing with the pros of each of the posinfos in the previous column*/
      for(k=0; k < thelattice.lat[i-1].height; k++){
        tgwp = tgtp = prevprob = bpprob = 0;
        tgwp = origprob;
        prob_tag2_given_tag1(thelattice.lat[i].col[j].tag, thelattice.lat[i-1].col[k].tag);
        prevprob = thelattice.lat[i-1].col[k].prob;
        bpprob = tgwp * tgtp * prevprob;
/*if it is the first posinfo checked in the previous column - set it to be the best previous*/
        if(k==0){
          thelattice.lat[i].col[j].prob = bpprob;
          thelattice.lat[i].col[j].previous = (thelattice.lat[i-1].col)+k;
        }
/*but if a better one arises - set this new one to be the best*/
        if(bpprob > thelattice.lat[i].col[j].prob){

111
Appendix 3 – Code

    thelattice.lat[i].col[j].prob = bpprob;
    thelattice.lat[i].col[j].previous = (thelattice.lat[i-1].col)+k;
    }
    }
    }
    }

int find_tagging(){
    Tag mptag;
    Posinfo prevpos;
    char *bpword;
    char *bptag;
    double bpprob;
    int i;
    if((bptagseq.tags = (Tag *)calloc(thelattice.width ,sizeof(Tag)))==NULL){
        fprintf(stderr, "ERROR: Calloc failed in allocating for bptagseq ");
        exit(1);
    }
    bptagseq.length = thelattice.width;
    bpword = thelattice.lat[thelattice.width-1].col[0].word;
    bptag = thelattice.lat[thelattice.width-1].col[0].tag;
    bpprob = thelattice.lat[thelattice.width-1].col[0].prob;
    prevpos = *(thelattice.lat[thelattice.width-1].col[0].previous);
    /*to handle the last column of the array and determine the most probable last tag*/
    for(i=1; i < thelattice.lat[thelattice.width-1].height;i++){
        if(thelattice.lat[thelattice.width-1].col[i].prob > bpprob){
            bpword = thelattice.lat[thelattice.width-1].col[i].word;
            bptag = thelattice.lat[thelattice.width-1].col[i].tag;
            bpprob = thelattice.lat[thelattice.width-1].col[i].prob;
            prevpos = *(thelattice.lat[thelattice.width-1].col[i].previous);
        }
    }
    /* tempword = strdup(bpword); */
    /* temptag = strdup(bptag); */
    bptagseq.tags[thelattice.width-1].word = bpword;
    bptagseq.tags[thelattice.width-1].tag = bptag;
    bptagseq.tags[thelattice.width-1].prob = bpprob;
    for(i=(thelattice.width - 2); i>=0; i--){
        bpword = prevpos.word;
        bptag = prevpos.tag;
        bpprob = prevpos.prob;
        if((i!=0)&&(prevpos.tag)!=NULL){
            prevpos = *(prevpos.previous);
        }
    }
    /* tempword = strdup(bpword); */

112
/* temptag = strdup(bptag); */

bptagseq.tags[i].word = bpword;
bptagseq.tags[i].tag = bptag;
bptagseq.tags[i].prob = bpprob;
}
}

/***********************************************************/
/* function to print_do_viterbi */
/***********************************************************/
int print_do_viterbi(){
    int i,j,k;
    double tgwp, tgtp, prevprob, bpprob, origprob;

    printf("\n");
    /*in the very first column, the probabilities do not change*/
    printf("the tags for '%s' are\n", thelattice.lat[0].col[0].word);
    for(i=0; i< thelattice.lat[0].height; i++){
        printf("tag:%s		prob:%e\n", thelattice.lat[0].col[i].tag, thelattice.lat[0].col[i].prob);
    }
    printf("These will remain unchanged\n\n");

    for(i=1; i < thelattice.width; i++){
        printf("the original tag(s) for '%s' are\n", thelattice.lat[i].col[0].word);
        for(j=0; j< thelattice.lat[i].height; j++){
            printf("tag:%s		prob:%e\n", thelattice.lat[i].col[j].tag, thelattice.lat[i].col[j].prob);
        }
    }
    printf("\n");
    /*this for is for dealing with each individual posinfo in the current column*/
    for(j=0; j < thelattice.lat[i].height; j++){
        origprob = thelattice.lat[i].col[j].prob;

        /*this was added because the value of 'thelattice.lat[i].col[j].prob' changes throughout*/
        /*and the original tag_given_word_prob was needed for calculation*/
        origprob = thelattice.lat[i].col[j].prob;

        /*this for is for dealing with the probs of each of the posinfos in the previous column*/
        for(k=0; k < thelattice.lat[i-1].height; k++){
            printf("Calculating path to tag %s from tag %s \n", thelattice.lat[i].col[j].tag, thelattice.lat[i-1].col[k].tag);
            tgwp = origprob;
            printf("Probability of %s being the tag for the word '%s': \n", thelattice.lat[i].col[j].word, origprob);
            tgtp = prob_tag2_given_tag1(thelattice.lat[i-1].col[k].tag, thelattice.lat[i].col[j].tag);
            printf("Probability of %s (the previous word) :\n", thelattice.lat[i-1].col[k].word, prevprob);
            bpprob = tgwp * tgtp * prevprob;
        }
    }
}

int main(){
    return 0;
}
printf("Overall probability of %s having tag %s :	%e\n", thelattice.lat[i].col[j].word, thelattice.lat[i].col[j].tag, bpprob);

/*if it is the first posinfo checked in the previous column - set it to be the best previous*/
if(k==0){
    thelattice.lat[i].col[j].prob = bpprob;
    thelattice.lat[i].col[j].previous = (thelattice.lat[i-1].col)+k;
    printf(" *%s is the most probable previous tag for %s\n", thelattice.lat[i-1].col[k].tag, thelattice.lat[i].col[j].word);
    printf(" when %s has %s as it's tag.\n\n", thelattice.lat[i].col[j].word, thelattice.lat[i].col[j].tag);
}

/*but if a better one arises - set this new one to be the best*/
if(bpprob > thelattice.lat[i].col[j].prob){
    thelattice.lat[i].col[j].prob = bpprob;
    thelattice.lat[i].col[j].previous = (thelattice.lat[i-1].col)+k;
    printf(" * This is the new most probable previous tag\n\n");
} else{
    if(k!=0){
        printf("The most probable previous tag not changed\n\n");
    }
}
}

void free_bp(){
    int p;

    free(bptagseq.tags);
    bptagseq.tags = NULL;
}

disamb.c – functions to provide further disambiguation

#include <stdio.h>
#include "datastructures.h"

extern char **tokens_ptr;
extern int numtokens;
extern Lattice thelattice;

/* a function to check to see if the unknown word is capitalised or not*/
/*returns 1 if the word is capitalised and 0 if it is not*/
int check_if_cap(char *word){
    if(isupper(word[0])!=0){
        return 1;
    }
}
/* a function which clarifies to the user the type of tag returned */
char * explain_tag(char *tag){
    char *result;
    
    if(strcmp(tag,"AJ0")==0)
        result = "Adjective (general or positive) ";
    else if(strcmp(tag,"AJC")==0)
        result = "Comparative Adjective";
    else if(strcmp(tag,"AJS")==0)
        result = "Superlative Adjective";
    else if(strcmp(tag,"AT0")==0)
        result = "Article ";
    else if(strcmp(tag,"AV0")==0)
        result = "General adverb ";
    else if(strcmp(tag,"AVP")==0)
        result = "Adverb particle ";
    else if(strcmp(tag,"AVQ")==0)
        result = "Wh-adverb ";
    else if(strcmp(tag,"CJC")==0)
        result = "Coordinating conjunction";
    else if(strcmp(tag,"CJS")==0)
        result = "Subordinating conjunction ";
    else if(strcmp(tag,"CJT")==0)
        result = "Subordinating conjunction 'that' ";
    else if(strcmp(tag,"CRD")==0)
        result = "Cardinal number";
    else if(strcmp(tag,"DPS")==0)
        result = "Possessive determiner ";
    else if(strcmp(tag,"DT0")==0)
        result = "General determiner";
    else if(strcmp(tag,"DTQ")==0)
        result = "Superlative Adjective ";
    else if(strcmp(tag,"EX0")==0)
result = "Existential 'there';"
else if(strcmp(tag,"ITJ")==0)
    result = "Interjection ";
else if(strcmp(tag,"NN0")==0)
    result = "Common noun -neutral for number ";
else if(strcmp(tag,"NN1")==0)
    result = "Singular common noun ";
else if(strcmp(tag,"NN2")==0)
    result = "Plural common noun ";
else if(strcmp(tag,"NP0")==0)
    result = "Proper noun ";
else if(strcmp(tag,"ORD")==0)
    result = "Ordinal numeral ";
else if(strcmp(tag,"PNI")==0)
    result = "Indefinite pronoun ";
else if(strcmp(tag,"PNP")==0)
    result = "Personal pronoun ";
else if(strcmp(tag,"PNQ")==0)
    result = "Wh-pronoun ";
else if(strcmp(tag,"PNX")==0)
    result = "Reflexive pronoun";
else if(strcmp(tag,"POS")==0)
    result = "Possessive marker ";
else if(strcmp(tag,"PRF")==0)
    result = "Preposition 'of'";
else if(strcmp(tag,"PRP")==0)
    result = "Preposition";
else if(strcmp(tag,"PUL")==0)
    result = "Punctuation: left bracket";
else if(strcmp(tag,"PUN")==0)
    result = "Punctuation: separator";
else if(strcmp(tag,"PUQ")==0)
    result = "Punctuation: quotation mark ";
else if(strcmp(tag,"PUR ")==0)
    result = "Punctuation: right bracket ";
else if(strcmp(tag,"TO0")==0)  
    result = "Infinitive marker 'to'";

else if(strcmp(tag,"UNC")==0)  
    result = "Unclassified ";

else if(strcmp(tag,"VBB")==0)  
    result = "Verb 'be'-pres tense form";

else if(strcmp(tag,"VBD")==0)  
    result = "Verb 'be'-past tense form";

else if(strcmp(tag,"VBG")==0)  
    result = "Verb 'be' -ing form";

else if(strcmp(tag,"VBI ")==0)  
    result = "Verb 'be' -infinitive form";

else if(strcmp(tag,"VBN")==0)  
    result = "Verb 'be'-past participle";

else if(strcmp(tag,"VBZ")==0)  
    result = "Verb 'be'- 's' form";

else if(strcmp(tag,"VDB")==0)  
    result = "Verb 'do'- finite form";

else if(strcmp(tag,"VDD")==0)  
    result = "Verb 'do'- past tense form";

else if(strcmp(tag,"VDG")==0)  
    result = "Verb 'do'- ing form";

else if(strcmp(tag,"VDI")==0)  
    result = "Verb 'do' -infinitive form";

else if(strcmp(tag,"VDN")==0)  
    result = "Verb 'do'-past participle";

else if(strcmp(tag,"VDZ")==0)  
    result = "Verb 'do'- 's' form";

else if(strcmp(tag,"VHB")==0)  
    result = "Verb 'have'- finite form";

else if(strcmp(tag,"VHD")==0)  
    result = "Verb 'have'- past tense form";

else if(strcmp(tag,"VHG")==0)  
    result = "Verb 'have'-ing form";

else if(strcmp(tag,"VHI")==0)  
    result = "Verb 'have'-infinitive form";
else if(strcmp(tag,"VHN") == 0)  
    result = "Verb 'have'- past participle";
else if(strcmp(tag,"VHZ") == 0)  
    result = "Verb 'have'- 's' form";
else if(strcmp(tag,"VM0") == 0)  
    result = "Modal auxiliary verb";
else if(strcmp(tag,"VVB") == 0)  
    result = "Verb - finite form";
else if(strcmp(tag,"VVD") == 0)  
    result = "Verb - past tense form";
else if(strcmp(tag,"VVG") == 0)  
    result = "Verb -ing form";
else if(strcmp(tag,"VVI") == 0)  
    result = "Verb - infinitive form";
else if(strcmp(tag,"VVN") == 0)  
    result = "Verb - past participle";
else if(strcmp(tag,"VVZ") == 0)  
    result = "Verb - 's' form";
else if(strcmp(tag,"XX0") == 0)  
    result = "Negative particle";
else if(strcmp(tag,"ZZ0") == 0)  
    result = "Alphabetical symbol";
else  
    result = "Portmanteau tag";

return result;
}

void divide_portmanteau(int col_num){
    char tag1[4];
    char tag2[4];
    int i,j,k;

    for(i=0; i < thelattice.lat[col_num].height ; i++){
        if(thelattice.lat[col_num].col[i].tag[3] == '.'){
            for(j=0; j<3; j++)
                tag1[j] = thelattice.lat[col_num].col[i].tag[j];
}
tag1[j] = '\0';
j++;

for(j;j<7;j++)
tag2[j] = thelattice.lat[col_num].col[i].tag[j];
tag1[j] = '\0';

for(k=0; k<thelattice.lat[col_num].height;k++){
    if((strcmp(tag1,thelattice.lat[col_num].col[k].tag)==0)||(strcmp(tag2,thelattice.lat[col_num].col[k].tag)==0)){
        thelattice.lat[col_num].col[k].prob = thelattice.lat[col_num].col[k].prob *
        (thelattice.lat[col_num].col[i].prob/2);
    }
}

thelattice.lat[col_num].col[i].prob = 0.0;

#include <stdio.h>
#include "datastructures.h"
#include <string.h>
#include <ctype.h>
#include <time.h>

#define YES "y\0"
define NO "n\0"
define FROMSTDIO "s\0"
define FROMFILE "f\0"
define FROMPARTICULARFILE "p\0"
define MAXLENGTH 1000

extern char **tokens_ptr;
extern int numtokens;
extern Posinfo *posinfo_ptr;
extern Lattice thelattice;
extern Tagging bptagseq;
extern int numunknowns;
extern char *suffix;

main(){

FILE *input_ptr;
FILE *output_ptr;
char search[MAXLENGTH];
Appendix 3 – Code

```c
char sentence[MAXLENGTH];
int i,k,s,letter,wordcount,unknownscount,duration,sentcount;
char menuchoice;
char anotheranalysis[5];
char qanotheranalysis[5];
char stdio_or_file[5];
char filename[15];
clock_t start;
clock_t end;
initialise_dbs();

/*prompt user for input type*/
printf("Would you like to use input from standard input or from a file?\n"); printf("Standard input & output - 's'\n"); printf("Input from & output to default files - 'f'\n"); printf("Input from particular file - 'p'\n"); printf("Please enter s, f or p : ");

while(gets(stdin_or_file)! = NULL){

    /*for getting input from a file*/
    if((strcmp(stdin_or_file,FROMFILE)==0)||(strcmp(stdin_or_file,FROMPARTICULARFILE) == 0)){

    /*if user wishes to enter his own filename*/
    /*print output to filename.out*/
    if((strcmp(stdin_or_file,FROMPARTICULARFILE)) == 0){
        printf("Please enter filename to read from :\n");
        gets(filename);

        if ((input_ptr = fopen(filename,"r")) == 0) {
            printf("Cannot read file : "%s"
",filename);
            exit(1);
        }else{
            printf("file :"%s" open\n",filename);
            strcat(filename,".out");
            if ((output_ptr = fopen(filename,"w")) == 0) {
                printf("Cannot write to file : "%s"
",filename);
                exit(1);
            }else{
                printf("file :"%s" open\n",filename);
            }
        }
    }

    /*to take the input from the default file*/
    }else{
        if ((input_ptr = fopen("text.txt","r")) == 0) {
            printf("Cannot read file :");
            exit(1);
        }else{
            printf("file :"text.txt" open\n");
        }
    }
```

120
if ((output_ptr = fopen("taggedtext.txt","w")) == 0) {
    printf("Cannot write to file : \"taggedtext.txt\"\n");
    exit(1);
} else {
    printf("file("taggedtext.txt") open\n");
}

/*pretty printing*/
printf("\n\n\t	");
for(k=0 ; k < 70 ; k++){
    printf("-");
}
printf("\n\n");
/*start the clock that times the whole tagging process*/
start = clock();
wordcount = unknownscount = sentcount = 0;
/*start extracting from the file*/
while((letter = fgetc(input_ptr)) != EOF){
    s = 0;
    if(isspace(letter)){
        sentence[s] = ' ';
    } else {
        sentence[s] = letter;
    }
    /*continue placing words in the sentence array until an end of sentence marker is encountered*/
    while((sentence[s] != '.') && (sentence[s] != '!') && (sentence[s] != '?') && ((letter = fgetc(input_ptr)) != EOF)){
        s++;
        if(letter == '\n'){  
            sentence[s] = ' ';
        } else {
            sentence[s] = letter;
        }
        /*a check to make sure that if a '.' is encountered - that it is not acting*/
        /*as a decimal point in a number - if so, include this in the sentence*/
        if((sentence[s] == '.') && (isdigit(sentence[s-1]))){
            letter = fgetc(input_ptr);
            s++;
        }
        sentence[s] = letter;
    }
    sentence[++s] = ' ';
    sentence[++s] = '"';
    /*perform the whole tagging function*/
    numunknowns = 0;
    tokenise(sentence);
    make_lattice();
    do_viterbi();
    find_tagging();
    sentcount = sentcount++;
}
wordcount = wordcount + numtokens;
unknownscount = unknownscount + numunknowns;
    fprintf(output_ptr,"
<sent>");
printf("\n<sent>");
for(i=1; i<bptagseq.length; i++){
    fprintf(output_ptr,"%s<%s> ",bptagseq.tags[i].word, bptagseq.tags[i].tag);
    printf("%s<%s> ",bptagseq.tags[i].word, bptagseq.tags[i].tag);
}
fprintf(output_ptr,"<sent>
");
printf("<sent>
");

printf("\n");
free_lattice();
free_bp();
fflush;
}

end = clock();
printf("\n\n");

/*to print the length of time that the tagging takes*/
if((duration = ((end-start)/CLOCKS_PER_SEC))>0){
    printf("\n\t\t\tStatistics\n");
    printf("\n\t\t\tTime taken : %d seconds for %d words of text\n",duration,wordcount);
    printf("\n\t\t\tNo. of unknown words encountered : %d\n",unknownscount);
    printf("\n\t\t\tNo. of sentences encountered (no. of lattices built : %d\n",sentcount);
    printf("\n\t\t\tSpeed of tagger : %d words/sec\n",(wordcount/duration));
}else{
    printf("\n\t\t\tTime taken : less than 1sec to tag %d words of text\n",wordcount);
    printf("\n\t\t\tNo. of unknown words encountered : %d\n",unknownscount);
    printf("\n\t\t\tNo. of sentences encountered (no. of lattices built : %d\n",sentcount);
}
/*pretty printing!*/
printf("\n\t\t\t\n");
for(k=0 ;k < 70 ;k++){
    printf("-");}
}
printf("\n");
/*close files*/
fclose(input_ptr);
fclose(output_ptr);

/*to tag a sentence entered from the terminal screen*/
else if(strcmp(stdio_or_file,FROMSTDIO) == 0){
    fflush(NULL);
    printf("Please enter your sentence:\n");
    if(gets(search) != NULL){
        strcpy(anotheranalysis, YES);
        while(strcmp(anotheranalysis, YES) == 0){
printf("What kind of analysis would you like?: \
");
printf("Simple - 1\n");
printf("Showing lattice - 2\n");
printf("Full explanation - 3\n");
printf("Please enter 1, 2 or 3 : ");
menuchoice = getchar();
switch(menuchoice){
case '1':
    /*to simply print the best path tagging for the sentence*/
    numunknowns = 0;
tokenise(search);
make_lattice();
do_viterbi();
find_tagging();
unknownscount = numunknowns;
printf("The most probable tag sequence is: \n");
for(i=1; i<bptagseq.length; i++){
    printf("word : %s\t tag : %s \", bptagseq.tags[i].word, bptagseq.tags[i].tag);
    printf("\t (%s)\n", explain_tag(bptagseq.tags[i].tag));
}
printf("No. of unknown words encountered : %d\n",unknownscount);
free_lattice();
free_bp();
printf("\n");
break;

case '2':
    /*to print the entire lattice for the sentence*/
    numunknowns=0;
tokenise(search);
make_lattice();
do_viterbi();
print_lattice();
find_tagging();
unknownscount = numunknowns;
printf("The most probable tag sequence is: \n");
for(i=1; i<bptagseq.length; i++){
    printf("word : %s\t tag : %s \", bptagseq.tags[i].word, bptagseq.tags[i].tag);
    printf("\t (%s)\n", explain_tag(bptagseq.tags[i].tag));
}
printf("No. of unknown words encountered : %d\n",unknownscount);
free_lattice();
free_bp();
break;

case '3':
    /*to print a full breakdown for the entire tagging process*/
    numunknowns = 0;
tokenise(search);
make_lattice();
print_do_viterbi();
find_tagging();
sentcount = 1;
wordcount = numtokens-1;
unknownscount = numunknowns;
printf("The most probable tag sequence is: \n");
for(i=1; i<bptagseq.length; i++){
    printf("word : %s\t tag : %s \n",bptagseq.tags[i].word, bptagseq.tags[i].tag);
}
printf("Statistics\n");
printf("No. of words in sentence : %d\n",wordcount);
printf("No. of unknown words encountered : %d\n",unknownscount);
free_lattice();
free_bp();
break;
default:
    printf("Error - unknown command\nNumber must be either 1, 2 or 3\n");
    break;
}fflush(NULL);
    printf("Would you like a different type of analysis? (y for yes, n for no) : ");
if(gets(qanotheranalysis)!= NULL{
    if((strcmp(qanotheranalysis, YES) == 0)){
        strcpy(anotheranalysis,YES);
    }else{
        strcpy(anotheranalysis, NO);
    }
}
else{
    printf("Invalid command\n");
}
printf("Would you like to use input from standard input or from a file?\n");
printf("Standard input & output - type 's'\n");
printf("Input from & output to files - type 'f'\n");
printf("Input from particular file - 'p'\n");
printf("Please enter s, f or p : ");
}

score.c – functions and a main to score and evaluate the tagger against pretagged texts

#include <stdio.h>
#include "datastructures.h"
#include <string.h>
#include <ctype.h>
#include <time.h>
define MAXLENGTH 1000
Appendix 3 – Code

char **tokens_ptr;
extern int numtokens;
extern int numunknowns;
extern Posinfo *posinfo_ptr;
extern Lattice thelattice;
extern Tagging bptagseq;
Tagging correct_tags;

int tokenise_score(char *sentence);

main(){

FILE *input_ptr;
FILE *output_ptr;
char word[30];
char tag[10];
char sentence[MAXLENGTH];
char filename[15];
int wordcount,unknownscount,sentcount,incorrect = 0;
int duration;
int k,l,i,j;
clock_t start;
clock_t end;

initialise_dbs();

printf("Please enter name of file to be scored: ");

gets(filename);

if ((input_ptr = fopen(filename,"r")) == 0) {
    printf("Cannot read file : \\
",filename);
    exit(1);
}else
    printf("file \"%s\" open\n",filename);

strcat(filename, ".out");

if ((output_ptr = fopen(filename,"w")) == 0){
    printf("Cannot write to file : \\
",filename);
    exit(1);
}else
    printf("file\"%s\") open\n",filename);

start = clock();
wordcount = 0;
while(fgets(sentence,MAXLENGTH,input_ptr)!=NULL){
    if(ispace(sentence[0])){
        continue;
    }
    else if(isalnum(sentence[0])){
        if((correct_tags.tags = (Tag *)calloc(MAXLENGTH,sizeof(tag)))==NULL){
            fprintf(stderr, "ERROR: Calloc failed");
            exit(1);
        }
    }

    correct_tags.length = MAXLENGTH;
    if((tokens_ptr = (char **)calloc(MAXLENGTH,sizeof(word)))==NULL){
        fprintf(stderr, "ERROR: Calloc failed");
        exit(1);
    }

    /*start the tagging process*/
    numunknowns = 0;
    tokenise_score(sentence);
    make_lattice();
    do_viterbi();
    find_tagging();
    sentcount++;
    wordcount = wordcount + numtokens;
    unknownscount = unknownscount + numunknowns;
    fprintf(output_ptr,"
<sent>");
    printf("<sent>");
    for(i=0; i<bptagseq.length; i++){  
        fprintf(output_ptr,"%s<%s> ",bptagseq.tags[i].word, bptagseq.tags[i].tag);
        printf("%s<%s> ",bptagseq.tags[i].word, bptagseq.tags[i].tag);
    }

    /*compare the pretagged file with the newly tagged one*/
    if(strcmp(bptagseq.tags[i].word,correct_tags.tags[i].word)==0){
        if(strcmp(bptagseq.tags[i].tag,correct_tags.tags[i].tag)!=0)
            incorrect++;
    }else
        printf("Error - mismatched words!");

}

fprintf(output_ptr,"<sent>\n");
printf("<sent>\n");

printf("\n");
free_lattice();
free_bp();
for(i=(numtokens-1);i>=0;i--){
    free(correct_tags.tags[i].tag);
    correct_tags.tags[i].tag = NULL;
}
Appendix 3 – Code

```c
free(correct_tags.tags);
correct_tags.tags = NULL;
fflush;
}
}

end = clock();
printf("n
");
/*to print the length of time that the tagging takes*/
if((duration = ((end-start)/CLOCKS_PER_SEC))>0){
    printf("Time taken : %d second(s) for %d words of text\n",duration,wordcount);
    printf("Speed of tagger when comparing: %d words/sec\n",(wordcount/duration));
    printf("No. of sentences encountered (no. of lattices built : %d\n",sentcount);
    printf("No. of unknown words encountered : %d\n",unknownscount);
    printf("Percentage of unknown words\n",((double)unknownscount/(double)wordcount)*100);
    printf("Accuracy : %f \%",((double)wordcount-incorrect)/((double)wordcount)*100);
}else{
    printf("Time taken : less than 1sec to tag %d words of text\n",wordcount);
    printf("No. of sentences encountered (no. of lattices built : %d\n",sentcount);
    printf("No. of unknown words encountered : %d\n",unknownscount);
    printf("Percentage of unknown words\n",((double)unknownscount/(double)wordcount)*100);
    printf("Accuracy : %d \%",((double)wordcount-incorrect)/((double)wordcount)*100);
}
/*pretty printing!*/
printf("n\ttt\t");
for(k=0 ;k < 70 ;k++){
    printf("-\t");
}
printf("n");
/*close files*/
fclose(input_ptr);
fclose(output_ptr);
}

int tokenise_score(char *sentence){
    extern Tagging correct_tags;
    char word[30];
    char tag[10];
    char *temptag;
    char *tempword;
    int k,j;
    int i=0;
    
```
int j=0;
    while(sentence[i] != '\n'){  
        k=0;
        l=0;
        while((sentence[i] != ':')&&(sentence[i] != '\n')){
            while(isspace(sentence[i])){
                i++;
            }
            word[k] = sentence[i];
            k++;
            i++;
        }
        word[k] = '\0';
        tokens_ptr[j] = strdup(word);
    } /*the ':' has been reached*/
    i++;
    while((sentence[i] != ')')&&(sentence[i] != '\n')){
        tag[l]=sentence[i];
        l++;
        i++;
    }
    tag[l] = '\0';
    temptag = strdup(tag);
    correct_tags.tags[j].word = tokens_ptr[j];
    correct_tags.tags[j].tag = temptag;
    correct_tags.tags[j].prob = 0;
    j++;
    numtokens = j;
    return 1;
}

**dbtest.c**- to observe the various hash tables in the database.

```
#include "lookup.h"
#include <stdio.h>
#include <string.h>

//Posinfo *alltagstructs;

extern Posinfo *posinfo_ptr;
extern Posinfo *affix_ptr;
double tgtp;

main () {

cchar search[42];
cchar search2[8];
cchar *ans;
cchar menuchoice;
int nooftags,i;
```
flush(NULL);

initialise_dbs();

printf("Which database would you like to access?\n");
printf("For 'all tags given a certain word' type 1\n");
printf("For 'bigram probabilities' type 2\n");
printf("For 'all tags given a certain word (based on it's suffix only)' type 3\n");
printf("Please type 1,2 or 3 - ");

while((menuchoice=getchar())!=NULL){
  switch(menuchoice){
  case '1':
    printf("\nPlease enter the word : ");
    if(scanf("%s", &search)!=NULL){
      if((ans = prob_tags_given_word(search)) != NULL)
        printf("The tags for that word are : \n%s\n", ans);
    }break;
  case '2':
    printf("\nPlease enter the first tag of the bigram : ");
    if(scanf("%s", &search) != NULL){
      printf("\nPlease enter the second tag of the bigram : ");
      if(scanf("%s", &search2)!=NULL){
        prob_tag2_given_tag1(search2,search);
        printf("\nThe probability of that bigram is : %f \n", tgtp);
      }
    }break;
  case '3':
    printf("Please enter the word : ");
    scanf("%s", &search);
    if((ans = most_prob_affix(search)) != NULL)
      printf("The tags for that largest suffix of that word are : \n%s\n", ans);
    break;
    default:
      printf("Error - unknown command\nNumber must be either 1, 2 or 3\n");
break;
}
printf("\n\nWhich database would you like to access?\n");
printf("For 'all tags given a certain word' type 1\n");
printf("For 'bigram probabilities' type 2\n");
printf("For 'all tags given a certain word (based on its suffix only)' type 3\n");
printf("Please type 1, 2 or 3 - ");
getchar();
}
}