Statistical Lexical Disambiguation

George F. Foster
School of Computer Science
McGill University, Montreal

A Thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
of the degree of Master of Science

Abstract

This research is concerned with a Markov-model-based solution to the problem of lexical disambiguation in natural language. We investigate three ways of improving current statistical solutions to this problem. First, three techniques for reducing or eliminating the necessity for a manually tagged training corpus are described. Two completely automatic methods are shown to be inadequate, but a semi-automatic method is found to be effective in reducing the number of tokens which must be manually analyzed. Second, various methods of improving the tagging performance of an automatic disambiguator are described. Good-Turing and combined-order estimation techniques are shown to yield small gains. Finally, an effective algorithm for detecting and flagging potential tagging errors is presented.

Dans le cadre de ce projet de recherche, nous nous intéressons à résoudre le problème de la désambiguisation lexicale de la langue naturelle en utilisant un modèle de Markov. Nous examinons trois méthodes visant à améliorer les solutions statistiques existantes pour ce problème. Nous décrivons d'abord trois techniques qui réduisent ou éliminent la nécéssité d'un corpus d'entraînement étiqueté à la main. De ces techniques, deux qui sont entièrement automatiques s'avèrent inadéquates, mais une troisième, semi-automatique celle-là, réduit effectivement le nombre d'occurences qui doivent être analysées manuellement. Nous décrivons ensuite différentes méthodes pour améliorer la qualité de l'étiquetage d'un système de désambiguisation automatique. Nous démontrons que les techniques d'estimation de Good-Turing et d'ordre combiné ne produisent que de maigres améliorations. Enfin, nous présentons un algorithme pour détecter et marquer les étiquetages possiblement erronés.

Acknowledgements

I would like to express my deepest gratitude to Pierre Isabelle for agreeing to serve as my unofficial advisor when he had no reason except kindness for doing so. He has provided inspiration and ideas, and has made it financially possible for me to complete this work.

I would also like to thank Michel Simard for translating my abstract on very short notice, Diane Goupil for her helpful comments, and Luc Boulianne for his understanding about my abuse of computer resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Lexical Disambiguation

Most words in human languages are ambiguous: they can have more than one sense, depending on the context in which they appear. In written language, this phenomenon is called *homography* and refers to different meanings attached to a lexical form[24]. *Lexico-syntactic ambiguity* is a type of homography in which a lexical form assumes different grammatical roles. The distinction between homography and lexical ambiguity[1] is illustrated by the following phrases:

a brush with death

a hair brush

brush your teeth

Here each instance of the word *brush* is distinct from a homographic point of view because each has a different meaning. The first two instances are lexically indistinguishable however, because *brush* acts as a noun in both.

The problem of resolving lexical ambiguities is known as *lexical disambiguation* or *tagging*. The latter term refers to the assignment of a single label or *tag* to each word which denotes its grammatical category.

[1]Henceforth *lexical ambiguity* is used as shorthand for the more accurate but unwieldy term *lexico-syntactic ambiguity*.

1.2 Computer Solutions

Lexical disambiguation is an important problem in computational linguistics because it is crucial to parsing, which in the traditional view is essential to a complete analysis of language. Disambiguating programs are also useful in their own right, and have potential applications in many language-related areas such as lexicography, speech recognition and synthesis, error correction, machine translation and text analysis.

It has been assumed that the disambiguation problem is not easily separable from higher levels of analysis in the traditional hierarchy: syntactic, semantic and pragmatic. However, recent work has shown that higher level knowledge is required in only a small minority of cases. Programs which use statistical methods and rely on very local context have achieved correct disambiguation rates of over 95%.

1.3 This Project

Although existing programs perform impressively, there are some areas in which improvement is possible. The purpose of this thesis is to investigate three of them: training—finding ways in which the manual effort required to train existing taggers can be reduced or eliminated; performance—finding ways of improving tagging performance; and error detection—having the disambiguator identify tag assignments about which it is uncertain.

In each area, several alternate approaches were compared. To facilitate testing, an experimental statistical tagging system was constructed. It is capable of making category assignments with a fairly low error rate for any text in any language for which a good dictionary is available.

1.4 Organization

The next chapter describes the tagging problem in more detail and outlines the type of statistical solution considered here. Previous work is reviewed and the deficiencies which motivated the current project are identified. Some ways in which they can

be rectified are proposed.

Chapter 3 describes the general methods used for testing. An overview of the software is given, and its theoretical basis is set out. Test corpora and performance metrics are described.

Chapters 4, 5 and 6 describe the results of the investigation in each area: training, performance and error flagging, respectively. The alternate methods used to attack each problem are described and tagging results are presented. Chapter 7 summarizes the findings.

The appendices contain documentation for the tagging system and a listing of category sets used for the test corpora.

Chapter 2

Background

2.1 The Nature of the Problem

In some cases, the disambiguation problem is easy enough to be solved by examining only the local context of an ambiguous word. For example, in the sentence

Electricians always wire carefully.

knowledge of the preceding word gives a strong indication that *wire* is a verb and not a noun. In other cases, such as

I prefer electricians that wire carefully.

Hand me that wire carefully.

there is no such local clue, but a complete syntactic analysis of the sentence would provide enough information for disambiguation. In the most general case, however, semantic or pragmatic knowledge is necessary. The sentence

Fish wire carefully.

could be interpreted as an injunction to a human wire-fisher or as a remarkable discovery about fish; only knowledge about which of the two is more likely in a wider context can be of help in resolving the ambiguity.

2.2 Traditional Approaches

The existence of sentences like the one in the last example has led computational linguists to suppose lexical disambiguation a fairly hard problem. Both syntactic and semantic analyses are clearly required for the general solution; it has also been assumed that both are required for reasonable tagging performance. Under this assumption, there is not much point in considering the problem in isolation; more sensible would be to integrate it into deeper levels of analysis such as parsing. This view seems to have been widely accepted: in a fairly recent survey of natural language parsers [7], for example, there is no mention of lexical disambiguation as a separate problem.

There were some early attempts to create stand-alone disambiguating programs for special purposes [41, 33]. The latest was Greene and Rubin's TAGGIT program [33], written circa 1971 to tag the million-word Brown corpus of American English. TAGGIT used a set of hand-written disambiguation rules which apply over a context of two words on either side of the word to be tagged. It attained a success rate of about 77%—low enough to seem to corroborate the assumption that disambiguation is hard.

2.3 The Statistical Paradigm

Recent developments have proven this assumption to be false. Statistical algorithms which work in linear time complexity and use only the previous part of speech and the current word to predict the current part of speech have been able to categorize over 95% of their input correctly. It appears that a significant proportion of sentences are of the type exemplified by "Electricians always wire carefully", which can be disambiguated reliably on the basis of local context. Many of the remaining sentences seem to fall into distinct patterns of use, so that the most likely categorization is correct in a large majority of cases.

Disambiguation has been one of the most striking successes to date in what might be termed an emerging statistical (or empirical, or corpus-based) paradigm in computational linguistics. Large scale statistical methods were first applied to

natural language modelling by researchers in speech recognition about fifteen years ago [22, 43, 37, 3]. In the early 1980's, workers associated with UCREL[1] [49], aware of achievements in speech recognition, built a successful statistical tagger as an aid to disambiguating the LOB (Lancaster/Oslo-Bergen) corpus [38]. Several other researchers have subsequently duplicated or built upon their work, as described below. Similar methods are beginning to be used for many other natural language applications including lexicography [39, 17, 18], machine translation [8, 9, 11, 20, 19], parsing [6, 13, 23, 49, 51], text analysis [14], and spelling correction [1, 16, 49, 25].

A hallmark of this paradigm is the use of large computerized corpora as sources of statistics from which parameters for a probabilistic model are automatically inferred. Currently, the model of choice for disambiguation (and many other problems as well) is one which is based, with varying degrees of rigour, on the *Hidden Markov Model* (HMM) formalism [50]. Another model—one which borrows more from linguistics—is that of a stochastic grammar [51, 23].

The statistical paradigm differs from the traditional, rule-based paradigm in three key ways which make it more suited to large scale practical applications (at least at the current state of the art). First, the models used are often simpler and algorithms have lower time complexity. Second, model parameters are estimated automatically, rather than manually via detailed linguistic analysis. This allows them to be tailored to any domain, including very large ones, with a minimum of human effort. Finally, it is easy to create statistical systems which do not "break" when confronted with difficult problems. This provides the robustness necessary to deal with the vagaries of natural language over large and varied domains.

The statistical approach is not without drawbacks. Simple models are often demonstrably too crude to attain the performance of which more sophisticated models are theoretically capable. This is certainly the case, for example, with lexical disambiguators which use only the previous part of speech and the current word. Another problem is that some models, particularly HMMs, tend to replace sophistication with brute force. Although time complexity may not suffer, space complexity usually does. A straightforward implementation of a third-order HMM

[1]The Unit for Computer Research in the English Language, affiliated with Lancaster University

to predict word sequence probabilities for a large vocabulary, for example, would require on the order of 10^{16} parameters. Cleverness in implementation and ample resources are required to overcome such difficulties.

2.4 Statistical Disambiguation

Six programs which use statistics for lexical disambiguation are described in the literature, due to Church [13], de Marcken [26], DeRose [28], Hindle [34], Merialdo [45] and the UCREL team [49]. These are summarized in (roughly) chronological order in the sections which follow.

To provide a framework for comparison, it is worthwhile to first sketch the basic plan of a statistical tagger. The type we consider here disambiguates a word in some textual context by computing a probabilistic score for every grammatical category which is valid for the word, then using the scores to decide the correct category. Three aspects of this process are fundamental: the form of the scoring function, the means of estimating its probabilistic components, and the method of choosing the correct category. These are discussed below under the headings *scoring*, *estimation* and *tagging*.

2.4.1 Nomenclature

We begin with a short list of definitions:

word A word in the usual sense, but including punctuation marks. A word is ambiguous if it has more than one permissible grammatical category.

token An occurrence of a word in a text (there are 19 words and 21 tokens in this sentence). This term is sometimes used in a wider sense to denote occurrences of entities other than words.

n-gram A sequence of n words or tokens.

category A grammatical category.

tag A label assigned to a token, denoting a single category.

n-cat A sequence of n categories or tags.

path A sequence of tags which represents a possible interpretation for a corresponding sequence of tokens.

tagged corpus A body of text where each token is tagged with the grammatical category to which it belongs.

Assume that a tagger has vocabulary V, category set C and operates on token sequence O. The current path is I. These are:

$$\begin{aligned} V &= \{v_1, \ldots, v_M\} \\ C &= \{c_1, \ldots, c_L\} \\ O &= o_1 \ldots o_T,\ o_t \in V\ \forall t \\ I &= i_1 \ldots i_T,\ i_t \in C\ \forall t \end{aligned}$$

Each word $v \in V$ has a set $C_v \subset C$ of $L_v < L$ permissible categories.

2.4.2 Scoring

A function to score a tag for some token can incorporate both lexical and contextual information. Lexical scores measure the strength of the association between the tag and the token. Contextual scores measure the likelihood of the tag given the context in which the token appears. Although it is possible to imagine many different ways of computing scores, previous work has demonstrated that nothing too elaborate is necessary. Because of this, and for reasons of efficiency, we limit the form of scoring function considered here to the following:

$$S(i_t, o_t, I) = S_L(i_t, o_t) S_C(i_{t-n_1} \ldots i_{t+n_2}),$$

where S_L is the lexical scoring component, S_C the contextual scoring component and the argument of S_C is some n-cat in I (where $n = n_1 + n_2 + 1$) which contains i_t. In other words, a scoring function is the product of a lexical scoring function which depends only on the current token and tag, and a contextual scoring function which depends only on a sequence of n tags, one of which is the current tag. The *order* of the scoring function is defined as the length, n, of the tag sequence used for context. It is important to note that this type of scoring function evaluates a

tag only with respect to a single path I, although in general a tag can be on many paths. The task of choosing the path(s) for which scores are to be computed is considered part of the tagging mechanism and is discussed in section 2.4.4.

It is most common to use probability estimates as scoring function components. The conditional probability of the current tag given the current word is often used as the lexical score, with the conditional probability of the current tag given the previous $(n-1)$-cat used as the contextual score. For example, with order = 3 this would be[2]:

$$S(i_t, o_t, I) = \Pr(i_t|o_t) \Pr(i_t|i_{t-2}i_{t-1}). \tag{2.1}$$

Although it is possible to come up with a probabilistic interpretation for this product, it should be noted that its use is motivated mainly by practical considerations—there are other scoring functions (described below) which are of more theoretical interest.

2.4.3 Estimation

At some level, the components of a scoring function are usually either probabilities or are congruent to probabilities. To use the scoring function, these probabilities must be *estimated* from some sample of text in such a way as to be representative of the domain over which the tagger is to operate. The process by which this is accomplished is called *training* and the text sample used is called the *training corpus.*

The obvious way to estimate probabilities is to use relative frequency counts from a tagged corpus. For example, suppose that O is a tagged corpus with tag sequence I; assuming the scoring method of equation 2.1, a relative frequency estimate for the lexical score of category c with word v would be:

$$\begin{aligned} \Pr(c|v) &= \Pr(c,v)/\Pr(v) \text{ (Bayes' law)} \\ &\simeq \frac{f(c,v)}{T} / \frac{f(v)}{T} \\ &\simeq f(c,v)/f(v) \end{aligned} \tag{2.2}$$

[2]Pr will be used to denote both probability and estimated probability unless there is a need to distinguish between the two.

where $f(e)$ is the frequency of event e in O. Similarly, a relative frequency estimate for the contextual score of category c_k, given the previous 2-cat c_ic_j would be:

$$\begin{aligned} \Pr(c_k|c_ic_j) &= \Pr(c_ic_jc_k)/\Pr(c_ic_j) \\ &\simeq \frac{f(c_ic_jc_k)}{T-2}/\frac{f(c_ic_j)}{T-1} \\ &\simeq f(c_ic_jc_k)/f(c_ic_j) \end{aligned} \tag{2.3}$$

The estimates in this example are called *maximum likelihood* estimates, because they have the property of maximizing the estimated probability of occurrence of the training corpus. An unfortunate side-effect of this is that they assign a probability of zero to any event which does not occur in the training corpus. (This can be readily verified by inspecting the numerator in the last line of equation 2.3.) Intuitively, it is unwise to assign zero probability to any grammatical construct because it has not occurred in a specific corpus, no matter how large the corpus or how unlikely the construct may seem. The problem is even worse for lexical probabilities, because certain valid word/category combinations are extremely rare.

If a statistical tagger attempts to disambiguate a token sequence which contains n-cats or word/category combinations to which it assigns a zero score, it is quite likely that it will fail to find a path. To avoid this problem, maximum likelihood estimates are usually modified in some way. The most common is to simply add a small amount to each score. More sophisticated methods of dealing with the problem are available, however, and are discussed below.

2.4.4 Tagging

Once the tagger has been trained, it can be used to disambiguate. We consider two types of algorithm to choose a path for the token sequence: *path-based* and *token-based.*

2.4.4.1 Path Based Tagging

Path-based tagging is the simpler of the two. It involves enumerating all possible paths corresponding to a token sequence and scoring each by taking the product of the scores of the tags which it comprises. The path with the highest score is then selected as the correct one for the sequence.

Path-based tagging is potentially very expensive, since the number of possible paths is exponential in the length of the token sequence. Fortunately, resource requirements can be reduced through the use of the well-known Viterbi algorithm [53] (or a variant thereof) to find the path with the highest score. This algorithm relies on the fact that the scoring function depends only on local context, and is of time and space complexity which are linear in the length of the sequence and polynomial, of order equal to the order of the scoring function, in the maximum number of categories per word. It is described in detail in chapter 3.

2.4.4.2 Token Based Tagging

An alternate disambiguating algorithm is token-based tagging. While path-based tagging picks the most likely grammatical interpretation for a sequence of tokens, token-based tagging picks the most likely interpretation for a single token, given the sequence in which it occurs. Each tag for a token is assigned a *total score* and the tag with the highest total score is chosen.

Total scores are not the same as the tag scores computed by the scoring function: while each tag has a single total score, it will in general have many tag scores, one for each path to which it belongs. A very simple way of computing a total score would be to ignore context and use the lexical component of the tag score, since this is the same for all paths. A more sophisticated total score for a tag is the sum of path scores for all paths to which it belongs. There is an analog to the Viterbi algorithm, sometimes called the *FB* (for "forward backward") algorithm, which allows all paths passing through a tag to be summed in linear time and space complexity.

2.4.4.3 Sequence Segmentation

For very long token sequences, even linear space complexity can be prohibitive. Space requirements can be reduced considerably by dividing such sequences into segments. Since the scoring function uses only n tags of context, an unambiguous $(n-1)$-gram in any sequence effectively splits it in two as far as scoring is concerned; neither ambiguous half has any affect on the other. A tagger can take advantage of

this by identifying segments which begin and end in unambiguous $(n-1)$-grams. For low n, such segments are usually fairly short, since most punctuation is unambiguous. For higher n, this method of segmentation could be simulated by using the last few tags of the previous segment after it has been disambiguated. In practice, however, n is low enough that this is not a necessity. Another method of segmenting a sequence would be to try to divide it into sentences, for which standard boundary conditions could be assumed. Although it is appealing to base path judgements on a well-defined grammatical unit, sentence boundaries can sometimes be difficult to identify. Sentences can also be long: one can always find extreme examples, such as the last chapter of Joyce's *Ulysses*, which would defeat a sentence-based tagger[3].

2.4.5 Tagging Example

An example will serve to illustrate the operation of a statistical tagger. We use the second order scoring function

$$S(i_t, o_t, I) = \Pr(o_t|i_t) \Pr(i_t|i_{t-1}) \tag{2.4}$$

which differs from the more common version of equation 2.1 in having a "reversed" lexical component. Path scores computed with this function have a probabilistic interpretation as $\Pr(I, O)$, ie the joint probability of the token sequence with a particular path. (This is the scoring function used by an HMM; it is described further in chapter 3).

Consider the following sentence, extracted from a corpus of parliamentary Hansard proceedings:

> "Tous les visiteurs étrangers en ont *conclu ce que nous* savons depuis toujours, à savoir que notre pays est incomparable."

The sequence *conclu ce que nous* can be disambiguated in isolation by a tagger with a second order scoring function, because both *conclu* and *nous* are unambiguous. It is analyzed as:

[3]That is, one which uses a straightforward implementation of the Viterbi algorithm. As Y. Normandin has pointed out, a *partial backtrace* technique of attemting to trace all currently active paths back to a single common node would probably be effective in reducing the space complexity of a sentence-based tagger to an acceptable level.

	CAT	2-CAT				WORD/CAT	
		CS	DT	PN	VB	ce	que
CS	105	1	29	40	3	–	57
DT	833	1	2	0	3	16	–
PN	293	0	5	58	212	12	16
VB	659	36	143	12	130	–	–

Table 2.1: Observed frequencies from 5,000 word training corpus

	CAT	2-CAT				WORD/CAT	
		CS	DT	PN	VB	ce	que
CS	113	2	30	41	4	–	57.5
DT	857	2	3	1	4	16.5	–
PN	324	1	6	59	213	12.5	16.5
VB	1045	37	144	13	131	–	–

Table 2.2: Modified frequencies

conclu/VB ce/(DT | PN) que/(CS | PN) nous/PN

where the tags denote basic parts of speech: CS—subordinating conjunction, DT—determiner, PN—pronoun and VB—verb.

The first step is to obtain a training corpus and collect frequencies from it. For this example, a 5,000 word hand-tagged subset of the Hansard corpus was used. The frequencies of the categories, 2-cats and word/tag combinations required for the above sequence are given in table 2.1.

The next step is to modify the observed frequencies so that no probability estimates will be zero. To do this, we assume that the training corpus has been augmented by another, in which every possible 2-cat and word/tag combination occurs exactly once. To avoid swamping the observed data, each word/tag combination in the fictitious corpus is counted as $1/L_v$ occurrences, where L_v is the number of possible categories for word v. Table 2.2 displays the results of this modification.

In the observed corpus, marginal sums taken over the frequencies of joint events

	Pr(CAT|X)				Pr(X|WORD)	
X	CS	DT	PN	VB	ce	que
CS	.014	.216	.295	.029	–	.509
DT	.002	.003	.001	.005	.019	–
PN	.003	.018	.180	.651	.039	.051
VB	.053	.208	.019	.189	–	–

Table 2.3: Contextual and lexical probabilities

equal the frequencies of individual events. For example, for any category c:

$$\sum_{i=1}^{M} f(c, v_i) = f(c).$$

A consequence of using a fictitious corpus to augment frequencies is that, in general, this relation will not hold: modified frequencies and any probability estimates made from them will be inconsistent. In order to preserve a probabilistic interpretation for scores, the frequencies of individual events can be *defined* as marginal sums over joint events for the purposes of estimation. The "CAT" column in table 2.2 contains the frequencies of individual categories which have been computed from joint word/category frequency sums in this way. (The same was done with 2-cat frequency sums, but the results are not listed.) Note that summed frequencies can be quite different from observed frequencies. That of the category *VB*, for example, is about 1.5 times greater than the observed frequency, due to the fact that many words in the vocabulary can act as verbs.

Table 2.3 contains probabilities estimated from the modified frequencies, rounded to three digits. Lexical probabilities for *conclu* and *nous* are not required because they can make no difference to the outcome.

Once estimates have been made, the sequence can be tagged. Table 2.4 lists the four possible paths for this sequence.

The number which follows each tag in each path is the partial path probability at that point; the number after the last tag in each path is the probability for that path. A path-based method would tag the sequence as

conclu/VB ce/DT que/CS nous/PN

1. VB(1.0×10^{0}) PN(7.2×10^{-4}) CS(1.1×10^{-6}) PN(3.3×10^{-7})

2. VB(1.0×10^{0}) PN(7.2×10^{-4}) PN(6.7×10^{-6}) PN(1.2×10^{-6})

3. VB(1.0×10^{0}) DT(4.0×10^{-3}) CS(4.7×10^{-6}) PN(1.4×10^{-6})

4. VB(1.0×10^{0}) DT(4.0×10^{-3}) PN(2.3×10^{-7}) PN(4.2×10^{-8})

Table 2.4: Alternate paths with partial scores

because path 3 in table 2.4 has the highest score. A token method which was based on the sum of the scores of all paths which pass through each tag would give:

conclu/VB ce/PN que/CS nous/PN

In this case, both methods give the wrong answer, although the token-based method is closer. It is interesting to note the difference made by the frequency modification step. If the observed frequencies had been used to compute lexical probabilities, the path with the highest score would have been path 2—the correct interpretation—instead of path 3. Of course, this cannot be taken as evidence that modified estimates are always worse than maximum likelihood. In fact, the problem of distinguishing between a relative pronoun and a subordinating conjunction is a good example of one which a 2-cat-based model is too weak to handle.

2.5 The UCREL Tagger

The UCREL tagger [49] is part of a system called CLAWS (Constituent Likelihood Automatic Word Tagging System), which analyzes raw input text and produces tagged output text in a series of stages. The actual tagging is performed by two of the stages, called *idiom tagging* and *tag disambiguation.* Idiom tagging is a non-statistical procedure which attempts to identify and tag common expressions. The details are not pertinent here, but UCREL reports a performance contribution of 3% for this component.

2.5.1 Scoring

The tag disambiguation component uses lexical scores which are based on a small set of qualitative "rarity markers". Each category which is valid for a word is labelled with a rarity marker which indicates its likelihood with respect to the word. The lexical scores themselves are scaling factors associated with the rarity markers, whose values were arrived at by "trial and error".

Contextual scores are second order; for the final version described in [49], they consist of the estimated ratio of the joint probability of the previous tag with the current tag, to the product of their individual probabilities[4]. Some heuristically selected 3-cats are also used to modify the 2-cat-based scores in this version.

2.5.2 Estimation

The tag disambiguation component of CLAWS was originally trained on the tagged Brown Corpus (the category set for the LOB corpus is quite similar to that for the Brown), then on the LOB corpus as more of it was disambiguated.

The estimation problem for lexical scores in this system is that of assigning rarity markers to the valid categories for each word; how this was accomplished is not specified in [49]. Estimation for contextual scores was by relative frequencies, with a small value added to each zero score.

2.5.3 Tagging

The tagging method is path-based, augmented by a token-based verification step. First the optimum path is found. Then, for each token, the total score of each valid tag is computed by summing the scores of all paths which pass through that tag. If the tag on the optimum path has a total score which exceeds that of any other tag for the token by some threshold (a separate threshold is used for each category), it is selected. If not, CLAWS does not disambiguate that token but instead produces a list, sorted by total score, of all possible tags and their total scores.

[4]This is quite similar to the information-theoretic concept of *mutual information*, defined as $\log \Pr(a, b)/(\Pr(a) \Pr(b))$, which measures the strength of the association between events a and b.

A serious deficiency in CLAWS is that it does not make use of the Viterbi algorithm, but actually enumerates all possible paths in order to compute their probabilities. When this causes it to run out of space, it resorts to "...a simpler method...which is an approximation to the method described here..." ([49], p 49). The alternate method is not described, nor is indication given of how often it had to be employed during tagging.

Final success rates for CLAWS on the LOB corpus (presumably after having been trained on the LOB) are 90% of ambiguous tokens correctly tagged and 96–97% of all tokens correctly tagged.

2.6 DeRose's Tagger

In creating his VOLSUNGA tagger [28], DeRose sought to improve on CLAWS in two ways: to increase efficiency by using the Viterbi algorithm instead of path enumeration for finding the best path; and to create a cleaner model by eliminating "unsystematic augments": idiom tagging, qualitative lexical scoring, 3-cat-based score modification, and token-based tagging. Despite being simpler and using less information than CLAWS, VOLSUNGA achieves similar performance.

2.6.1 Scoring

Lexical scores are computed from conditional word/tag probabilities as in equation 2.1, except that excessively high scores are "normalized" by truncating them at some (unspecified) threshold value. Contextual scores are computed by the second order analog to equation 2.1.

2.6.2 Estimation

VOLSUNGA was trained on the Brown corpus. Scores were estimated using relative frequencies, apparently without modification. The problems with this estimator would not have surfaced if, as was probably the case, DeRose trained VOLSUNGA on the entire Brown corpus and thereafter confined his testing to portions of it.

2.6.3 Tagging

Tagging is based solely on the best path. Performance on the Brown corpus without using lexical scores was 92–93%; performance with lexical scores was 96%.

2.7 Church's Tagger

Church's tagger [13], is similar to deRose's, except that it uses third order contextual scoring.

2.7.1 Scoring

Lexical scores are as in equation 2.1. Contextual scores are "backward" 3-cat-based probabilities, in which the current tag is predicted from the two succeeding tags:

$$S_C = \Pr(i_t | i_{t+1} i_{t+2}).$$

2.7.2 Estimation

The Brown corpus was used for training. Probabilities were estimated by relative frequencies, with each observed frequency augmented by one. Modifying the estimator in this way has the property of preserving normalization, as explained in section 2.4.5.

2.7.3 Tagging

Tagging is by best path, with sequences evaluated from right to left. Performance on the Brown corpus was 95–99%.

2.8 Hindle's Tagger

Hindle's tagger [34] is a component of Fidditch, his wide-coverage English parser. It uses statistical methods to acquire a set of symbolic disambiguation rules which are applied deterministically. This type of system is outside the scope of the discussion here, but it is worth noting that results appear to be similar to those obtained by the "pure" statistical approach.

2.9 de Marcken's Tagger

De Marcken's tagger [26] was designed to serve as the front end for a parser. It incorporates a novel method of tagging, which includes the feature, also used in CLAWS, of presenting several alternate tags when no clear disambiguation can be made.

2.9.1 Scoring

The scoring function is identical to that of equation 2.1, except that contextual probabilities are second order.

2.9.2 Estimation

The tagger was trained on the LOB corpus. The estimation method was the same as that used by Church.

2.9.3 Tagging

The standard Viterbi algorithm for finding the best path moves over its input from left to right, one token at a time. It maintains a set of scored partial paths which consists of the best path that passes through each tag for the latest token considered. When the end of the sequence is encountered, it picks the path in this set which has the highest score.

De Marken's tagger uses a variation on this algorithm which dispenses with the backtracking necessitated by delaying path selection until the final token has been seen. For each token, it picks the tag with the highest partial path score. It also picks as alternates any tags which lie on partial paths having scores within some factor of the best score. As de Marcken points out, this algorithm does not necessarily give the same results as the standard Viterbi (ie, find the optimum path), even if no alternate tags are selected. This is because the best partial path at some token will not necessarily remain the best path once all of the input has been seen.

Despite its simplicity, de Marcken's tagging method achieves a 96% success rate on the LOB corpus, which is comparable to that of the much more intricate UCREL

tagger on the same corpus. When an average of one alternate tagging was admitted for about 10% of all tokens, the rate climbed to 98.6%. It should be noted however, that one can achieve arbitrary levels of performance by admitting more and more alternate tags and counting any token whose set of alternates includes the correct tag as a success. A more significant datum is the *efficiency* of the algorithm by which alternates are chosen, as discussed in chapter 6 below.

2.10 Merialdo's Tagger

Merialdo [45] used the most systematic and sophisticated approach of any in the group reviewed. He also tested different options for training and tagging.

2.10.1 Scoring

A 3-cat-based Hidden Markov Model is used to compute scores. HMMs are described in chapter 3 and will not be presented in detail here. The scoring function is the third order analog to that of equation 2.4.

2.10.2 Estimation

Merialdo's tagger was trained on the LOB corpus, modified to use a tag set consisting of 76 tags instead of the original 133. Two types of estimation are considered: relative frequencies modified by interpolation with a uniform distribution; and reestimation.

Reestimation is an iterative algorithm which maximizes the probability which the HMM assigns to the training corpus. It is a maximum likelihood estimator which, in principle, will yield exactly the same results as maximum likelihood estimation from relative frequencies. Its advantage is that, unlike the relative frequency method, it can be used on untagged as well as tagged training corpora.

2.10.3 Tagging

Merialdo's paper describes the use of both path and token-based tagging. The path-based method employs the usual Viterbi algorithm. The token-based method

computes total scores from the joint probability of each tag[5] with that of the entire sequence.

When trained with relative frequencies, the tagger attained a success rate of 97% on the LOB corpus with path-based tagging and a slightly higher rate with token-based tagging. Reestimation over a large untagged corpus was found to improve on the relative frequency estimates from a small tagged corpus, until the size of the latter reached about 5,000 sentences.

2.11 Problems with Existing Taggers

Although the programs summarized above perform well, they are not without deficiencies.

2.11.1 Training

Perhaps the most obvious problem with existing taggers is that, with the exception of Merialdo's, they must be trained from a tagged corpus. Any statistical tagger is limited to the language and tagging scheme of the corpus on which it was trained, and is also—in a sense—optimized for the language variety to which that corpus belongs. Since large tagged corpora are not common and take a long time to produce manually, this limits the usefulness of these programs.

2.11.1.1 Reestimation

Merialdo's method provides a solution to the problem. However, as is demonstrated in his paper, reestimation is a weaker method than training with relative frequencies from a tagged corpus. It is also a much more complex method, both in terms of programming effort and resource requirements.

2.11.1.2 Training From an Untagged Corpus

Other solutions are worth investigating. One very simple idea which does not seem to have been tried is that of using the "naturally" unambiguous words in an untagged

[5]More precisely, the HMM state—the two are not equivalent for a third order HMM.

corpus as a source of relative frequencies for estimation. Although this data can say little about lexical probabilities, their lack might not have too serious an effect.

2.11.1.3 Bootstrap Training

Another solution is suggested by the capability of statistical taggers to identify alternate tags. This is a semi-automatic "bootstrap" method, in which a tagger trained on a small hand-tagged corpus is used to disambiguate a larger corpus and flag questionable tag assignments for manual review. If the tagger is accurate in identifying assignments of which it is uncertain, only a modest human effort would be required to yield a final corpus with a low error rate. It should also be possible to use the procedure recursively to quickly generate very large tagged corpora.

2.11.2 Performance

Although a success rate of 97% seems excellent, there is an alternate measure, suggested by de Marcken, which makes it appear less spectacular. This is the average number of words per error, in which 97% is rendered as about 33. For a human, a mistake every few sentences on a task as easy as tagging would be a dismal record[6]. Furthermore, even this figure is probably inflated, because it reflects tagging performance on the training corpus. Unless the training corpus is superbly representative of the domain (and this is unlikely for the megaword corpora used) one can expect performance to be somewhat lower on other text.

There is clearly a limit to how well statistical taggers of the type considered here can perform. It is not clear what this limit is, nor how it can be achieved. It does not seem, however, that the repertoire of techniques available within the current scope has been exhausted. At least two new techniques can be identified: better estimators for low frequency events, and combined order models.

[6]If this seems discouraging, however, it should be kept in mind that progress has been made: TAGGIT's rate was roughly one error every four words.

2.11.2.1 Low Frequency Estimators

As described in section 2.4.3 above, natural language has the property that, no matter how large a corpus is chosen, certain events will occur with low or zero frequency. There is considerable literature which deals with the problem of estimating probabilities for such events, most of it from the field of speech recognition (see, eg [21, 15, 37, 40, 46, 47, 48]). Two common estimators are *Held Out* and *Good-Turing*, both of which rely on a modification of observed frequencies before they are used in the standard formula (eg, equations 2.2 and 2.3) to compute probabilities. Either of these methods should result in improved performance compared to the more ad hoc ways of avoiding zero probabilities used by the taggers reviewed above.

2.11.2.2 Combined Order Estimators

As the order of a scoring function is increased, the potential accuracy of its predictions increases, but it becomes less reliable. This is due to the fact that for a given training corpus size there are proportionally fewer $(n+1)$-cats represented than n-cats; on average, an estimate made from an $(n+1)$-cat will involve a lower frequency than one made from an n-cat and hence will be less apt to be representative. A solution to this problem is to combine scoring functions of different orders in some appropriate way so that the reliability of the lower order function(s) complements the precision of the higher order functions(s). Two types of combination have been used in speech recognition: linear, in which the final scoring function is a linear combination of functions of different orders [3, 29, 31, 52]; and non-linear, in which some other method is used for combination [15, 40].

2.11.2.3 Other Enhancements

Although outside the scope of this project, many interesting ways of enhancing and extending pure HMMs for improved natural language modelling have been investigated by researchers in speech recognition. Some of these are described in the papers [2, 9, 10, 27, 30, 42, 44].

2.11.3 Error Detection

The results of the de Marcken and UCREL research show that a tagger which produces occasional ambiguous assignments is both viable and useful. The problem of making ambiguous assignments can be seen as part of a more general (and less ambitious) problem of identifying erroneous assignments. A systematic investigation into the ways in which a tagger can extract information from the model about tagging uncertainties seems worthwhile.

Recall that there are two basic methods of tagging, based on the highest scoring path and the path consisting of the highest scoring tag for each token. This suggests that when either tagging method is chosen, the other can be used to corroborate the results. It is also possible to go one step further and use the second best path of each type for additional corroboration.

Another method of identifying uncertainty would be to have the tagger recognize when it is making a decision based on a context or lexical association which has occurred with very low frequency during training. In such cases, its knowledge is probably insufficient to allow it to properly distinguish between the alternatives.

Chapter 3

Methods

The purpose of this thesis is to test the viability of some of the ideas for improving statistical taggers mentioned in the last section of the previous chapter. This chapter sets out the general methods used to do so.

3.1 Software

An experimental system for performing lexical analysis and disambiguation was constructed. The system is language independent and capable of converting raw (unprocessed) input text into a sequence of tagged tokens. It comprises two independent parts: *lex*, a lexical analysis program; and *ytag*, a lexical disambiguation system. This section gives an overview of both parts. Further details are in the appendices: appendix A contains documentation for lex; appendix B contains an account of how lex was customized for French, using two large test corpora; and appendix C contains documentation for ytag.

3.1.1 Lex

Lexical analysis is used here to mean the task of converting unprocessed text into a sequence of tokens (words or punctuation), and assigning to each token a set of tags denoting its permissible grammatical categories. When working with a tagged corpus such as the Brown or LOB, there is no need for lexical analysis, as the corpus is already tokenized, and the set of permissible parts of speech for each token can be

inferred from an examination of the entire corpus. However, one of the objectives of this project was to be able to use untagged corpora for testing, and this necessitated the construction of a lexical analyzer.

By the above definition, lexical analysis consists of two separate problems: tokenization and tag set assignment. Tokenization has not been the subject of much formal investigation but tag set assignment has, especially insofar as it coincides with the problem of morphological analysis (see, eg, [12, 36]).

As an example of what is involved, consider the task of identifying proper nouns. This is a tag set assignment problem but one which also serves to give the flavour of a typical tokenization problem. Many proper nouns will not be found in a dictionary, so recognition must be based on their lexical form. If it is assumed, optimistically, that sentence boundaries can be detected, then proper nouns which do not begin a sentence do not pose too much of a problem, as they are usually capitalized. This is not the case for those which begin a sentence, as of course all such words are customarily capitalized. Those which are not found in a dictionary can reasonably be assumed to be proper nouns, but no decision can be made on those which are, as many proper nouns are homographs. The only solution in this case is to assign multiple tags and thus add to the burden of later stages of analysis. Proper nouns are not the only problem which confronts an analyzer; others include abbreviations, acronyms, numerals, hyphenated forms, expressions, quotations, foreign words, inflected and derived forms, and words which contain or abut punctuation marks.

No attempt was made to model this complexity in a formal way. Lex uses a pragmatic approach which consists of four steps, executed in sequence until a tag set has been found: tokenization, dictionary lookup, token transformation, and guess. To preserve language independence, all application-specific knowledge is read in from external data files. Briefly, the steps are as follows:

tokenization This step identifies the next token from the input. One regular expression is used to match tokens and another to match whitespace; the two are leapfrogged over each other in a standard way. Tags for certain tokens, such as punctuation, are assigned at this point.

dictionary lookup This step attempts to find the token in a dictionary. Provision is made for a dictionary with morphological information: if the initial lookup fails, a set of suffix transformations can be applied to test if the token is an inflected form of some word in the vocabulary.

token transformation This step applies a sequence of more complex transformations to tokens which match any of a corresponding sequence of triggering regular expressions. It is appropriate for capitalized forms and those which contain hyphens, apostrophes, periods, etc. After each transformation, the new form(s) is looked up in the dictionary. Matches can result in token suffixes being pushed back onto the input.

guess The last resort is to guess a set of tags. Different sets may be associated with different tokens, depending on which of a list of regular expressions matches.

Although considerable effort was invested in lex, both in programming and creating data files, lexical analysis is not really central to this thesis. Accordingly, all further details have been relegated to the appendices as mentioned above.

3.1.2 Ytag

Ytag is a lexical disambiguation system. Its only link with lex is that it expects input in the same format that lex uses for its tokenized output. To be precise about the formats of corpora, we adopt the following nomenclature:

raw corpus Unprocessed natural language text.

lexed corpus A corpus in the format written by lex and expected as input by ytag.

tagged corpus A corpus in which each token is labelled with a single nominal grammatical category. The output from ytag is a tagged corpus.

untagged corpus Any corpus which is not a tagged corpus.

It is a simple matter to convert any tagged corpus such as the Brown or LOB into a lexed corpus so that it can be used to test ytag.

Stage	Program	Input File(s)	Output File	Report Pgm
statistics collection	coll	lexed	stats	rstats
estimation	estm	stats	HMM	valhmm
reestimation	reestm	lexed, HMM	HMM	valhmm
tagging	tag	lexed, HMM	tagged	perf

Table 3.1: Ytag stages

Ytag is designed to facilitate experimentation with different ways of attacking the tagging problem within an HMM-based framework. It comprises a collection of separate programs which interact through files (or Unix pipes). The process of estimating a model and using it to tag is divided into four stages: statistics collection, estimation, reestimation and tagging. With each stage there is associated an intermediate file, a program to generate it and another program to report on its contents. Each generating program has switches which allow different modes to be selected. The stages are summarized in table 3.1.

Two noteworthy characteristics of ytag are its segmented design and the fact that it is based on an explicit formal HMM.

A segmented design has several advantages. First, it is easy to analyze the effects of using different modes for each stage, because the contents of intermediate files may be examined with the reporting programs. Second, each stage can be tested in isolation without the necessity of re-running all previous stages. Finally, memory requirements are lower than they would be for a monolithic program; this can be a significant consideration when working with very large corpora.

The central module in Ytag implements operations on an abstract HMM data type. This is made explicit in that the intermediate file written by the estimation and reestimation programs, and expected as input by the tagging program, is an encoded HMM.

The main reason for adopting HMMs is practical. Code is cleaner and algorithms, particularly complex ones such as reestimation, are easier to program. Scoring functions of different orders can be elegantly accommodated by mapping them to the same underlying HMM implementation. Finally, the general purpose HMM

module is reusable. This would facilitate experimentation with scoring functions not considered here such as, for example, ones based on the last n content words rather that the last n-cat.

There are also theoretical advantages. Unlike most of the more ad hoc methods discussed in the previous chapter, HMMs are a complete, albeit weak, model of the way language is produced. This means that the scores assigned to various language phenomena are true probability estimates. It also means that probabilities are available for more phenomena—for example, any sequence of words—than those usually considered to be directly relevant to the tagging problem. The strengths and weaknesses of a model are easier to assess when its underlying assumptions are apparent.

3.2 Hidden Markov Model Theory

Due to the importance of HMMs to this thesis, it is worthwhile to give a formal definition and a description of the main algorithms at this point. The presentation which follows is of an abstract HMM; the application of HMMs to the tagging problem is discussed in the following section.

3.2.1 Definition

We view a Markov model[1] as a source which generates a sequence of T output symbols, $O = o_1 \ldots o_T$, by entering a sequence (or path) of states, $I = i_1 \ldots i_T$, and producing one symbol per state entered. Formally, an HMM is a tuple (V, Q, Φ, A, B), where:

$V = \{v_1, \ldots, v_M\}$ is a set of output symbols

$Q = \{q_1, \ldots, q_N\}$ is a set of states

$\Phi = \{\phi_i,\ i = 1, \ldots, N\}$ is a set of initial state probabilities: $\phi_i = \Pr(i_1 = q_i)$

$A = \{a_{ij},\ i, j = 1, \ldots, N\}$ is a set of state transition probabilities: $a_{ij} = \Pr(i_{t+1} = q_j | i_t = q_i)$

[1]The viewpoint and much of the notation are those of Rabiner and Juang [50]

$B = \{b_{ij}, i = 1, \ldots, N, j = 1, \ldots, M\}$ is a set of output probabilities: $b_{ij} = \Pr(o_t = v_j | i_t = q_i)$

The source begins in some state $i_1 = q_i$ with probability ϕ_i. If it is in state $i_t = q_j$ at any time t, it will make a transition to a next state $i_{t+1} = q_k$ at time $t+1$ with probability a_{jk}, and produce symbol $o_{t+1} = v_l$ with probability b_{kl}[2]. Note the key Markov properties: the probability of the transition to a next state and that of producing the current symbol depend solely on the current state.

The source is called hidden because, although the string of output symbols which it produces is observable, the corresponding sequence of states is, in general, not. A non-hidden source is one in which each state generates some symbol with probability one.

We adopt the term *ergodic* to refer, loosely, to an HMM in which most transition and output probabilities are non-zero. A *non-ergodic* model is one in which a significant proportion of either distribution *is* zero.

3.2.2 Symbol Sequence Probability

The probability, $\Pr(O)$, assigned to any symbol sequence O by an HMM is the sum of the probabilities with which it produces O by each possible state path:

$$\Pr(O) = \sum_{\text{all } I} \Pr(I, O) \tag{3.1}$$

where

$$\begin{aligned} \Pr(I, O) &= \Pr(O|I) \Pr(I) \\ &= \prod_{t=1}^{T} b_{i_t o_t} \times \phi_{i_1} \prod_{t=1}^{T-1} a_{i_t i_{t+1}} \end{aligned}$$

Since the number of paths is exponential in T, the use of equation 3.1 to compute symbol sequence probabilities is not practical except for very short sequences.

An alternate algorithm takes advantage of the fact that at any time t, all paths

[2]Occasionally, we will abuse this notation to let the i's and o's stand for state and symbol *indices*, so that we write, for example: ϕ_{i_t}, $a_{i_t i_{t+1}}$, and $b_{i_t o_t}$. No distinction between states and state indices is necessary.

must pass through one of N states. It is based on the recursion:

$$\begin{aligned}\alpha_{tj} &= \Pr(i_t = q_j, o_1 \ldots o_t) \\ &= \begin{cases} b_{jo_1}\phi_j, & t = 1 \\ b_{jo_t}\sum_{i=1}^{N} a_{ij}\alpha_{(t-1)i}, & 1 < t \leq T \end{cases}\end{aligned}$$

so that

$$\Pr(O) = \sum_{j=1}^{N} \alpha_{Tj}. \tag{3.2}$$

An inductive proof demonstrates that the recursive expression for α_{tj} equals $\Pr(i_t = q_j, o_1 \ldots o_t)$. The base case is obvious. For t we have:

$$\begin{aligned}\alpha_{tj} &= b_{jo_t}\textstyle\sum_{i=1}^{N} a_{ij}\Pr(i_{t-1} = q_i, o_1 \ldots o_{t-1}) \text{ (induction hypothesis)} \\ &= b_{jo_t}\textstyle\sum_{i=1}^{N} \Pr(i_t = q_j | i_{t-1} = q_i, o_1 \ldots o_{t-1})\Pr(i_{t-1} = q_i, o_1 \ldots o_{t-1}) \\ &= b_{jo_t}\textstyle\sum_{i=1}^{N} \Pr(i_{t-1} = q_i, i_t = q_j, o_1 \ldots o_{t-1}) \text{ (Bayes' Law)} \\ &= \textstyle\sum_{i=1}^{N} \Pr(o_t | i_{t-1} = q_i, i_t = q_j, o_1 \ldots o_{t-1})\Pr(i_{t-1} = q_i, i_t = q_j, o_1 \ldots o_{t-1}) \\ &= \Pr(i_t = q_j, o_1 \ldots o_t)\end{aligned}$$

In the second and fourth lines we have made use of the Markov properties described in the previous section. Equation 3.2 allows sequence probabilities to be computed in $O(TN^2)$ time.

3.2.3 State and State Transition Probabilities

It is often useful to know the probability that an HMM will be in a particular state q_i at time t, or the probability that a transition between two states, q_i and q_j, will occur at t. We use the following notation for these events:

$$\gamma_{ti} = \Pr(i_t = q_i, O)$$

and

$$\delta_{tij} = \Pr(i_t = q_i, i_{t+1} = q_j, O).$$

Since

$$\gamma_{ti} = \sum_{j=1}^{N} \delta_{tij}$$

we concentrate on finding a method of computing δ. From the definition:

$$\begin{aligned}
\delta_{tij} &= \Pr(i_t = q_i, i_{t+1} = q_j, o_1 \ldots o_t, o_{t+1} \ldots o_T) \\
&= \Pr(i_t = q_i, i_{t+1} = q_j, o_1 \ldots o_t) \Pr(o_{t+1} \ldots o_T | i_t = q_i, i_{t+1} = q_j, o_1 \ldots o_t) \\
&= \Pr(i_t = q_i, i_{t+1} = q_j, o_1 \ldots o_t) \Pr(o_{t+1} \ldots o_T | i_{t+1} = q_j) \\
&= \Pr(i_t = q_i, o_1 \ldots o_t) \Pr(i_{t+1} = q_j | i_t = q_i) \Pr(o_{t+1} \ldots o_T | i_{t+1} = q_j)
\end{aligned}$$

If we define

$$\beta_{tj} = \Pr(o_t \ldots o_T | q_j)$$

then we can write

$$\delta_{tij} = \alpha_{ti} a_{ij} \beta_{(t+1)j}$$

Since we have an algorithm for computing α from the previous section, it only remains to specify one for β. This is the following "backwards" recursion:

$$\beta_{tj} = \begin{cases} b_{jo_T}, & t = T \\ b_{jo_t} \sum_{k=1}^{N} a_{jk} \beta_{(t+1)k}, & 1 \geq t < T \end{cases}$$

The proof is analogous to that given for the α recursion.

Thus, δ_{tij} (and hence γ_{ti}) can be computed in $O(TN^2)$ time by making a single pass over O: from the beginning of the sequence to t to compute α_{ti}; and from the end of the sequence to $t+1$ to compute $\beta_{(t+1)j}$.

3.2.4 Finding the Most Likely Path

The path I for which $\Pr(I, O)$ is a maximum is also of interest. It can be found using the Viterbi algorithm [53], which operates on the same basic principle as the algorithm for computing the symbol sequence probability given above. The idea is to keep track of the most likely partial paths which end in each state at a given time t, and extend these to the most likely partial paths into each state at $t+1$.

Formally, let V_{tj} be the probability of the highest probability path which ends in the jth state at time t, and σ_{tj} be the state at time $t-1$ on this path. Then

$$V_{tj} = \begin{cases} b_{jo_1} \phi_j, & t = 1 \\ b_{jo_t} \max_{1 \leq i \leq N} (a_{ij} V_{(t-1)i}), & 1 < t \leq T \end{cases}$$

and

$$\sigma_{tj} = \arg \max_{1 \leq i \leq N} (a_{ij} V_{(t-1)i}), 1 < t \leq T$$

Once V and σ have been computed for all t, the most likely path can be found by backtracking over σ:

$$i_t = \begin{cases} \arg\max_{1 \leq j \leq N} V_{Tj}, & t = T \\ \sigma_{(t+1)i_{t+1}}, & 1 \leq t < T \end{cases}$$

The proof that V has the required properties proceeds by induction on t. The base case is obvious, since there is only one path through each state at $t = 1$. The induction step is also straightforward. The most likely path through state q_j at t must come from some state q_i at $t - 1$. By the induction hypothesis, this path has probability $V_{(t-1)i}$ at $t - 1$, and hence probability

$$V_{(t-1)i} a_{ij} b_{jo_t}$$

at t. Clearly, q_i is the state for which this product is a maximum.

Proof of the backtracking step is trivial: it merely finds the path of highest probability at the end of the sequence, then backtracks along the σ's to retrieve the states on this path.

The time complexity of the Viterbi algorithm is $O(TN^2)$.

3.2.5 Reestimation

Before using an HMM, it is necessary to specify values for its parameters—the initial, transition and output distributions Φ, A and B. The usual way to do this is to infer them from some symbol sequence which is representative of the domain to be modelled. This procedure is called training the model; the symbol sequence used is called a training set.

For some modelling problems, parameters can be estimated directly from relative frequencies in the training set using the maximum likelihood estimator described in section 2.4.3 or some other method. This is the case with all problems which give rise to non-hidden models. In general, however, direct methods cannot be used because states are hidden and the frequencies of state transitions and state/symbol associations are therefore not available.

A key result is the reestimation algorithm due to Baum [5], which provides a way to estimate parameters for hidden state models. Reestimation is an iterative algorithm which is guaranteed to converge to a set of parameters for which the probability assigned by the HMM to a given training set is a local maximum.

Two factors affect the success of this technique. The first is the starting point: since the maximum located by the algorithm is local, it obviously depends on the initial set of parameters. The degree of dependence varies inversely with the number of constraints placed on the model by the domain. If there are no constraints then the model is fully ergodic and the maximum located by the algorithm is very dependent on the starting point. On the other hand, if there are strong constraints then the model may be completely non-ergodic (ie, one state per symbol) and reestimation will always converge to the same maximum likelihood estimate obtained by the relative frequency method, no matter what starting point is taken.

The other factor which affects the quality of the model obtained by reestimation is how representative (in some appropriate sense) the training set is of the domain to be modelled. This is a concern which is common to all methods which attempt to draw conclusions about a population from a sample.

3.2.5.1 Reestimated Parameters

The reestimated parameters are defined in terms of various probabilities assigned to a training sequence O by the current model:

$$\begin{aligned}
\overline{\phi_i} &= \Pr(i_1 = q_i, O)/\Pr(O) \\
\overline{b_{ij}} &= \textstyle\sum_{t=1, o_t=v_j}^{T} \Pr(i_t = q_i, O) / \sum_{t=1}^{T} \Pr(i_t = q_i, O) \\
\overline{a_{ij}} &= \textstyle\sum_{t=1}^{T-1} \Pr(i_t = q_i, i_{t+1} = q_j, O) / \sum_{t=1}^{T} \Pr(i_t = q_i, O)
\end{aligned}$$

If the parameters of the current model are replaced by these estimates, the new model will yield a higher value for $\Pr(O)$.

Algorithms to compute all of the probabilities in the reestimation equations have been given in previous sections. Replacing the probabilities with the expressions

used previously, we have:

$$\begin{aligned}
\overline{\phi_i} &= \gamma_{1i} / \Pr(O) \\
\overline{b_{ij}} &= \sum_{t=1, o_t=v_j}^{T} \gamma_{ti} / \sum_{t=1}^{T} \gamma_{ti} \\
\overline{a_{ij}} &= \sum_{t=1}^{T-1} \delta_{tij} / \sum_{t=1}^{T} \gamma_{ti}
\end{aligned}$$

Recall that the computations of γ and δ for any state at time t require a forward recursion from the beginning of the sequence to t to compute the α component, and a backward recursion from the end of the sequence to $t+1$ to compute the β component. This suggests that the values of γ and δ for all t required by the reestimation equations can be computed in two passes over the sequence: a forward pass to compute (and store) all values of α, and a backward pass to compute β and the necessary products of α and β. This algorithm allows each reestimation iteration to be completed in $O(TN^2)$ time and $O(N(N+V+T))$ space. It is known as the *FB*, or *Forward-Backward* algorithm.

3.3 Using HMMs for Tagging

When natural language is modelled as a Markov source, output symbols are words and states are based on some attribute of a word sequence which captures context. It is common in speech recognition, for example, to use n-grams as states, an assignment which has the advantage of resulting in a non-hidden model. Another simple—but hidden—model uses n-cats as states. This is the natural choice for the tagging application because it converts the tag assignment problem into one of finding an optimum state path.

The overlap between the notation used in the previous section for an HMM, and that used in section 2.4 to describe statistical tagging accords with an n-cat-based model. V and O are identical in both contexts; I is not, but there is a close relation between state and tag paths. Henceforth, when there is a need to distinguish between the two, we use I for the former and J for the latter. The mapping between Q and C depends on the order of the HMM.

The details of HMM-based tagging break down conveniently into the same categories—scoring, estimation, and tagging—that were used previously to describe

statistical tagging in general.

3.3.1 Scoring

Recall that the scoring function defined in section 2.4.2 was the product of lexical and contextual components.

3.3.1.1 Contextual Scores

Contextual scores for HMM tagging are state transition probabilities. The context for any tag therefore consists of a previous state and a current state. To be consistent with the earlier use of n as the number of tags used for context, we define an n-th order HMM as one whose states are $(n-1)$-cats, since two states will span n tags.

It is possible within this framework to choose any position for the current tag within the current n-cat. This position determines the tag's context, which may be anything from the preceding $(n-1)$-cat to the succeeding $(n-1)$-cat. The effect of context position has not been formally investigated, but Church's tagger provides evidence that it is small. For convenience, ytag uses the preceding $(n-1)$-cat as context.

3.3.1.2 Lexical Scores

Lexical scores are HMM output probabilities. In fact, output probabilities are more general than the lexical scores defined in section 2.4.2 because they use the current *state* to predict the probability of a symbol whereas the latter use only the current *tag*. However, given the fact that tag context is already represented by transition probabilities, it is hard to see that it could be of much additional value for predicting symbols. For this reason, and to reduce the cost of the model, the output probabilities for all states which end in the same tag were mapped to the same value. That is:

$$Pr(v|j_{t-n+2}\ldots j_t) \simeq \Pr(v|c) \text{ whenever } j_t = c$$

for any word v and category c.

3.3.1.3 Complete Scoring Function

The complete scoring function for HMM tagging is:

$$S(i_t, o_t, I) = \Pr(o_t|i_t) \Pr(i_t|i_{t-1}).$$

Bayes' Law and the Markov properties can be applied to transform this into

$$S(i_t, o_t, I) = \Pr(o_t, i_t|i_{t-1})$$

so that the score associated with each state may be seen to be the joint probability of the state with the current output symbol, given the previous state.

3.3.2 Estimation

There are two ways of estimating HMM parameters: from relative frequencies as described in section 2.4.3, or by reestimation. Only reestimation can normally be used on an untagged corpus, since there is no straightforward way to collect relative frequencies from ambiguous words.

A point not covered previously is the estimation of HMM initial probabilities, $\Pr(qi = i_1)$, for each state q_i. These are supposed to indicate when a state is likely to begin a path. If the corpus to be tagged is segmented as described in section 2.4.4 however, paths will begin in quite arbitrary circumstances. It is difficult to know how to estimate initial probabilities in this case, but absolute probabilities can be used as good approximations. This means interpreting each ϕ_i as $\Pr(q_i)$ instead of $\Pr(q_i = i_1)$. This is the approach taken by ytag.

3.3.3 Tagging

Conceptually, tagging a token sequence with an n-cat-based HMM involves enumerating all tag paths which correspond to the sequence, converting these into state paths, choosing an optimum state path, and converting this back into a tag path.

Converting a tag path into a state path presents a problem. Consider the relationship between some tag path J and the corresponding state path I when $n > 1$:

$$i_t = j_{t-n+2} \ldots j_t \text{ for } t = n-1 \ldots T$$

where $j_t \in J$ and $i_t \in I$, $\forall t$. The problem is that the first $n-2$ states are not uniquely specified by the tag path[3]. It can be dealt with by assuming that any tags are possible for the $n-2$ tokens which precede the first token in the sequence which gave rise to the tag path. This can be expensive however, because it means that each tag path corresponds to L^{n-2} state paths. A better solution is to segment the corpus to be tagged so that each ambiguous token sequence is preceded by a known $(n-2)$-cat which fixes an initial state for each alternate tag path.

The tagging program in ytag goes one step further by segmenting on the basis of unambiguous $(n-1)$-cats. This not only fixes the initial state for each tag path, but implies a single initial state for all alternate tag paths. It has the effect of minimizing the contribution of initial probabilities; these are only used when it is impossible to find an unambiguous $(n-1)$-cat, as is sometimes the case at the beginning of the corpus.

Once state paths are available, the best can be chosen on the basis of path or token information, as described in section 2.4.4. The natural choice for path-based tagging is the path I for which $\Pr(I,O)$ is a maximum; this is directly available via the Viterbi algorithm. The natural choice for token- based tagging is the path I for which $\Pr(i_t,O)$ is a maximum for each $i_t \in I$. Using the FB algorithm, $\gamma_{tj} = \Pr(i_t = q_j, O)$ can be computed for all t and j in the same linear time complexity as the Viterbi algorithm, and it is a simple matter to pick the state of maximum probability at each t.

The final step is to convert the best state path into a tag path. This is easily accomplished, because the tag for each state is just the last tag in the $(n-1)$-cat which corresponds to the state.

3.3.4 Representation

Space is an important concern in HMM-based tagging because of the large matrices required to store probabilities. For an M word and L tag vocabulary, an nth order model has M output symbols and $N = L^{n-1}$ states. A straightforward

[3]This problem is not unique to HMMs: it will attend any tagging method which uses an nth order scoring function.

representation of the model would require L^{n-1} locations for initial probabilities, L^{2n-2} for transition probabilities, and $M \times L^{n-1}$ for output probabilities. Typical values of M and L are 100,000 and 100 respectively, so a third order model would need over 100 million locations in this scheme.

Ytag supports a general non-ergodic representation for an HMM which reduces this requirement in two ways.

First, the transition matrix does not have to be $N \times N$, if some transitions are guaranteed never to occur. This is the case in the tagging application whenever n is greater than 2. A transition between two states is only possible if the last $n-2$ tags in the first state's $(n-1)$-cat coincide with the first $n-2$ tags in the second state's $(n-1)$-cat. In other words, each of the L^{n-1} states can go to only one of L possible next states. The space requirement for the transition matrix is therefore reduced to L^n.

The second economy concerns output probabilities. Rather than storing a probability for each symbol/state combination, ytag stores probabilities for a short list of state *codes* for each symbol. Each code represents a set of potential states, of which a subset is valid in the context of any particular previous state. This representation works well with all of the algorithms described above, which need to iterate down a list of states for a current symbol in the context of a list of states for a previous symbol. Before each such iteration, the list of codes for the current symbol is expanded into a list of valid states. Although it takes time to perform the expansion, this scheme is faster overall if the subset of states which are valid in any one context is small, as is the case with the tagging application.

Given the restriction, described above, that output probabilities depend only on tags, it is possible to use tags as state codes for the tagging application. Expanding the list of codes into a list of states amounts to enumerating the $(n-1)$-cats which begin with the last $n-2$ tags of each previous state and end in each current alternate tag. The space requirement for the output matrix is $M \times L_v$ in this scheme, where L_v is the average number of tags per word. This is significantly less than $M \times L^{n-1}$, since L_v is usually much less than L. It has the added benefit of being order-independent.

When full use is made of ytag's non-ergodic features, an nth order HMM can be stored in $O(L^n + M)$ space. For the third order HMM of the example above, if L_v is assumed to be ≤ 10, this means a drop of two orders of magnitude, from 100 million to about one million locations.

3.4 Testing

The validity of each estimation or tagging method under consideration was tested empirically by measuring tagging performance. For this, the ideal measure would of course be some index of tagging potential which was independent of any particular training and test corpora. Unfortunately, no such index is known, nor is it likely, given the profusion of tagging schemes and the diversity of corpora, that the formulation of one would be simple.

In the absence of a universal measure, an effort has been made to ensure that the results cited below are at least somewhat independent of the test apparatus. The emphasis is on relative results: if one technique is found to be significantly better than another, it is assumed that the difference will generalize to any domain, even though absolute performance may vary.

A standard set of corpora and test metrics were used for evaluation.

3.4.1 Measurement

The usual measure of tagging performance is success rate (SR): the proportion of correctly tagged tokens. The problem with SR is that it gives no indication of how much work was actually performed by the tagger. A success rate of 99% is not very impressive, for example, if 95% of tokens were unambiguous to begin with. While this example is extreme, there is enough variety in both corpora and tagging schemes to render SR performance figures imprecise at best.

Most factors which affect the difficulty of a tagging task are not easy to measure. The exception is ambiguity: in some sense, the higher the ambiguity of a corpus, the harder it is to tag. This observation motivates the definition of an alternate measure of performance, *chance gain* (CG), as the ratio of a tagger's performance

on the ambiguous words in a corpus to the performance expected if tags were picked at random. Formally:

$$\mathrm{CG} = \frac{\mathrm{SR}_a}{T_a / \sum_{t=1, L_{o_t}>1}^{T} L_{o_t}}$$

where L_{o_t} is the number of tags for token o_t, SR_a is the success rate on the ambiguous portion of the corpus, and T_a is the number of ambiguous tokens. Because tagging difficulty is not solely a function of ambiguity, CG is quite crude as a universal measure of performance, but it is nonetheless useful as a complement to SR.

SR and CG are the standard measures of performance in the chapters which follow.

3.4.2 Corpora

In keeping with the goal of reducing dependence on a single corpus, three corpora were used for testing: the French version of the Canadian Hansard (Hans); the LOB corpus (LOB); and a version of the LOB with a simplified tag set (LOB/s). A lexed version of Hans was created with the program lex and a French morphological dictionary; lexed versions of the other two corpora were prepared by making a list for each word of all categories assigned to the word throughout the corpus. As Merialdo has pointed out, it is possible that this introduces a favourable bias by omitting some categories which a dictionary would indicate as valid for a word, but which happened never to occur in the LOB in conjunction with that word.

The three corpora differ in their categorization schemes. LOB uses fine categories which include attributes such as person and number; the other two have coarse categories which are mostly limited to part of speech. The category sets are listed in appendix D.

A one million token training set and a smaller test set were extracted from separate portions of each corpus. For the two LOB corpora, the test set consisted of ten 10,000 token segments selected at random from the original corpus[4]. No tagged version of Hans is available, so a small 5,000 word test set was hand tagged. Table 3.2 summarizes the training sets and gives some pertinent statistics.

[4]It was possible to use a one million token training set with a disjoint 100,000 token test set because the LOB corpus actually contains about 1,115,000 tokens when punctuation is included.

Corpus	Tokens	Ambiguous Tokens	Words	Ambiguous Words	Categories
Hans	1M	313460	23294	5179	34
LOB/s	1M	508622	46065	4359	41
LOB	1M	550369	46065	5357	131

Table 3.2: Training corpora

Seg	LOB		LOB/s	
	SR	CG	SR	CG
1	98.0	3.74	97.8	3.12
2	97.9	3.77	97.6	3.10
3	97.6	3.70	97.3	3.05
4	97.9	3.76	97.6	3.09
5	97.7	3.81	97.5	3.17
6	97.8	3.82	97.5	3.12
7	97.9	3.82	97.8	3.10
8	97.6	3.76	97.5	3.08
9	97.9	3.73	97.7	3.09
10	97.6	3.66	97.3	3.11
test	97.7	3.75	97.4	3.09

Table 3.3: Tagging results for 100,000-token segments from LOB and LOB/s

3.4.3 Significance

As it would probably be meaningless to characterize the tagging difficulty of language with a statistical distribution, no formal analysis of statistical significance was undertaken. To give some idea of the representativeness of a 100,000 word test set for the LOB corpus, ten consecutive 100,000 word segments were extracted and tagged with a simple second order model, trained on the entire corpus. The results for both LOB and LOB/s are presented in table 3.3; the final line in the table gives the results for the segments used for testing.

There are several things to note about these results. First, there is not much

measure	LOB		LOB/s	
	SR	CG	SR	CG
mean	97.8	3.77	97.6	3.10
sdev	.15	.040	.18	.031

Table 3.4: Means and standard deviations for 100,000-token segments from LOB and LOB/s

variation between 100,000 word segments, despite the fact that the LOB corpus comprises text drawn from a number of different domains. This is corroborated by the sample means and standard deviations in table 3.4. Second, the test sets are fairly representative, and all are within approximately one standard deviation of the mean. Third, SR and CG are not strongly correlated, a fact which helps to justify the use of CG. Finally, note that SR scores for the two category sets are similar, although CG scores for LOB/s are consistently lower. This suggests that the extra precision afforded by a fine tag set can overcome the higher ambiguity which it introduces.

For the Hans corpus, as only a single hand-tagged 5,000 word test set was available, no survey of the kind presented in table 3.3 was possible. A rough indication of the representativeness of a 5,000 word corpus may be gotten from table 3.5, which contains tagging results for ten 5,000 word segments selected at random from LOB/s, whose category set is similar to that of Hans. The standard deviations for these data are .47 for SR and .069 for CG.

Seg	LOB/s	
	SR	CG
1	97.2	3.13
2	98.1	3.23
3	96.6	3.97
4	97.2	3.18
5	97.6	3.07
6	97.3	3.09
7	98.0	3.11
8	97.8	3.07
9	97.4	3.10
10	98.0	3.12

Table 3.5: Tagging results for 5,000-token segments from LOB/s

Chapter 4

Training

The requirement to train on a tagged corpus limits the application of most existing statistical taggers to domains for which large tagged corpora are already available. In this chapter, we investigate three ways of circumventing this limitation: by collecting statistics from an untagged corpus; by reestimation; and by semi-automatic bootstrapping from a small tagged corpus.

4.1 Training From an Untagged Corpus

Some words in any corpus are unambiguous. The idea of training from an untagged corpus is to use these words as a "naturally" disambiguated training corpus. This approach is appealing because it is very simple to implement. Its success depends on three assumptions: that enough naturally occurring unambiguous n-cats are available, that these are sufficiently representative of the population to specify a good model, and that doing without lexical probabilities will not have too serious an effect.

The first assumption can be satisfied if a large enough training corpus is available, no matter how small the proportion of unambiguous n-cats may be. Since untagged corpora are abundant, it is safe to assume that this condition can always be met. The remaining two assumptions can be tested empirically.

4.1.1 Implementation

Frequencies of unambiguous n-cats were collected from lexed training corpora in an obvious way: by simply eliminating all ambiguous words and treating each resulting sequence of unambiguous words as if it had occurred in isolation. To avoid normalization problems resulting from disparities between the frequencies of n-cats for different n, transition probabilities were estimated from marginal frequencies, as described in section 2.4.5 above.

The method used to estimate lexical probabilities takes advantage of the knowledge implicit in having a small set of permissible tags for each token. Consider the maximum likelihood formula for estimating lexical probabilities:

$$\Pr(v_i|c_j) \simeq f(v_i, c_j)/f(c_j).$$

In an untagged corpus, the value of the numerator in this expression will be identical for all categories c_j which are valid for word v_i, but this is not necessarily the case for the denominator. Category frequencies can be counted over both ambiguous and unambiguous tokens as:

$$f(c_j) \simeq \sum_{t=1, c_j \in C_{o_t}}^{T} 1/L_{o_t}$$

where C_{o_t} is the tag set for token o_t and L_{o_t} is the size of C_{o_t}. This divides the frequency "mass" of 1 normally assigned to each token equally among the tags in its tag set[1].

This technique was used to estimate different lexical probabilities for each category validly associated with each word. It is interesting to note its implication: if a rare category and a common category are both permissible for some word, the rare category will be considered more probable.

4.1.2 Testing

The method was tested by making a straightforward comparison between a second order tagger trained on an untagged corpus and one trained on a tagged version of

[1] An analogous method could have been devised for transition probabilities, but this would have been more difficult and in any case would just emulate what reestimation accomplishes in a provably optimum way.

Training Corpus	Size	Lex Probs		Trans Probs		All Probs	
		SR	CG	SR	CG	SR	CG
Untgd. Hans	1,000,000	88.8	1.41	93.0	1.69	94.8	1.81
Untgd. LOB/s	1,000,000	72.7	1.56	74.6	1.67	81.2	2.09
Tagged LOB/s	251,283	90.9	2.69	88.9	2.56	97.1	3.07
Untgd. LOB	1,000,000	68.4	1.72	69.4	1.78	75.5	2.20
Tagged LOB	209,705	82.1	2.67	91.0	3.28	97.2	3.71

Table 4.1: Tagging performance with tagged and untagged training corpora

the same corpus. To be fair, the size of the tagged corpus was limited to the effective size of the untagged corpus—the number of unambiguous 2-cats which it contained. The results for each of the standard training/test combinations are given in table 4.1 (the entry for tagged training on Hans is missing because there is no tagged version of the Hans training set). The columns entitled "Lex probs" and "Trans Probs" contain the results obtained using only the indicated set of probabilities when a uniform distribution was assumed for the others.

It is obvious from these data that training on an untagged corpus is a very weak method. Both lexical and contextual probabilities appear to be much less accurate than their counterparts estimated from a tagged corpus. This is not too surprising in the case of lexical probabilities, since one would not expect overall tag frequencies to be a very good indicator of the grammatical behaviour of individual words. It is somewhat more surprising that the contextual estimates are so poor. Clearly, the distribution of unambiguous 2-cats is quite atypical.

It is possible that performance would improve if a more powerful model or a larger training corpus were used. However, preliminary investigations showed that SR actually decreased slightly when a 3-cat-based model was used, probably due to the extreme scarcity of unambiguous 3-cats. Given this, it was not considered worthwhile to experiment with other tagging variations. Likewise, larger corpora were found to make little difference; the curve of performance versus training corpus size seems to be essentially flat above approximately 10^4 tokens.

Another way of improving performance would be to modify the basic method

Training Corpus	Size	Lex Probs		Trans Probs		All Probs	
		SR	CG	SR	CG	SR	CG
LOB/s	1,000	79.9	2.02	85.3	2.35	90.1	2.70
LOB	1,000	77.6	2.31	85.8	2.90	88.3	3.08

Table 4.2: Tagging performance with small tagged training corpora

by trying to identify and correct for the source of bias in unambiguous n-cat frequencies—perhaps an overrepresentation of punctuation marks or an underrepresentation of grammatical words. But the amount of effort involved in such a modification seems hardly justified given the weakness of the foundations on which it would build. An indication of just how weak these are can be gotten by comparing table 4.1 with table 4.2, which contains results for a second order tagger trained on a 1,000 word tagged corpus.

4.2 Reestimation

Recall from section 3.2.5 that reestimation is an iterative algorithm which finds a set of parameters for an HMM so as to maximize the probability, $\Pr(O)$, which it assigns to a training sequence O. The fact that no unique state sequence need be specified for O makes reestimation of interest for the tagging problem.

There is no guarantee that an HMM which maximizes $\Pr(O)$ will be optimal for tagging. However, as mentioned in section 3.2.5, for a fully constrained model in which each symbol sequence completely specifies a corresponding state sequence, reestimation will give the same results as maximum likelihood estimation from relative frequencies. The model we use for tagging is not *fully* constrained, but it is quite strongly constrained because the set of permissible categories for each word is small compared to the total number of categories. The hope is therefore that category constraints will guide convergence to a set of parameters not too far from the ideal of direct estimation from a tagged corpus.

4.2.1 Implementation

As the implementation of the reestimation algorithm is not completely straightforward, it is worth describing some of the salient features.

Recall from section 3.2.5 that the reestimation parameters are:

$$\begin{aligned}
\bar{\phi}_i &= \Pr(i_1 = q_i, O)/\Pr(O) \\
\bar{b}_{ij} &= \textstyle\sum_{t=1,o_t=v_j}^{T} \Pr(i_t = q_i, O) / \sum_{t=1}^{T} \Pr(i_t = q_i, O) \\
\bar{a}_{ij} &= \textstyle\sum_{t=1}^{T-1} \Pr(i_t = q_i, i_{t+1} = q_j, O) / \sum_{t=1}^{T} \Pr(i_t = q_i, O).
\end{aligned}$$

4.2.1.1 Initial Probabilities

The comments made in section 3.3.2 concerning the estimation of initial probabilities pertain to reestimation as well. Basing reestimated initial probabilities solely on the first token in a training corpus as in the formula above means that only those states corresponding to tags which are valid for the first token will have non-zero initial probabilities. To improve on this very arbitrary estimate, the absolute probability of each state was used instead. The reestimate for initial probabilities becomes:

$$\bar{\phi}_i = \sum_{t=1}^{T} \Pr(i_t = q_i, O)/\Pr(O)$$

4.2.1.2 Sequence Buffering

Using this version of initial probabilities, the reestimated parameters expressed in terms of the γ and δ functions defined in section 3.2.3 are:

$$\begin{aligned}
\bar{\phi}_i &= \textstyle\sum_{t=1}^{T} \gamma_{ti} / \Pr(O) \\
\bar{b}_{ij} &= \textstyle\sum_{t=1,o_t=v_j}^{T} \gamma_{ti} / \sum_{t=1}^{T} \gamma_{ti} \cdot \\
\bar{a}_{ij} &= \textstyle\sum_{t=1}^{T-1} \delta_{tij} / \sum_{t=1}^{T} \gamma_{ti}
\end{aligned}$$

With the FB algorithm, the sums over δ and γ in these expressions may be computed in two passes over the training sequence: a forward pass to compute and store values of the α function; and a backward pass to compute values of the β function, take products of α and β to get γ and δ, and sum γ and δ.

In practice, space limitations often preclude storage of all the α values for long training sequences. Ytag detects when this will be the case and compensates automatically. It finds the number of α values which *can* be stored and divides the

training sequence into segments of this length, then makes a forward pass over the sequence to compute and store seed values of α at the beginning of each segment. When the end of a new segment is encountered during the backward pass, the α values for that segment are computed from its seed value and stored. Since each α value must be computed twice, this method takes more time than the FB algorithm, but it remains in the same time complexity class.

4.2.1.3 Scaling

Another practical detail concerns the computation of α and β. Recall that these functions are computed with the recursions:

$$\alpha_{tj} = \begin{cases} b_{jo_1}\phi_j, & t = 1 \\ b_{jo_t}\sum_{i=1}^{N} a_{ij}\alpha_{(t-1)i}, & 1 < t \leq T \end{cases}$$

and

$$\beta_{tj} = \begin{cases} b_{jo_T}, & t = T \\ b_{jo_t}\sum_{k=1}^{N} a_{jk}\beta_{(t+1)k}, & 1 \leq t < T \end{cases}.$$

For a sequence of length T, the values of α_{Tj} and β_{1j} are proportional to p^T, where $p < 1$, so clearly some modification to the above formulas is required to prevent underflow for any sizeable T. Ytag uses the following scaled versions:

$$\hat{\alpha}_{tj} = \begin{cases} s_1 b_{jo_1}\phi_j, & t = 1 \\ s_t b_{jo_t}\sum_{i=1}^{N} a_{ij}\hat{\alpha}_{(t-1)i}, & 1 < t \leq T \end{cases}$$

and

$$\hat{\beta}_{tj} = \begin{cases} s_T b_{jo_T}, & t = T \\ s_t b_{jo_t}\sum_{k=1}^{N} a_{jk}\hat{\beta}_{(t+1)k}, & 1 \leq t < T \end{cases}$$

where s_t is a scaling factor whose value for each t is chosen as α is computed in order to control its rate of growth[2]. When computed over the entire training sequence, α and β differ by only a factor of the initial probabilities, so it is possible to use the same s_t for both.

[2]It is theoretically possible to choose a constant scaling factor rather than one which depends on t, but this is difficult in practice; values which are too small or too large will rapidly tend to zero or infinity.

A nice property of $\hat{\alpha}$ and $\hat{\beta}$ is that they can be substituted directly for α and β in the reestimation equations. To demonstrate this, we need to show that:

$$\hat{\alpha}_{tj} = \prod_{u=1}^{t} s_u \alpha_{tj} \tag{4.1}$$

which is easily accomplished by induction on t. For the base case, we have

$$\hat{\alpha}_{1j} = s_1 b_{jo_1} \phi_j = s_1 \alpha_{1j}.$$

And for the inductive step:

$$\begin{aligned}
\hat{\alpha}_{tj} &= s_t b_{jo_t} \textstyle\sum_{i=1}^{N} a_{ij} \hat{\alpha}_{(t-1)i} \\
&= s_t b_{jo_t} \textstyle\sum_{i=1}^{N} a_{ij} \prod_{u=1}^{t-1} s_u \alpha_{(t-1)i} \\
&= \textstyle\prod_{u=1}^{t} s_u b_{jo_t} \sum_{i=1}^{N} a_{ij} \alpha_{(t-1)i} \\
&= \textstyle\prod_{u=1}^{t} s_u \alpha tj.
\end{aligned}$$

Similarly, for β we have:

$$\hat{\beta}_{tj} = \prod_{u=t}^{T} s_u \beta_{tj}. \tag{4.2}$$

Now if $\hat{\delta}$ and $\hat{\gamma}$ are defined as

$$\hat{\delta}_{tij} = \hat{\alpha}_{ti} a_{ij} \hat{\beta}_{(t+1)j}$$

and

$$\hat{\gamma}_{ti} = \sum_{j=1}^{N} \hat{\delta}_{tij}$$

then equations 4.1 and 4.2 allow us to write

$$\hat{\delta}_{tij} = S\delta_{tij}$$

and

$$\hat{\gamma}_{ti} = S\gamma_{ti}$$

where $S = \prod_{u=1}^{T} s_u$. The reestimation formulas with $\hat{\alpha}$ and $\hat{\beta}$ substituted for α and β then become

$$\begin{aligned}
\hat{\phi}_i &= \textstyle\sum_{t=1}^{T} S\gamma_{ti} / \sum_{j=1}^{N} S\alpha_{Tj} \\
&= \bar{\phi}_i \\
\hat{b}_{ij} &= \textstyle\sum_{t=1, o_t=v_j}^{T} S\gamma_{ti} / \sum_{t=1}^{T} S\gamma_{ti} \\
&= \bar{b}_{ij} \\
\hat{a}_{ij} &= \textstyle\sum_{t=1}^{T-1} S\delta_{tij} / \sum_{t=1}^{T} S\gamma_{ti} \\
&= \bar{a}_{ij}
\end{aligned}$$

4.2.1.4 Floor Thresholds

Because reestimation finds a maximum likelihood estimate, the probabilities that it assigns to states, state transitions and state/symbol associations which do not occur in the training corpus will tend to zero as the algorithm converges. Ytag uses a very simple method to avoid this undesirable condition: it maintains a floor threshold value for each set of probabilities—initial, transition and output[3]. If a particular reestimated parameter is not above the applicable floor threshold, then it is not used to replace the old value of that parameter. Preliminary experimentation with different floor threshold values indicated that zero works best. All of the results reported below were generated with floor thresholds set to zero for all three sets of probabilities.

4.2.1.5 Convergence Measurements

The obvious point at which to stop iterating is when the tagging performance of the reestimated HMM reaches a peak. Since it is not always possible or convenient to assess tagging performance *a priori* however, two other measures of convergence were considered.

The first is a direct measure, *model error*, which is the average change in parameters between the original and reestimated HMMs. It is defined as the sum over all parameters of the absolute value of the difference in each parameter, divided by the total number of parameters which were changed. Model error should decrease as the algorithm converges.

Another measure is perplexity [4], which is commonly used in speech recognition. Perplexity is inversely proportional to the probability which the HMM assigns to the training sequence, normalized to be independent of the length of the sequence. As such, it is a measure of the poorness of the "fit" between the model and the sequence, and is affected by both the difficulty of the sequence and the quality of the model. If the hypothesis which motivates the use of reestimation for tagging is correct, low perplexity should correlate with high tagging performance.

[3]There are more sophisticated ways of accomplishing this. One is the *deleted interpolation* method of the IBM group [3].

Perplexity is defined as

$$\frac{1}{\sqrt[T]{\Pr(O)}}$$

or, equivalently, in terms of the notation of section 4.2.1.3:

$$e^{(\ln S - \ln \sum_{j=1}^{N} \hat{\alpha}_{Tj})/T}.$$

This is easy to compute because $\hat{\alpha}$ is available during reestimation and $\ln S$ is just $\sum_{t=1}^{T} \ln s_t$.

4.2.2 Testing

Three factors were tested for their effects on the reestimated model: the number of iterations, the model components and the initial model.

4.2.2.1 Iterations

Figure 4.1 shows SR for a 2nd order model plotted against number of iterations for each of the three standard corpora. Reestimation was applied to both lexical and contextual probabilities, beginning with a uniform distribution. The shape of the curve is roughly the same for each test corpus, with peak performance attained after about 12 iterations. It is interesting to note that for the corpus with the most precise tag set—the LOB—performance actually begins to degrade after the peak is reached. This may be a consequence of the fact that reestimation is a maximum likelihood method: as the fit of the model to the training data is improved, idiosyncrasies in that data become more significant. The reason the phenomenon does not occur for the other two corpora may be that their coarser tag sets preclude an exact fit to the training data.

Figures 4.2 and 4.3 display the model error and perplexity for the same test sequence. Model error is clearly not a very good indicator of peak tagging performance. Apart from an initial very sharp decrease during the first few iterations, it follows a more or less uniform exponential decay until well past the performance peak. The oscillations in this measure may be a result of using floor thresholds to keep parameters from being driven to zero.

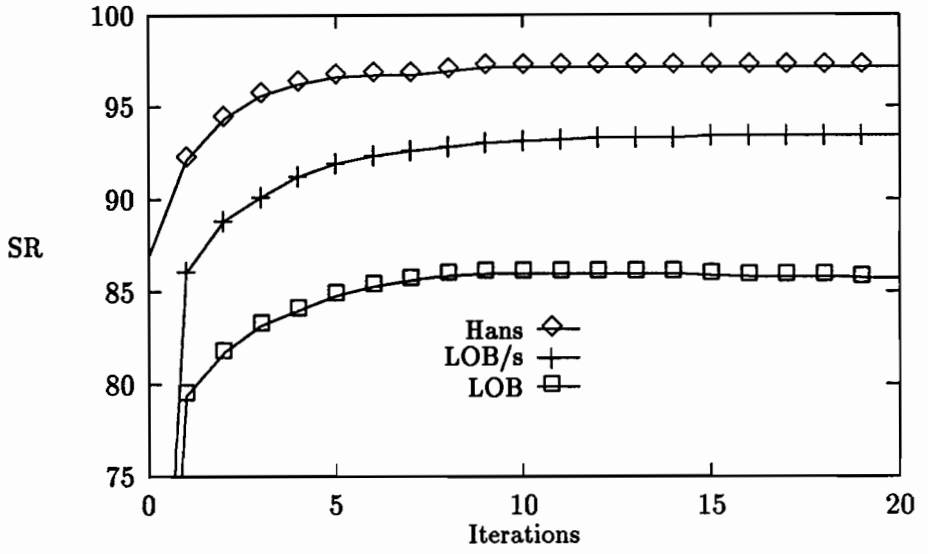

Figure 4.1: Performance of 2nd order reestimated model versus number of iterations. Peak values for SR and CG are: Hans—97.1, 1.96; LOB/s—93.4, 2.84; and LOB—86.0, 2.93.

Perplexity is slightly better at predicting peak performance. By the twelfth iteration, perplexity has essentially reached a minimum for each corpus, although in each case it continues to decrease slowly. (This accords with the above explanation of the performance decay observed for LOB.) The perplexity minima cited in figure 4.3 clearly illustrate the nature of this measure. The minimum is higher for LOB/s than for LOB because, although the token sequence is the same, the finer category set of the LOB makes for a more precise language model. The low perplexity of the Hans corpus is an artifact of the easy token sequence, which contains only half as many word types as the LOB corpora and is drawn from a much more homogeneous source.

A comparison between table 4.2 and figure 4.1 shows that reestimation is a more promising method than relative frequency training on an ambiguous corpus. The reestimated model performed slightly better than the model trained on a tagged 1,000 word corpus for LOB/s and slightly worse for LOB. Given this reasonable start, it is worth investigating some variations to see if the relative performance can be improved.

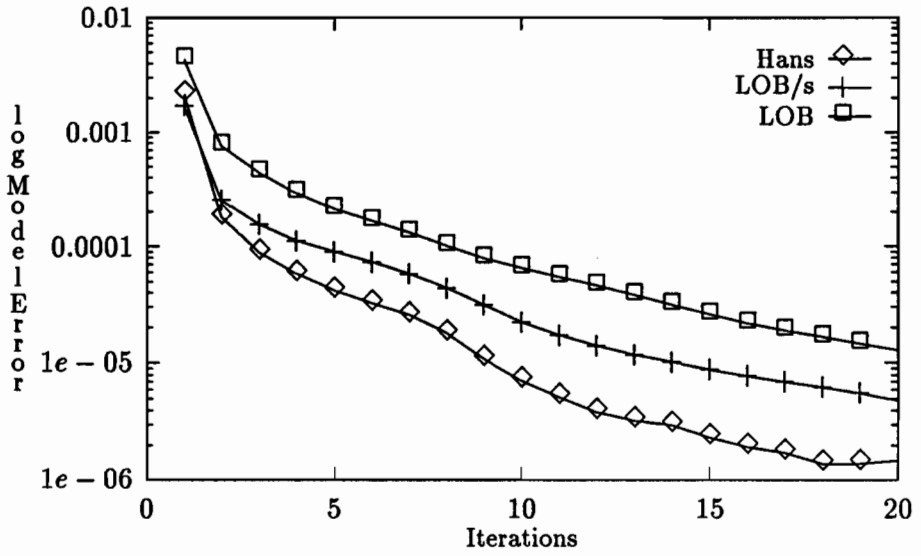

Figure 4.2: Model error for successive iterations

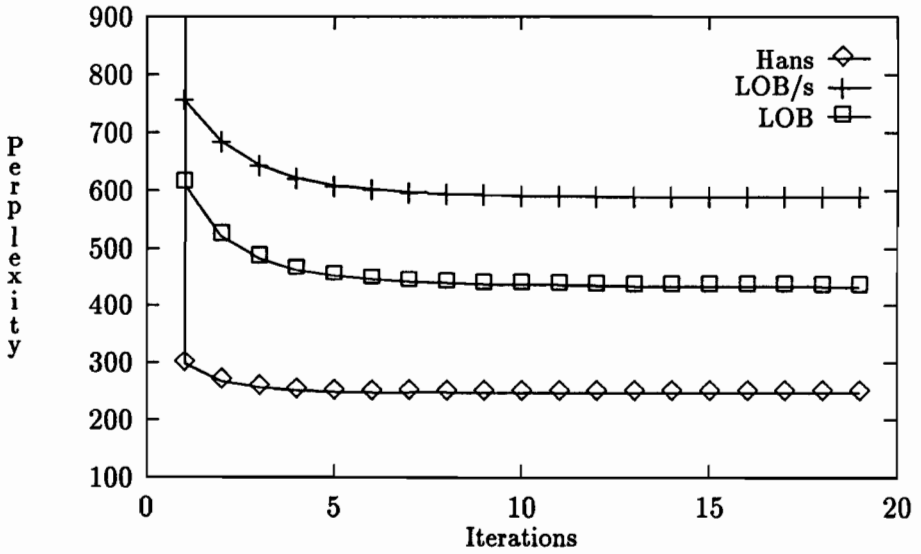

Figure 4.3: Perplexity for successive iterations. Lowest values are: Hans—246; LOB/s—589; and LOB—431.

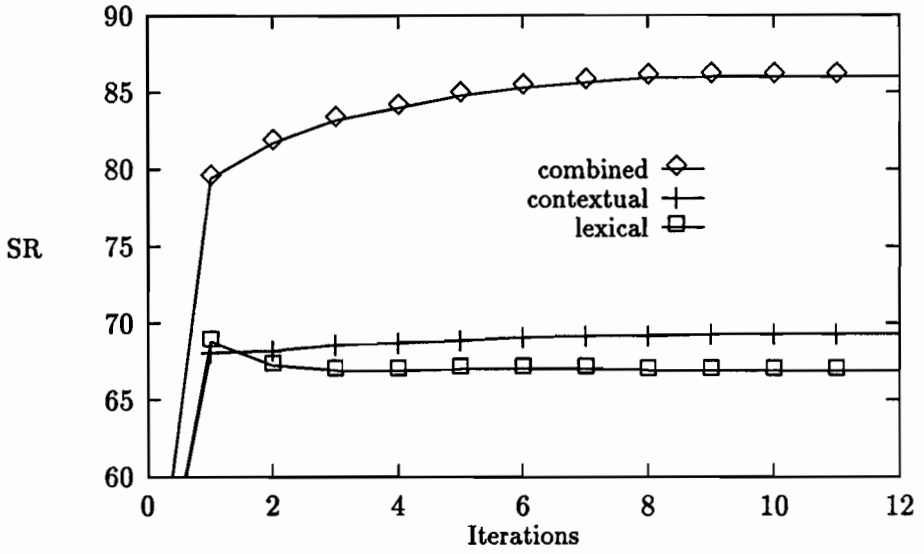

Figure 4.4: Performance of single-component 2nd order reestimated models on the LOB corpus versus number of iterations.

4.2.2.2 Model Components

It is possible to reestimate a subset of HMM parameters while holding the others fixed. This suggests an experiment to determine the contribution of the main model components—contextual and lexical probability sets—to the reestimated model[4]. Figure 4.4 shows the performance of models in which only a single component was reestimated. Compared to the standard combined model, these do very poorly.

The decline in performance for the model based solely on lexical probabilities suggests that reestimation of lexical probabilities might actually be detrimental to the performance of the combined model. This hypothesis was tested by reestimating both components for the first iteration (since the lexical model peaked at that point), then only contextual probabilities thereafter. Although the resulting model did better than either of the single-component models in figure 4.4, it was not as good as the combined model. Apparently there is a synergistic effect from which the combined model benefits.

[4]Initial probabilities are ignored because, in ytag's implementation, they play a negligible role.

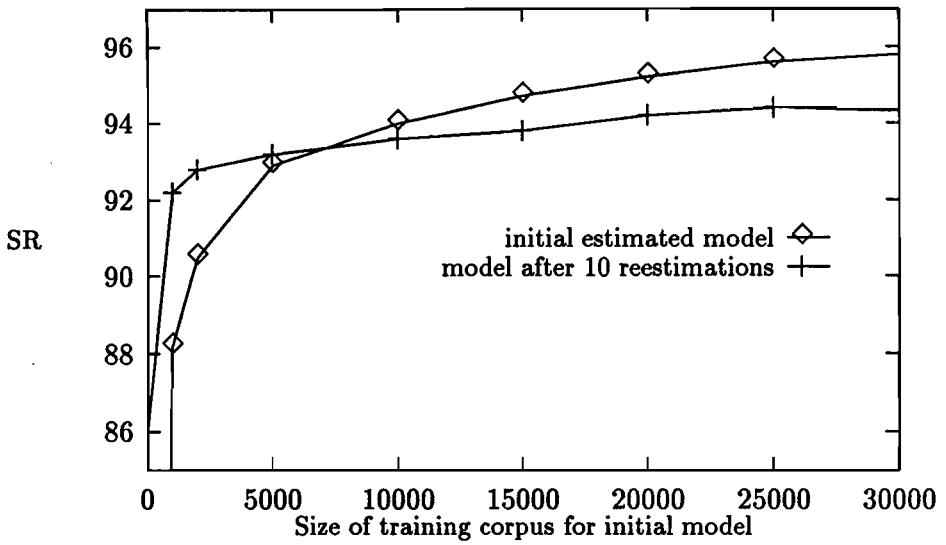

Figure 4.5: Performance of 2nd order reestimated models on the LOB corpus versus size of training corpus used for initial model.

4.2.2.3 Initial Model

Since the maximum $\Pr(O)$ found by reestimation is a local one, it is possible that it depends on the model with which the algorithm begins. For tagging purposes, the most fruitful way to modify the uniform initial model used previously is probably to add knowledge by training on a small tagged corpus. Figure 4.5 shows the effects of such a change on reestimated models after 10 iterations.

It is evident that there is some dependence on starting point: an initial model trained on a 1,000 word corpus, for example, produced a reestimated HMM which achieved an SR of 92.2% on the LOB, compared to 86.0% when the initial model was uniform. However, this rate of gain does not persist with increasing training corpus size; the performance curve quickly flattens after about 4,000 words.

Interestingly, perplexity seems to be quite independent of the initial model in this experiment. Since perplexity is a fairly direct measure of $\Pr(O)$, this suggests either that each initial model directs convergence to a separate local maximum or that all converge on a single broad maximum. The former possibility is implausible given the observed stability of the reestimation algorithm; if the latter is the case, it emphasizes the weakness of the link between $\Pr(O)$ and tagging performance.

4.2.2.4 Conclusion

None of the variations investigated was able to make reestimation as powerful a training method as direct estimation from relative frequencies over a tagged training corpus. Reestimation is useful to enhance the performance of a directly estimated model, but, as is evident from figure 4.5, only if the training corpus size is less than about 7,000 words. This is essentially the same conclusion reached by Merialdo in [45], although the cutoff point he finds is considerably higher.

4.3 Bootstrap Training

Given the inadequacy of the automatic training methods described in the preceding sections, it appears that any successful training technique must rely, at least in part, on human tagging. In this section we sketch a bootstrapping method whose aim is to make more efficient use of expensive hand-tagged tokens. It should be noted that the work described below is intended as a proof of concept rather than a thorough analysis of the technique.

4.3.1 Method

The standard manual training method involves hand tagging a training corpus from which the parameters of a model can be directly estimated. Bootstrap training is an attempt to improve on the performance of such a model. It consists of three basic steps and can be iterated any number of times:

1. Use an existing model to tag some corpus larger than the one on which it was trained.

2. Manually correct some of the errors in the resulting tagged corpus.

3. Use the corrected corpus to train a new model.

This algorithm depends on having some means of identifying errors which is more efficient than just checking each token. One such is to have the tagger itself flag assignments of which it is uncertain. As described in chapter 6 below, this is quite feasible; by flagging a fairly modest proportion of the tokens in a corpus,

ytag can detect the majority of the errors it has made. The details of how this is accomplished are not pertinent here, except that it is easy to adjust the number of flags and hence the number of tokens which must be hand checked.

Another condition on which the validity of the method depends is that the performance of the bootstrapped model be better than that of a standard model trained on a corpus which contains the same number of tokens as were hand checked during the bootstrapping procedure. This is not a foregone conclusion, because the corrected corpus will contain some errors unless a very high proportion of its tokens were hand checked.

4.3.2 Testing

Parameters to the bootstrapping procedure include the number of iterations, the size of the initial corpus, the amount by which the corpus size is increased on each iteration and the number of tokens which are checked on each iteration. The procedure was tested for only one combination of parameter values: a single iteration, an initial corpus size of 10,000 tokens, and a final corpus size of 100,000 tokens, of which 10,000 were checked. This setup is fairly realistic because 20,000 tokens can be tagged with a modest effort and performance from a 100,000 token training corpus is quite respectable.

The LOB corpus was used to simulate hand tagging and checking. An initial model was generated from a tagged 10,000 token segment and used to tag an untagged 100,000 token segment (disjoint from the smaller segment), and flag 10% of it. Each of the 10,000 flagged tokens in the resulting corpus was then assigned the correct tag by comparing it to the original tagged LOB. The corrected 100,000 token corpus was then used to generate a final model. The performances of both the initial and final models were tested on the standard test corpora. The results for LOB and LOB/s are shown in table 4.3.

The first two lines in this table give the performances of the initial and final bootstrapped models; the last two lines give the performances of standard models of the indicated sizes. Bootstrapping is clearly worthwhile for both corpora, but the results are especially good for the LOB. The performance gain achieved by

Model	LOB		LOB/s	
	SR	CG	SR	CG
initial (10k)	95.4	3.59	94.5	2.91
final (100k)	96.9	3.69	95.8	2.99
std (20k)	96.1	3.64	95.6	2.98
std (100k)	97.2	3.71	97.0	3.06

Table 4.3: Performances of bootstrapped and non-bootstrapped models

bootstrapping is more than twice the gain achieved by tagging an additional 10,000 tokens for standard training. Moreover, the performance of the final bootstrapped model is within .3% of the performance of a standard model trained on 100,000 tagged tokens, despite the fact that the latter would require five times more tokens to be hand tagged than the former.

Chapter 5

Performance

The work described in the previous chapter suggested that the best method of training an HMM for tagging is direct estimation from a tagged training corpus. In this chapter, we take that as a starting point for the investigation of two enhanced methods of estimation: Good-Turing frequency modification and order combination. Since the aim is to identify the ingredients of an optimum model, two other factors which were largely ignored in the previous chapter—model order and tagging method—are also considered. Models of orders of 2 and 3 are tested with both token-based maximum likelihood (ML) tagging (the method of the previous chapter) and path-based Viterbi (P1) tagging, as described in section 3.3.

5.1 Good-Turing Estimation

As discussed previously, the problem with maximum likelihood estimation for natural language phenomena is that any practical corpus is bound to be small compared to the number of valid constructions admitted by the language. Hence there will always be a significant number of rare events which are predicted poorly, most of which will be assigned a probability of zero because they do not occur in the training corpus.

A general form of solution to this problem relies on modifying the observed frequencies so as to make them more accurate; in particular, zero frequencies are proscribed. The modified frequencies are then used in the standard maximum like-

lihood formula to estimate probabilities.

The frequency modification technique used in the previous chapter and in the tagging example of chapter 2 was the *augmented corpus* (AC) method, in which it is assumed that the training corpus has been augmented by one containing a single occurrence of every possible n-cat and permissible word/tag combination. Individual frequencies and frequency totals are modified in accordance with this assumption. In ytag's version of AC, word occurrences are assumed to be *ambiguous*, and word/tag frequencies are counted using the method for ambiguous corpora described in section 4.1.1. This prevents the augment data from overwhelming the observed data when the vocabulary size is large. Words encountered during tagging which are not in the vocabulary are treated the same as words in the vocabulary which did not occur in the training corpus[1].

A more sophisticated technique uses the Good-Turing (GT) formula, which is a general method for estimating the proportions of binomially distributed types in a mixed population. It was proposed by Good in [32] and has been widely used in language models for speech recognition and elsewhere. In this section, we compare the performances of the GT and AC estimators for n-cat and word/category probabilities.

5.1.1 Implementation

The GT formula defines a modified frequency f^* as

$$f^* = (f+1)N_{f+1}/N_f \tag{5.1}$$

where f is the observed frequency and N_f is the number of types (ie, word/category combinations or n-cats) with frequency f. The probability of any type which occurs with frequency f is then estimated as usual:

$$\Pr = f^*/N$$

[1]Recall that the input to ytag is a lexed corpus, so the permissible tag sets for such words are always known.

where N is the size of the sample:

$$N = \sum_{\text{all } f} f N_f.$$

The main problem in using GT estimation for natural language is that the N_f data are unruly, especially for large f. The set of points labelled "raw N_f" in figure 5.1 shows N_f values plotted against frequency on a log/log scale for word/tag combinations in the Hans corpus; the behaviour for n-cat values is similar. It is obvious from these data that some smoothing must be undertaken. Equation 5.1 is critically dependent on the ratio of adjacent N_f values, and for the raw data these are very erratic and in some cases undefined because N_f is zero.

Ytag uses a two-step procedure to revise the raw data. The first step is to make more reliable estimates for N_f where data are sparse. It is accomplished using a formula borrowed from Church and Gale [15]:

$$N^*_{f_j} = \frac{2N_{f_j}}{f_k - f_i}$$

where f_i, f_j, and f_k are successive frequencies for which the corresponding N_f's are non-zero. This averages each non-zero N_f over the zero intervals which border it; the factor of 2 accounts for the overlap between successive averaging intervals. The main effect of this step is to bring the long tail of high frequency points into line with the rest of the data by decreasing their N_f values below one. This can be seen in figure 5.1, where the revised points are labelled "averaged N_f". Note that the averaging has no effect on points which are not adjacent to a frequency for which N_f is zero.

The second step is to fit a curve to the averaged points. This is somewhat difficult because, as is apparent from figure 5.1, the relationship between f and N_f is something like:

$$N_f = K/f^n$$

which has a very sharp bend near the origin and is not easy to fit with a simple function such as a least-squares polynomial. However, the data can be rendered more tractable by taking the logarithm of both f and N_f at each point. Ytag adopts this ad hoc strategy, then uses a third order least-squares polynomial to fit

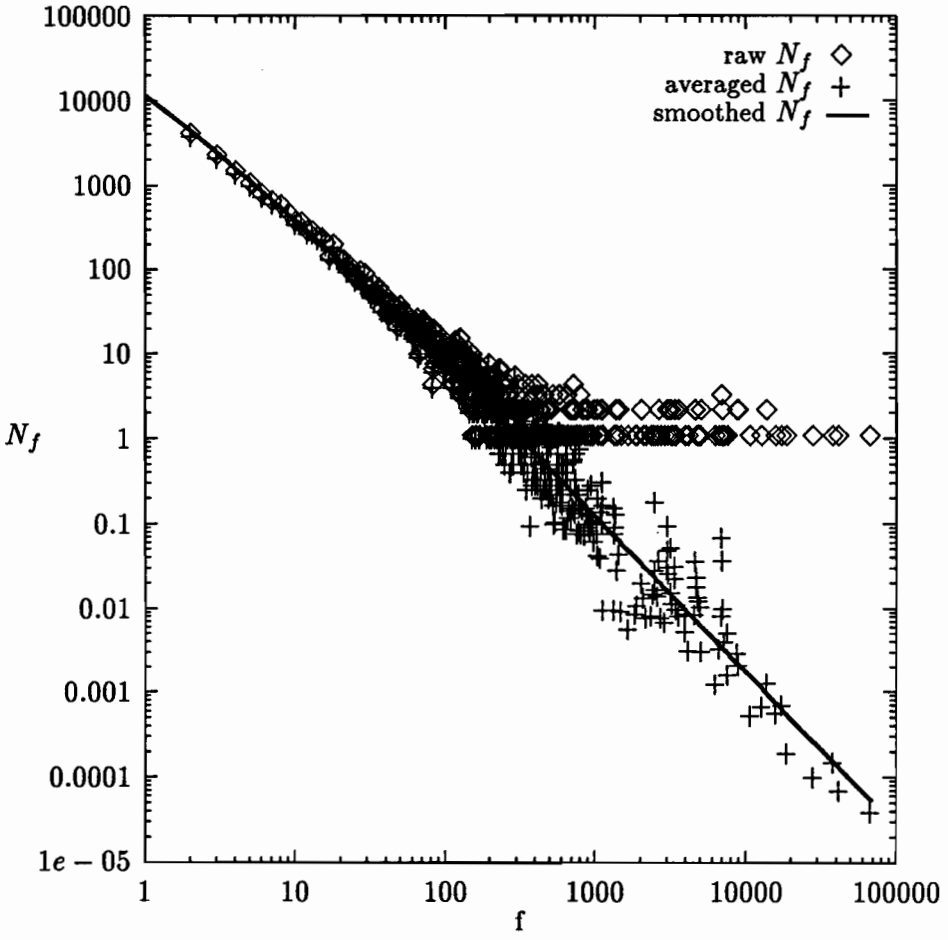

Figure 5.1: Raw, averaged and smoothed N_f versus f for word/tag combinations in the Hans corpus. The smoothed curve is a third order polynomial least squares fit.

Corpus	ML Tagging		P1 Tagging	
	SR	CG	SR	CG
LOB/s	97.298/97.349	3.08/3.09	97.270/97.331	3.08/3.09
LOB	97.501/97.561	3.73/3.74	97.492/97.554	3.73/3.74

Table 5.1: Comparison of tagging performance between AC/GT estimators for 2nd order model.

Corpus	ML Tagging		P1 Tagging	
	SR	CG	SR	CG
LOB/s	97.535/97.634	3.10/3.10	97.557/97.641	3.10/3.10
LOB	97.614/97.725	3.74/3.75	97.608/97.717	3.74/3.75

Table 5.2: Comparison of tagging performance between AC/GT estimators for 3rd order model.

the transformed points. The final smoothed N_f values are the antilogs of the points on this curve. They are shown as "smoothed N_f" in figure 5.1.

5.1.2 Testing

GT estimation was compared to AC estimation using 2nd and 3rd order taggers on the LOB and LOB/s corpora with both maximum likelihood (ML) and best path (P1) tagging methods, as mentioned above. The results for 2nd order tagging are presented in table 5.1, and those for 3rd order tagging in table 5.2. For both of the standard measures of performance, the figures for AC and GT in these tables are given together, separated by a slash, (ie "AC/GT") to facilitate comparison.

The performance improvement resulting from the use of GT is disappointing considering the amount of effort invested. It is only about .05% for 2nd order and .1% for 3rd order taggers; the difference between ML and P1 tagging methods is insignificant. The fact that GT performs relatively better with a 3rd order model is probably due to the fact that 3-cats are rarer than 2-cats. For example, in the LOB corpus the mean number of tokens per 2-cat type is 145 while the mean per

related to that described by Katz in [40]. Although it is applicable to both initial and transition probabilities, ytag uses it only for the latter.

The idea is that no (AC or GT modified) frequency is ever used to make a probability estimate unless it is above some threshold. If an n-cat frequency does not meet this criterion, then the frequency of its rightmost constituent $(n-1)$-cat is tested, and so on recursively to lower orders until the threshold condition is met or only a single category is left. All of the estimates made for any order are scaled so that they sum to the "unused" probability mass from the next higher order. This maintains the requirement that all probabilities sum to one and also means that the role of lower order estimates is merely one of distinguishing between alternatives for which higher order estimates are unreliable; no lower order estimate ever contributes information to a higher order estimate which is considered reliable.

Formally, let

> C be the set of categories, n the order of the model, and f denote frequency as usual;
>
> τ be the threshold frequency; and
>
> $\vec{c} = c_{i_1} \ldots c_{i_{n-1}}$ be an $(n-1)$-cat which corresponds to some state in an nth order model, and $\vec{c}_r = c_{i_{n-r+1}} \ldots c_{i_{n-1}}$ be the rightmost $(r-1)$-cat in $\vec{c}$.

The rth order estimate of the probability of any category c given $\vec{c}$ (equivalently, the probability of a transition from the state corresponding to $c_{i_1} \ldots c_{i_{n-1}}$ to the the state corresponding to $c_{i_2} \ldots c_{i_{n-1}} c$) for all categories $c \in C$ is given by the recursion:

$$\overline{\Pr}^{(r)}(c|\vec{c}) = \begin{cases} \mu_r / S_r \times f(\vec{c}_r c)/f(\vec{c}_r), & f(\vec{c}_r c) \geq \tau \text{ or } r = 1 \\ \overline{\Pr}^{(r-1)}(c|\vec{c}), & \text{else} \end{cases}$$

where μ_r is the probability mass remaining for order r and S_r is the sum of all pure rth order probability estimates. Both are defined in terms of C_r, the set of remaining categories for which an rth order (or lower) estimate is required:

$$C_r = \begin{cases} C, & r = n \\ \{c \in C_{r+1} | f(\vec{c}_{r+1} c) < \tau\}, & 1 \leq r < n \end{cases}$$

So that

$$\mu_r = \begin{cases} 1, & r = n \\ \mu_{r+1} - \sum_{c \in (C_{r+1} - C_r)} \overline{\Pr}^{(r+1)}(c|\vec{c}), & 1 \leq r < n \end{cases}$$

and

$$S_r = \sum_{c \in C_r} f(\vec{c}_r c)/f(\vec{c}_r).$$

The algorithm based on these recursions begins with $r = n$ and stops when C_r is empty or when $r = 1$. Note that when $\tau = 0$, C_{n-1} is always empty, so the computation reduces to pure nth order estimation. Although ytag uses AC or GT modified frequencies for all orders, frequency modification is only strictly necessary for $r = 1$, to prevent these last-recourse estimates from being zero.

More sophisticated versions of the algorithm are possible, in which τ varies with order or even with individual state, but these variations were not implemented in ytag.

5.2.2 Testing

Combined order estimates were tested with a 3rd order GT model using ML and P1 tagging methods on the LOB and LOB/s corpora. A range of different threshold values was used to determine the optimum for each corpus. The results are shown in figure 5.2 for LOB and figure 5.3 for LOB/s.

Order combination is clearly effective only for the LOB corpus, where it improves performance over pure 3rd order GT estimation by slightly over .2%. For LOB/s, the best combined order estimates are exactly the same as the pure 3rd order GT estimates given in table 5.2. As before, the difference between ML and P1 tagging is minuscule, with ML tagging having a slight edge.

The reason for the impotence of order combination on the LOB/s corpus seems to be that a million words is enough to saturate 3-cat frequencies for the simple 41-category set used. That is, if all 3-cats are classed roughly as either grammatical or ungrammatical, then a majority of grammatical 3-cats have appeared much more often than have those in the ungrammatical class. In other words, most 3-cat frequencies are accurate.

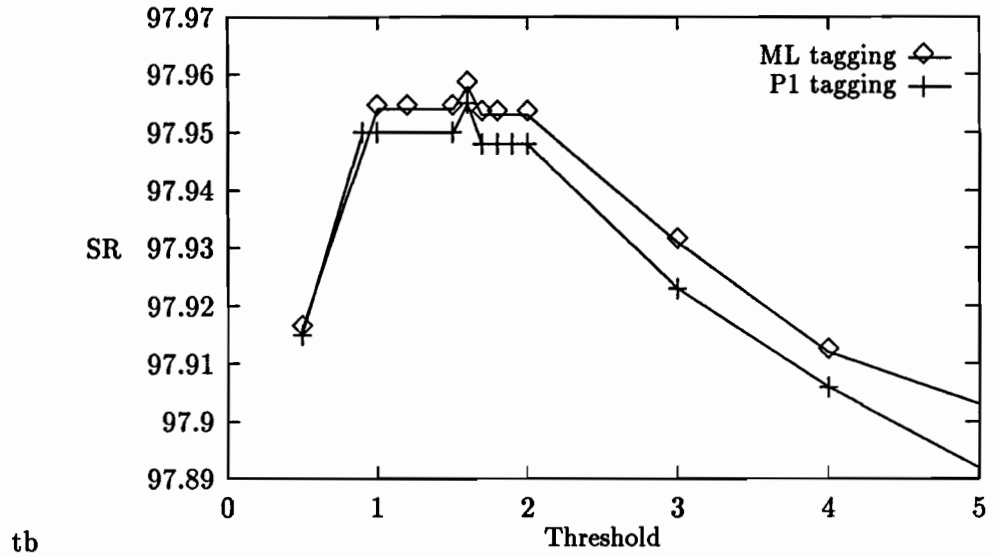

Figure 5.2: Performance of a combined 3rd order GT model on the LOB corpus versus combination threshold frequency. Peak is SR = 97.958 and CG = 3.76 for ML tagging at a threshold of 1.6.

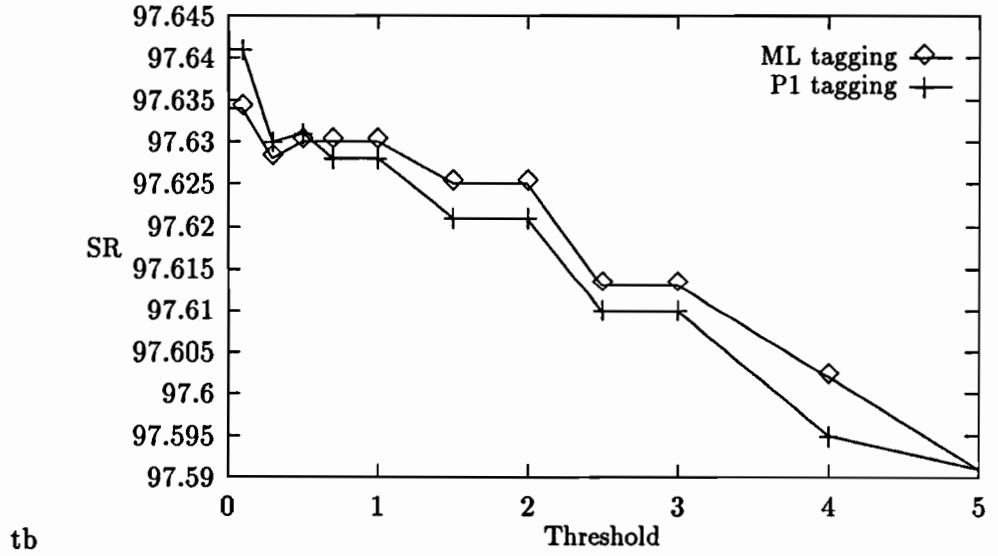

Figure 5.3: Performance of a combined 3rd order GT model on the LOB/s corpus versus combination threshold frequency. Peak is SR = 97.641 and CG = 3.10 for P1 tagging at a threshold of 0.0.

To see why order combination can be detrimental in this situation, consider some low frequency (ie, ungrammatical) 3-cat $c_ic_jc_k$, where

$$f(c_ic_jc_k) < \tau$$

and suppose that the 2-cat c_jc_k is very common:

$$f(c_jc_k) \gg \tau.$$

The combined order estimate based on $c_ic_jc_k$ will then be:

$$\begin{aligned} \Pr(c_k|c_ic_j) &\simeq \Pr(c_k|c_j) \\ &\simeq f(c_jc_k)/f(c_j). \end{aligned}$$

Without making any special assumptions about $f(c_j)$ or $f(c_ic_j)$, one might expect this estimate to be considerably higher than that made from the original 3-cat frequency, ie:

$$f(c_jc_k)/f(c_j) \gg f(c_ic_jc_k)/f(c_ic_j).$$

So that if $f(c_ic_jc_k)$ is accurate, the revised estimate is not.

Some supporting evidence for the saturation hypothesis is that the performance difference between pure 2nd and 3rd order models listed in tables 5.1 and 5.2 is greater by a factor of about 1.8 for the LOB/s corpus than for the LOB. If it is assumed that 3-cat frequencies for the latter are relatively *un*saturated (which is reasonable, given that its category set is almost three times larger) and hence somewhat inaccurate, this is what would be expected. More direct evidence is provided in figure 5.4, which shows number of 3-cat types plotted against corpus size for both training corpora. Although the curve for LOB/s is not quite flat at one million words, both its slope and second derivative are less than those of the LOB curve. Unfortunately, no larger tagged corpus is available on which to test the behaviour of the LOB's category set at saturation.

It can be concluded that order combination is worthwhile for precise categorization schemes and/or small training corpora. However, given an appropriate mechanism for choosing a threshold value, combined order estimates are guaranteed to perform no worse than pure nth order estimates for any corpus and category set.

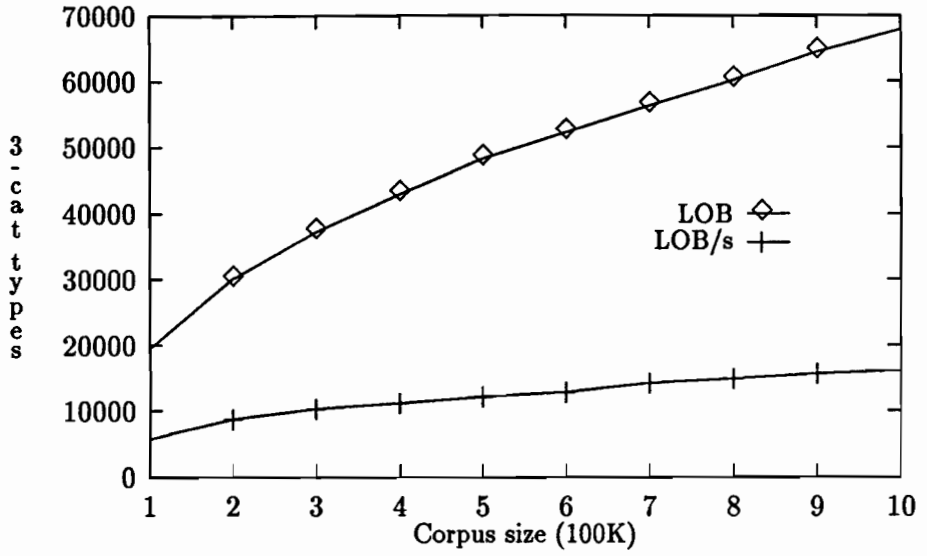

Figure 5.4: Number of 3-cat types versus corpus size for standard corpora.

5.3 Performance Limits

The work described in this chapter and the previous one served to identify an optimum tagger within the current framework as one which is based on a 3rd order HMM trained with combined order Good-Turing estimation on a tagged corpus, and which uses the maximum likelihood path for tagging. When trained on a million word subset of the standard LOB corpus, the tagger was able to successfully disambiguate 98.0% of a representative 100,000 word test corpus disjoint from the training corpus. As it is customary in the computational linguistics literature to cite performance figures when training and test sets are identical, we note that the optimum tagger successfully disambiguated 98.3% of its million word training corpus, with a chance gain of 3.79.

It is interesting to speculate on how close this tagger comes to achieving the best performance of which a pure HMM-based approach is capable. An easy upper bound on SR for an nth order tagger is just the number of disambiguations which require n or fewer tags of context, as the tagger cannot be expected to cope successfully with those which require more. It might be the case that this bound is only dependent on n, and that arbitrary performance levels could be attained by using HMMs of high

enough order. But if the difference between second and third order models observed here is indicative, the orders required to achieve significant gains are likely to be very high. Even under the optimistic assumption that performance varies linearly with order, for example, an extrapolation from the most generous interpretation of this difference predicts that a 7th order model would be needed to achieve a success rate of 99%. Besides being esthetically unappealing, such a huge model would present considerable practical problems. It seems best to concede that HMM-based tagging is most appropriate for those problems which depend on local context and are thus accessible to low-order models. Accordingly, we can limit the scope of this discussion to third order HMMs without too much loss of generality.

For a third order model, the contextual performance bound is the proportion of disambiguations which require three or fewer tags, and this can be established by an analysis of some representative corpus. This may be misleading however, as it sets a human performance standard; there are probably some fundamental inaccuracies in the HMM language model which would prevent a tagger from attaining that standard, even if it used the same amount of context as a human competitor. To get a more realistic bound we can use an alternate approach which consists in identifying the parameters important to the tagging process and extrapolating performance with respect to each. In addition to order, there are at least three such parameters: estimation method, tagging method and training corpus size. The corpus and category set used for testing will also affect the results obtained, but as we have seen, the *relative* results are fairly insensitive to these quantities. That is, what is of interest is the model's performance relative to the best that can be achieved, and it can be hoped that this ratio will be roughly the same for all test corpora and tagging schemes.

It is difficult to quantify the effect of estimation method on performance. However, the scant difference (about .4%) between the most simple-minded and the most sophisticated techniques tested indicates that it is small. It can be expected to become smaller still with increasing training corpus size, as more saturation of 3-cat frequencies occurs and simple maximum likelihood estimates become more accurate. While it is certainly possible to refine the estimation techniques used here, it

seems unlikely that any refinements within the scope of a strict HMM-based model would have significantly greater effect.

The situation is similar for tagging method. The two methods tested yielded remarkably close results, considering that they work quite differently. These algorithms are the only two which are theoretically optimum in any sense. In fact, the only method in the literature which is substantially different from either is that of de Marcken, which sacrifices some information[3] for the sake of speed. As would be expected, de Marcken's results are not quite as good (96% for a 2nd order tagger trained on the LOB and tested on a 64,000 token subset of that corpus) as those achieved by the standard methods (both around 97.5% for a 2nd order tagger with the standard test and training corpora used here).

The final important parameter is training corpus size. Obviously, larger training corpora will give improved performance, up to some eventual limit. In the absence of a tagged training corpus larger than one million words, this limit cannot be determined directly, but performance can be extrapolated from smaller corpora. Figure 5.5 shows SR plotted against training corpora which range from 100,000 to one million words. There is a clear levelling trend, and at about 700,000 words the curve becomes quite flat. The two lower curves in this figure are the performances which resulted when only lexical or contextual information was used. The contextual curve displays the same levelling characteristic as the combined curve but the lexical curve continues to grow, and it is probably this component that will yield most of the performance gains for larger training corpora. However, figure 5.5 demonstrates that the overall performance is quite insensitive to variations in lexical accuracy, so these gains are unlikely to be significant.

To conclude, the tagger described here seems to be quite close to the optimum for a third order HMM-based approach. Although it is impossible to give an exact figure, it would be surprising if changes to the estimation or tagging methods, or increases in training corpus size resulted in SR gains of more than half a percent.

[3] De Marcken's algorithm ignores all tokens to the right of the current token.

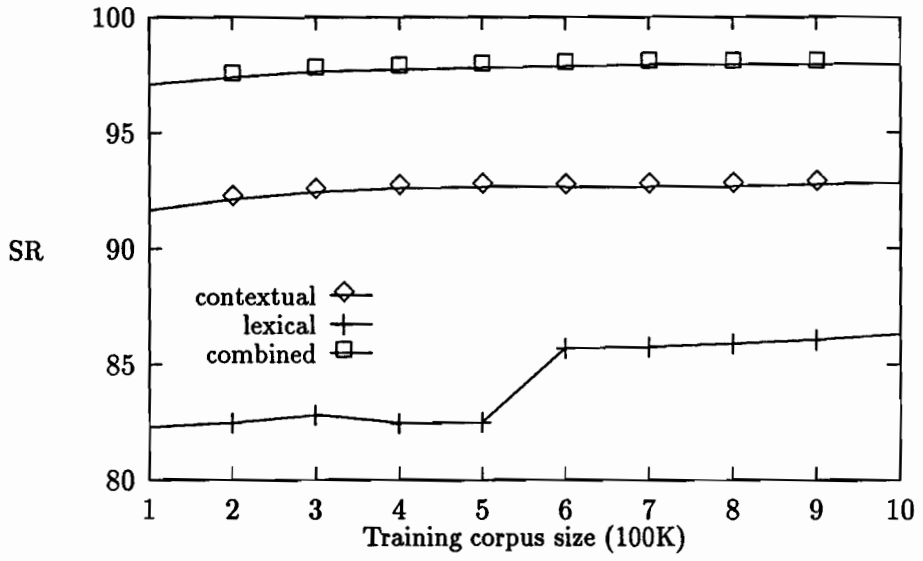

Figure 5.5: Tagging performance versus training corpus size for LOB.

Chapter 6

Error Detection

The probabilities which an HMM-based tagging system associates with tag assignments are a reflection of the confidence with which it makes the assignments. If the model is accurate, assignments which it deems improbable or those for which competing alternatives have similar probabilities should be the ones on which it has a good chance of making an error. This suggests the possibility of having a tagger identify its own potential errors by evaluating assignment probabilities. This chapter describes an error detection mechanism which is based on that idea. The emphasis is on identifying errors rather than proposing alternatives, but the work can easily be extended to include the latter capability.

6.1 Implementation

In the previous chapter we considered methods of tagging based on two state paths, identified as P1—the best, or Viterbi path; and ML—the path of most likely states. An obvious method of error detection is to compare these paths to each other using some criterion and flag as errors the differences established by that criterion. Ytag's error detection mechanism is based on a generalization of that idea.

6.1.1 Error Detection Algorithm

The algorithm uses three (not necessarily distinct) state paths: a *tagging path* (denoted I_{tag}), a *comparison path* (I_{cmp}), and a *replacement path* (I_{rep}). The first step

is to use the tagging path to tag some prefix, O, of the remaining token sequence in the usual way. The comparison path is then used to identify states on the tagging path which are potentially erroneous, and the replacement path is used to furnish alternates for those states. An error flag is raised for each token in O where the replacement path tag differs from the tagging path tag. (An easy extension to this scheme would be to make the replacement path tags available in the output as proposed alternatives to the tagging path tags.)

Crucial to this process is the criterion by which tagging path states are deemded erroneous; ytag uses one of two criteria. *State-based* comparison indicates an error at each state i_{tag_t} where:

$$\Pr(i_{tag_t}, O) / \Pr(i_{cmp_t}, O) < r$$

for some threshold value r. *Path-based* comparison deems all states on I_{tag} erroneous if:

$$\Pr(I_{tag}, O) / \Pr(I_{cmp}, O) < r.$$

So state-based comparison tests the relative likelihood of each state on the tagging path, while path-based comparison tests the relative likelihood (or grammaticality) of the path as a whole. A more general scheme would combine these two criteria in some way, but as will be evident from the test results, a simple method such as comparing states on dubious paths is unlikely to be effective.

Four standard paths are used for tagging, comparison and replacement. In addition to the ML and P1 paths, these include the path of second most likely states (NL) and the second most likely path (P2). The most natural use for NL is to corroborate ML in a state-based comparison, and the most natural use for P2 is to corroborate P1 in a path-based comparison. The four paths are defined formally as:

ML The path $I = i_1 \ldots i_T$ of most likely states $i_t = \arg\max_{j=1,N} \Pr(i_t = q_j, O)$.

NL The path $I = i_1 \ldots i_T$ of 2nd most likely states $i_t = \text{2nd}\arg\max_{j=1,N} \Pr(i_t = q_j, O)$.

P1 The path $I' = \arg\max_{\text{all}\, I} Pr(I, O)$

P2 The path $I' = 2\text{nd}\arg\max_{\text{all}\,I} Pr(I, O)$

The comparison methods outlined above constitute a test for errors which is based on the relative probability of the state or path used for tagging. One way of assessing relative probability is to use the probability of another state or path (such as the one with second highest probability) for comparison; another is to use the total probability of all paths. For both state and path-based comparison, the sum over all paths is just $\Pr(O)$, the total sequence probability. Ytag defines a special "path", TT, which may be selected as the comparison path and which causes $\Pr(O)$ to be used in place of $\Pr(I_{cmp}, O)$ or $\Pr(i_{cmp_t}, O)$ for all t. From Bayes' law, it is apparent that this test is equivalent to comparing the conditional probabilities $\Pr(I_{tag}|O)$ or $\Pr(i_{tag_t}|O)$ to some fixed threshold value.

In addition to TT, there is another special path, PX, which may be used in the role of replacement path, and which is guaranteed to generate a path of tags which differs from the original tag assignment for O at each point. This is useful when it is desired to force each potential error identified during comparison to be flagged as such. For instance, suppose that the tagging path is ML and the comparison path is NL. One might think that all potential errors would be flagged if NL was also used as the replacement path, since NL differs from ML at each token for which more than one state is possible. However, the fact that two states differ does not necessarily mean that the tags to which the states map will differ as well; hence, replacing an ML state by an NL state does not guarantee that an error flag will be raised, as it would if the PX state were used.

6.1.2 Computation

The error detection algorithm involves five paths and two comparison criteria. This section describes how these items were computed. As will be evident from what follows, no item requires more than $O(TN^2)$ time, so this is the complexity of the entire detection algorithm as well.

6.1.2.1 Computing TT

In section 3.2.2, the function α_{tj} was shown to be equal to the joint probability of a given state and the symbol sequence up to time t:

$$\alpha_{tj} = \Pr(i_t = q_j, o_1 \ldots o_t)$$

so the total probability is just:

$$\Pr(O) = \sum_{j=1}^{N} \alpha_{Tj}.$$

The time required to compute α_{Tj} for all j is $O(TN^2)$, so this is the complexity for $\Pr(O)$ as well.

6.1.2.2 Computing ML and NL

Recall from section 3.2.3 that the probability that the HMM will be in state q_i at time t is given by the γ function:

$$\Pr(i_t = q_j, O) = \gamma_{tj}$$

which can be computed with the FB algorithm in $O(TN^2)$ time. Each state i_t on the ML path can thus be obtained by maximizing over γ at t:

$$i_t = \arg \max_{j=1,N} \gamma_{tj}.$$

And the states on the NL path are those which correspond to the second highest γ value. Once γ has been computed for all times t and states j, either of these paths can be found in $O(TN)$ time, so the total time complexity is:

$$O(TN) + O(TN^2) = O(TN^2).$$

6.1.2.3 Computing P1 and P2

The P1 path is identical to the Viterbi path (I for which $\Pr(I, O)$ is a maximum) and may be computed via the Viterbi algorithm in $O(TN^2)$ time as described in section 3.2.4.

The algorithm to find the second most likely path is similar to the Viterbi algorithm and operates in the same time complexity. Recall from section 3.2.4 that

the Viterbi algorithm keeps track of the most likely partial path ending in each state at any given time. To find the second most likely path, we need to keep track of the second most likely partial paths as well. When the end of the symbol sequence is reached, the algorithm backtracks along either the second most likely path which ends in the same state as the most likely path, or the most likely path of highest probability out of those paths which end in any other state.

Formally, let V_{tj} be the probability of the highest probability path which ends in state q_j at time t, and σ_{tj} be the state at time $t-1$ on this path, as in section 3.2.4. Let $\bar{V}_{tj}$ be the probability of the second highest probability path which ends in the state q_j at t, and $\bar{\sigma}_{tj}$ be the state at time $t-1$ on this path. As shown in section 3.2.4:

$$V_{tj} = \begin{cases} b_{jo_1}\phi_j, & t = 1 \\ b_{jo_t} \max_{1 \le i \le N}(a_{ij} V_{(t-1)i}), & 1 < t \le T \end{cases}$$

and

$$\sigma_{tj} = \arg \max_{1 \le i \le N}(a_{ij} V_{(t-1)i}),\ 1 < t \le T.$$

The recursion is more complicated for the second most likely path probability:

$$\bar{V}_{tj} = \begin{cases} 0, & t = 1 \\ b_{jo_t} \max[a_{\sigma_{tj} j} \bar{V}_{(t-1)\sigma_{tj}}, \max_{i, i \ne \sigma_{tj}}(a_{ij} V_{(t-1)i})], & 1 < t \le T \end{cases} \tag{6.1}$$

and

$$\bar{\sigma}_{tj} = \begin{cases} \sigma_{tj} & \text{if } a_{\sigma_{tj} j} \bar{V}_{(t-1)\sigma_{tj}} \ge \max_{i, i \ne \sigma_{tj}}(a_{ij} V_{(t-1)i}) \\ \arg\max_{i, i \ne \sigma_{tj}}(a_{ij} V_{(t-1)i}) & \text{otherwise} \end{cases}$$

The backtracking step is similar to that for the Viterbi algorithm. Recall that the last state in the Viterbi path I is

$$i_T = \arg \max_{1 \le j \le N} V_{Tj}.$$

The last state in the second most likely path $\bar{I}$ is

$$\bar{i}_T = \begin{cases} i_T & \text{if } \bar{V}_{T i_T} \ge \max_{j, j \ne i_T}(V_{Tj}) \\ \arg\max_{j, j \ne i_T}(V_{Tj}) & \text{otherwise} \end{cases}$$

The remainder of the path is obtained by backtracking over the $\bar{\sigma}$'s:

$$\bar{i}_t = \bar{\sigma}_{(t+1)\bar{i}_{t+1}},\ 1 \le t < T.$$

The proof that $\bar{V}_{tj}$ has the required properties (ie, is the probability of the second most likely path through q_j at t) is by induction on t. The base case is obvious; since there is only one path through each state at $t = 1$, the second best path is assigned a probability of 0.

For the induction step, suppose that there is some path which passes through state q_h at time $t-1$ and q_j at t, with probabilities $P_{(t-1)h}$ and $P_{tj} = b_{jo_t} a_{hj} P_{(t-1)h}$ respectively, such that:

$$P_{tj} > \bar{V}_{tj}.$$

From 6.1, we can write:

$$\bar{V}_{tj} \geq b_{jo_t} a_{ij} \times \begin{cases} \bar{V}_{(t-1)i} & \text{if } i = \sigma_{tj} \\ V_{(t-1)i} & \text{else} \end{cases}$$

so that:

$$P_{(t-1)h} > \begin{cases} \bar{V}_{(t-1)h} & \text{if } h = \sigma_{tj} \\ V_{(t-1)h} & \text{else} \end{cases}$$

From the induction hypothesis and the established properties of V_{tj}, this equation can only be satisfied if $h = \sigma_{tj}$, so we have shown that only the most likely path can have a higher probability than $\bar{V}_{tj}$. It remains to note that the two paths are not identical, which is clear from an inspection of 6.1. The proof of the backtracking step is trivial.

6.1.2.4 Computing Comparison Criteria

Path-based comparison requires that $\Pr(I, O)$ be known for both the tagging and comparison paths. For a given sequence of states I, this quantity may be computed directly from the HMM parameters in $O(T)$ time:

$$\begin{aligned} \Pr(I, O) &= \Pr(O|I) \Pr(I) \\ &= \prod_{t=1}^{T} b_{i_t o_t} \times \phi_{i_1} \prod_{t=1}^{T-1} a_{i_t i_{t+1}}. \end{aligned}$$

Given any two paths $I = i_1 \ldots i_T$ and $J = j_1 \ldots j_T$, state-based comparison requires that $\Pr(i_t, O)$ and $\Pr(j_t, O)$ be known for each t. These are available in $O(TN^2)$ time via the γ function and the FB algorithm.

6.2 Testing

Eight error detection methods were tested on the standard LOB corpus, using the optimum tagger described in the previous chapter. Before presenting the results, we describe the method used to measure detection performance and give the rationale for limiting the number of methods considered to eight.

6.2.1 Measurement

Four basic parameters are sufficient to characterize the performance of an error detector on a given corpus:

T The total number of tokens tagged.

E The number of tokens erroneously tagged.

F The number of tokens flagged.

H The number of "hits", that is, valid error flags.

There are several quantities of interest which can be defined in terms of these parameters. The first is the *effectiveness* of the detector—the proportion of actual errors which are flagged:

$$\text{effectiveness} = H/E.$$

Of course, by taking F large enough, one can attain arbitrary levels of effectiveness, and it is no feat to achieve an effectiveness of 100% when F is close to T. This suggests that some notion of *efficiency* is required. One measure is the proportion of error flags which are hits:

$$\text{efficiency} = H/F.$$

Another measure related to efficiency is *cover*—the proportion of flags relative to the total number of tokens:

$$\text{cover} = F/T.$$

Both effectiveness and efficiency can be assessed by comparing them to the values which would be expected if F flags were assigned at random. These values

can be computed from the number of *chance hits*–the expected number of hits from a random flagging procedure for given F, E and T. If it is assumed that each token has probability F/T of being flagged, and an (independent) probability E/T of being an actual error, the probability of a random hit is EF/T^2. If the token probabilities are independent, the number of random hits will follow a binomial distribution, with an expected value of:

$$\text{chance hits} = EF/T.$$

To be completely fair, the random flagging procedure should be limited to the ambiguous tokens in the corpus, so T in the preceding equation should be changed to T_a, the number of ambiguous tokens. This method of calculation was used for the figures cited below.

All of the detection methods described in the first section of this chapter depend on the threshold parameter r. As r is increased, F will also increase, and so, hopefully, will H. The most useful description of a detection method is thus a characterization of its performance over the entire range of r for which H changes. Potential applications for the detector will have different requirements which may correspond to operation within different regions of this range. For instance, a parser may require effectiveness to be 100% regardless of the cover required to attain this level. With human checking, on the other hand, high cover may be impractical, and an effectiveness significantly lower than 100% may be tolerable.

In the results presented below, a standard plot is used to characterize a detector's performance for various values of r. Since r is of no intrinsic interest (and differs from method to method), cover (which varies with r) is used as the independent variable. The dependent variable is effectiveness. The slope of the resulting curve serves as a rough guide to efficiency, because the latter measure is proportional to the ratio of effectiveness to cover. Good performance is therefore reflected in a curve which has a steep slope and which reaches maximum effectiveness well before 100% cover.

6.2.2 Choosing Detection Methods

It is obvious that the number of detection methods admitted by Ytag's framework is impractically high; the field must be winnowed in some way. Fortunately, not all possible path combinations are interesting. For a start, only the two best paths—P1 and ML—need to be considered in the role of tagging path.

Another economy arises from the fact that ML and P1 are very similar. On the standard LOB test corpus, for example, they differed on less than 0.1% of all tokens. This means that their maximum effectiveness is a negligible 5% of the erroneous tokens (which constitute about 2% of the whole corpus). This is not unexpected, but somewhat disappointing because, had the paths differed significantly, their similar (high) tagging success rates would have made them very efficient at mutual error detection. In any case, the impact on the testing process is that the two are interchangeable in the role of tagging path. For the sake of symmetry, ML was used for all state-based comparisons and P1 was used for all path-based comparisons.

A third reduction comes from the fact that, for each method of comparison, only two of the available comparison paths can reasonably be expected to be good indicators—those which represent the second most likely alternative (P2 for path-based and NL for state-based), and the total probability TT. Furthermore, the best of the two can be identified by testing each with the same replacement path, and only the best need be considered with any other replacement paths. This limits the number of detection methods which must be tested to eight; four for each method of comparison.

6.2.3 Results

Path and state-based comparisons are considered separately in the following two sections; the final section compares the two and summarizes results. For brevity, each detection method is identified by a tuple:

$$(comparison, I_{tag}, I_{cmp}, I_{rep})$$

where *comparison* indicates the comparison method: "P" for path-based and "S" for state-based.

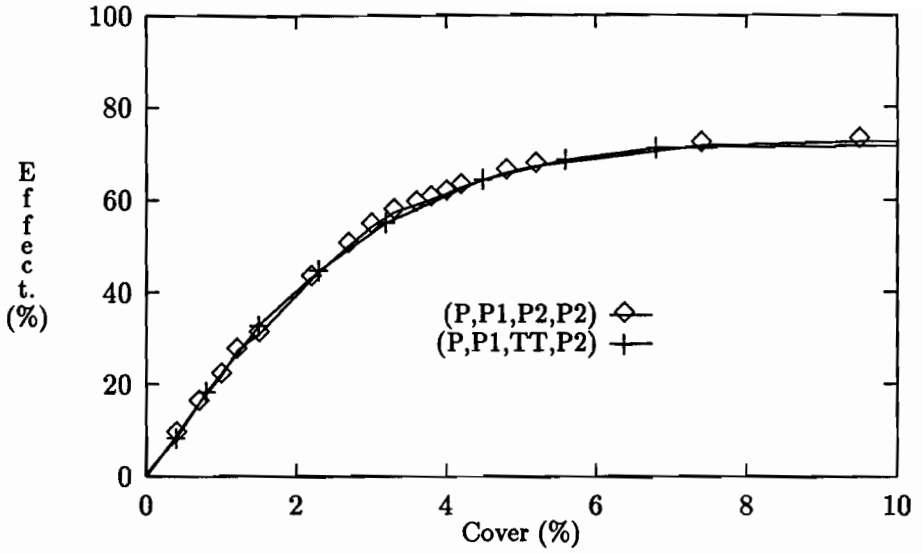

Figure 6.1: Path-based error detection performance with comparison paths P2 and TT.

6.2.3.1 Path Based Comparisons

The first step is to decide whether P2 or TT makes the best comparison path. Figure 6.1 shows the performances of (P,P1,P2,P2) and (P,P1,TT,P2). It is obvious from the plot that the two paths give virtually identical results, with P2 having a very slight, but not statistically significant, advantage. TT can therefore be used for practical purposes, as it is much simpler to compute than P2.

The next step is to compare the possible replacement paths P2, PX and NL. Figure 6.2 shows the performances of these replacements when P2 is used as a comparison path.

Each of the curves follows a similar trajectory: an initial region of high efficiency, then a levelling off as maximum effectiveness is approached. Maximum effectiveness depends on the number of states which differ between P1 and the replacement path; as might be expected, it is lowest for P2 and highest for PX. Unfortunately, efficiency runs in the opposite direction. The best curve is a composite "envelope" of all three which follows (P,P1,P2,P2) up to an effectiveness of 70% and a cover of about 10%, then (P,P1,P2,NL) up to an effectiveness of 90% and a cover of about 30%, and finally (P,P1,P2,PX) up to an effectiveness of 100% and a cover of about 50%. This

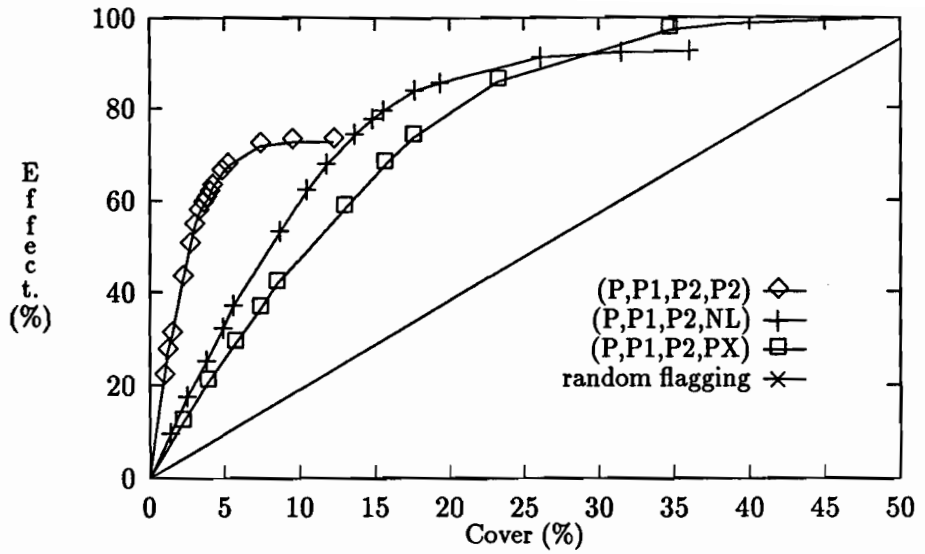

Figure 6.2: Path-based error detection performance with comparison path P2 and replacement paths P2, PX and NL.

last segment is quite poor, as it is barely better than the cover of 52% required by a random flagging process to attain (an expected) 100% effectiveness.

6.2.3.2 State Based Comparisons

The same testing scheme was used for the state-based comparison methods. Figure 6.3 shows the performance for comparison paths NL and TT. As before, the two are very similar; since they are equally easy to compute, either could be used in practice.

Figure 6.4 shows the performances of replacement paths NL, PX and P2, when NL is used as the comparison path. The shape of the curves is similar to those for path-based comparison; the main difference is that the optimum envelope consists of only two segments, due to the fact that the maximum effectiveness of P2 is low (as would be expected, given the similarity of P2 to P1, and P1 to ML). The envelope follows (S,ML,NL,NL) up to an effectiveness of 93% at a cover of about 13%, then (S,ML,NL,PX) up to 100% effectiveness at about 40% cover.

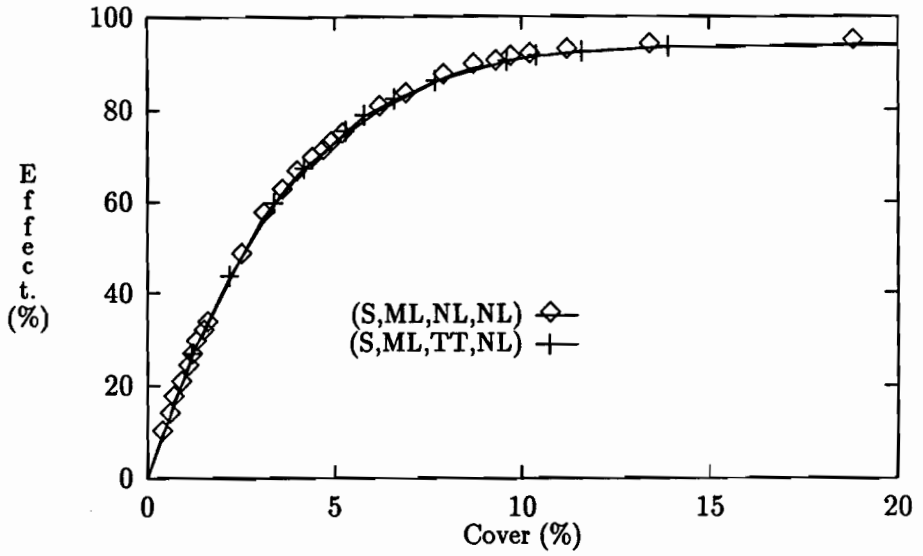

Figure 6.3: State-based error detection performance with comparison paths NL and TT.

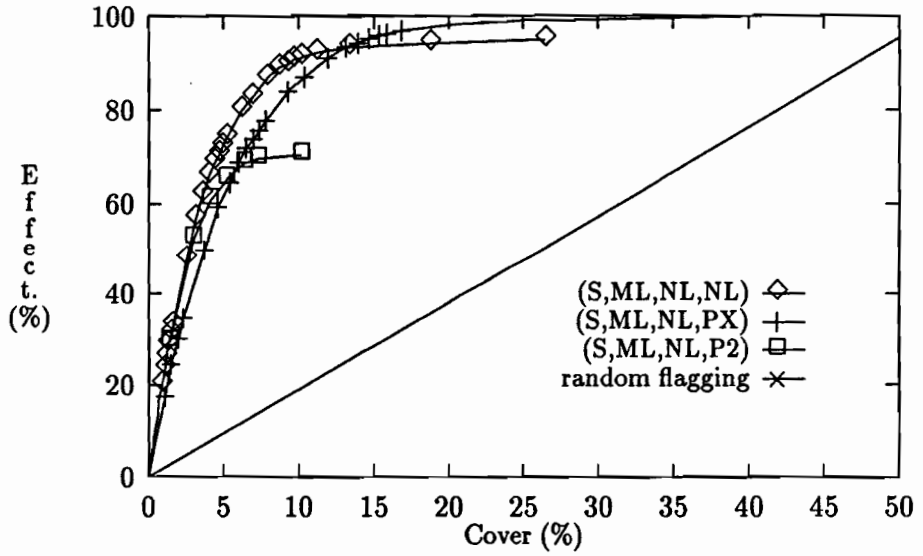

Figure 6.4: State-based error detection performance with comparison path NL and replacement paths NL, PX and P2.

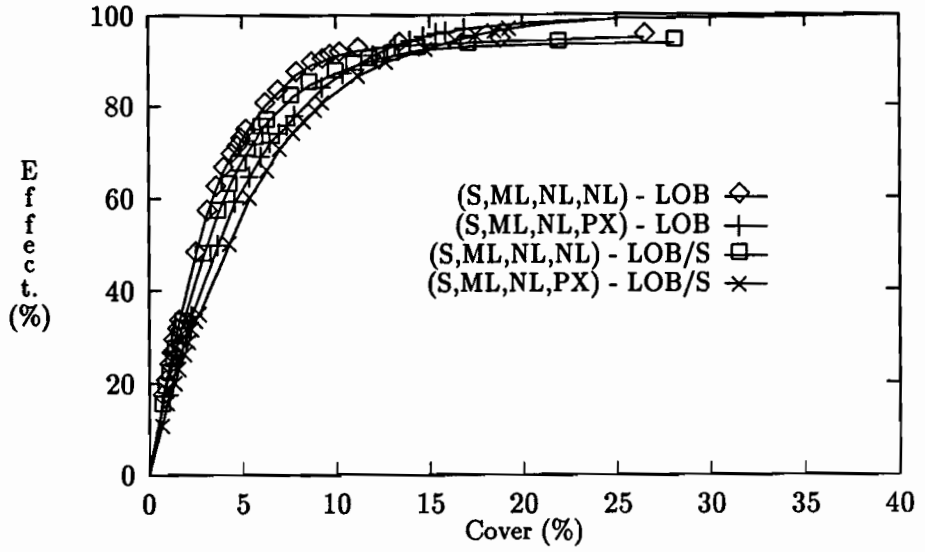

Figure 6.5: State-based error detection on LOB and LOB/S

6.2.3.3 Summary

It is obvious from the data presented in the preceding sections that state-based comparison is superior to path-based comparison; the envelope curve of the former method is everywhere greater than that of the latter, significantly so for effectiveness above 70%. Although this result was derived from testing on the LOB corpus, it can be expected to generalize; figure 6.5 shows that the behaviour of a state-based detector is very similar on the LOB and LOB/S corpora. For practical purposes then, one could use (S,ML,NL,NL) for applications where an effectiveness lower than about 90% is tolerable, and for which small savings in cover are important; otherwise, (S,ML,NL,PX) would be the best method.

The performance curve of the optimum detector can be divided into two regions depending on whether or not it operates significantly better than a random flagging process on the errors which it has yet to detect. At any point along the curve, the number of undetected errors is proportional to the distance between the curve and the horizontal line representing 100% effectiveness. If a random flagging process (limited to previously unflagged ambiguous tokens) were begun at any such point, its performance would be represented by a line between that point and the point on the 100% effectiveness line at which all ambiguous tokens were flagged. Figure 6.6 shows

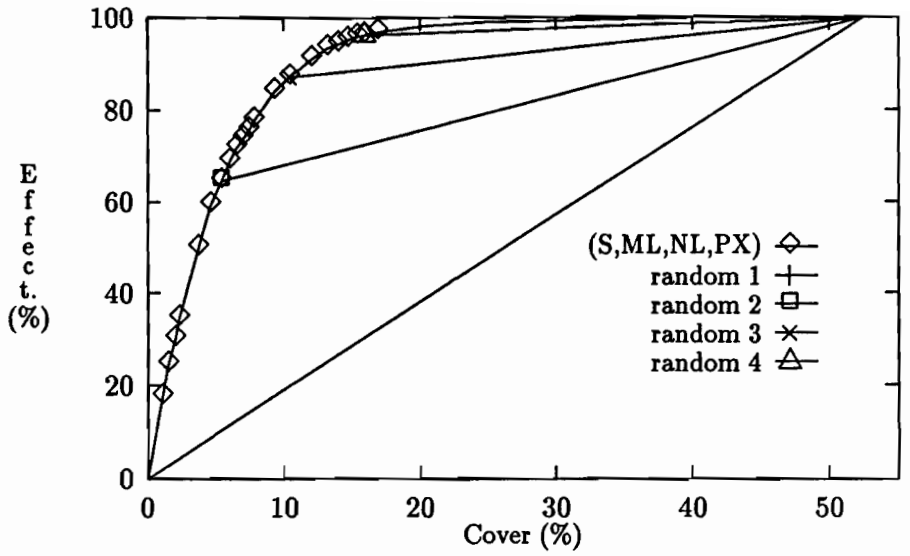

Figure 6.6: State-based error detection compared to random flagging.

several random performance lines plotted at various points along the performance curve for (S,ML,NL,PX). It can be seen that, for the LOB, the boundary between the two regions occurs slightly above 95% effectiveness, as this is the point at which performance approaches that of a random process.

The boundary is distinct enough to admit the hypothesis that each region corresponds to a specific type of error: the first region to *soft* errors, about which the tagging model has some knowledge; and the second to *hard* errors, about which it has none. The model cannot hope do any better than chance on the detection of hard errors; fortunately, these seem to constitute less than 5% of the total error population. 99.9% of all (LOB) tokens will be either correctly tagged or soft errors, and all soft errors can be detected if a flagging level of 15% is accepted.

Chapter 7

Conclusion

This thesis is concerned with improving current statistical solutions to the problem of lexical disambiguation in natural language. Three goals were identified: finding a way to reduce the manual effort required to train an automatic tagger; improving tagging performance, given an optimum training method; and having a tagger detect its own potential errors.

7.1 Methods

Different approaches for each goal were compared experimentally by measuring the performance of an automatic statistical tagging system on a standard set of test corpora.

The tagging system is explicitly based on a Markov model for natural language, in which states correspond to sequences of grammatical categories. Having a clearly defined theoretical basis facilitated the identification and computation of various interesting quantities such as the second most probable sequence of categories for a token sequence.

Three corpora were used for testing: the LOB [38], a version of the LOB with a simplified tag set, and a French Hansard corpus. A one-million-word training segment and a smaller test segment (100,000 words for the LOB; 5,000 words for the Hansard) were extracted from different portions of each corpus. The idea of using three different corpora was to make the results—particularly the relative results, in

which different methods were compared—more general.

7.2 Training

Two ways of eliminating, and one of reducing, the manual effort required to train a tagger were tested.

The first method involves the use of the unambiguous words which occur in any untagged corpus as a "naturally" disambiguated training set. This method proved to be very weak, both because it offers no good way of estimating lexical probabilities, and because the distribution of unambiguous category sequences does not seem to be typical. The performance of a tagger trained on the standard corpora with this method was about 10% lower than that of one trained on a tiny 1,000 word hand-tagged corpus, so attempts at refining the algorithm were not considered worthwhile.

The second method makes use of the reestimation algorithm to find a set of model parameters for which the probability assigned to an untagged training corpus by the model is a local maximum. The effects of three factors—the number of iterations, the lexical and contextual components, and the initial parameter set—were tested to find the model which gave the best tagging performance. The optimum combination was found to be about 12 iterations with a combined-component model, starting from an initial set of parameters estimated (directly) from a tagged corpus of about 2,000 words. The performance of the resulting tagger was reasonable, but still significantly worse than one trained on a 10,000 word hand-tagged corpus.

The final method is a semi-automatic bootstrapping approach in which a tagger is first trained on a small hand-tagged corpus, then used to tag a larger corpus and flag potential errors for manual correction; the corrected corpus can be used to train an improved tagger. If the flagging algorithm is accurate, this procedure can be repeated with larger and larger corpora to generate a sequence of increasingly accurate taggers. Although only a preliminary investigation of this method was undertaken, it shows considerable promise of being able to reduce the number of tokens which must be manually tagged in order to attain a given level of performance.

7.3 Performance

The effort to improve tagging performance centered on two methods of probability estimation which are more powerful than the straightforward use of relative frequencies: Good-Turing and order combination. Both of these techniques have been applied to natural language modelling by researchers in speech recognition, but neither has previously been used specifically for lexical disambiguation. Both methods are intended primarily to address the problem of making accurate estimates for events (category sequences or word/category co-occurrences) which occur rarely or not at all in a training corpus.

Good-Turing estimation modifies the observed frequency of an event—by an amount which depends on the number of other events which occurred with that frequency—prior to using it in the standard relative-frequency formula. It was found to yield a very slight gain of about .1% with a third order model.

Combined order estimation is only applicable to probabilities for category sequences. It uses lower order estimates (ie estimates from shorter sequences) to replace those higher order estimates which are judged unreliable because they depend on frequencies below a certain threshold. When used with an optimum threshold value, this method was found to yield a gain of about .2% over pure Good-Turing estimates for a third order model.

Although the gains from these two estimators were slight, they nevertheless combined to boost performance with a third order model on the LOB to 98.0% for disjoint test and training corpora, and 98.3% for identical test and training corpora. It was argued that these performance figures must be close to the maximum attainable by the type of statistical method considered here. Supporting evidence for this contention was produced by extrapolating performance from other relevant factors: model order, tagging method and training corpus size.

7.4 Error Detection

The final portion of the thesis concerns automatic error detection. A general framework for error detection was described, based on the premise that tagging errors

correspond to situations in which the model assigns similar probabilities to competing alternatives. The framework accommodates several alternate tag sequences which may be compared using either sequence probabilities or individual tag probabilities as criteria.

The best method was found to be a comparison of tag probabilities between the most likely and the second most likely tag for each token. The number of errors detected with this method was found to increase rapidly as a function of the number of tokens flagged until the latter reached about 15% of the test corpus, at which point it levelled off quite abruptly. This behaviour led to the hypothesis that tagging errors fall into two categories: *soft* errors, about which the model has some knowledge, and *hard* errors, about which it has none. By flagging 15% of the LOB test corpus, the tagger was able to ensure that 99.9% of all tokens were either correctly tagged or were flagged soft errors.

7.5 Future research

The most promising area for future research indicated by this work is that of improved error detection and bootstrap training algorithms. It is quite unlikely that significant gains in raw tagging performance can be achieved within the strict framework considered here, but modifications to the framework, particularly those which incorporate longer-distance dependencies, would certainly be worth investigating.

Appendix A

Lex: A Lexical Analyzer for Natural Language

A.1 Introduction

This note describes lex, a lexical analyzer for natural languages. Lex is intended to be language-independent; it provides a basic algorithm which must be augmented by language-specific rules and data read in from external data files. The input to lex is any text file; the output is a listing of the words in the file, one per line, with one or more alternate tags—which may denote grammatical category or any other property defined by the user—assigned to each (note that lex does not attempt to perform tag disambiguation).

A.1.1 Nomenclature

For the present purposes, lexical analysis will be taken to mean the process of converting an input text into a sequence of strings, called tokens (usually words or punctuation), and assigning to each token a set of alternate labels, called tags. Each tag is a free form string which is defined in the data files and is not interpreted in any way by lex.

The nomenclature may be summarized as follows:

token a string from the input, possibly modified in some way

tag a label assigned to a token

tag set a set of alternate tags for a token

A.1.2 Running Lex

Lex requires four data files named *token*, *dict*, *inflect* and *rules* (see section A.3). Although these names are fixed, the location of the files may be varied: if not found in the current directory, they are sought at the paths contained in the *LEXDIR* and *HOME* environment variables, in that order.

The syntax for invoking lex is as follows:

`lex -h`, or

`lex [-v | -x` *data* `| -g` *data*`] [`*text-file* `[`*lex-file* `| -o]]`

where:

text-file is the input file to be analyzed; if omitted, lex reads from standard input and writes to the standard output.

lex-file or **-o** specify that the output be written to *lex-file* or the standard output, respectively; if both are omitted, (and *text-file* was specified) the output is written to *name*`.lex`, where *name* is the base name of *text-file*.

-h produces a help message

-v selects verbose progress reports

-x directs that the data sources specified by *data* be ignored.

data may contain any of the letters T, D, I or R, standing for the token, dict, inflect and rule data files; or the letters t, d, i or r for the corresponding internal sources. The purpose of this option is mainly to allow the compiled-in information in internal sources to be ignored for testing purposes; in the current implementation, this is only relevant for the dictionary.

-g directs that C versions of the specified data files be written, in a format suitable for compiling and linking with lex source code. *data* has the same interpretation as in the previous option, except that no distinction is made between

lower and upper case letters. The C data files are always written to the current directory; each has the same base name as the data file from which it was generated, and an extension of .c. The input file *text-file* may be omitted with this option, in which case lex will stop after having generated the selected C data files. The purpose of having compiled-in data is to avoid the delay caused by reading large external data files. Currently, dict is the only data file for which this capability is provided.

A.2 Algorithm

The program first identifies some prefix of the remaining input as a token, then chooses a tag set for it. It consists of four steps, of which the last three are performed only if the previous step failed to find a tag set.

A.2.1 Token Matching

The first step is to identify the next token. This is accomplished by running two finite automata (DFAs) over the input in succession. The first is used to read through zero or more whitespace characters and locate the next character of interest. The second accepts a regular set which is intended to include all possible tokens; the next token is just that portion of the input which it matches, beginning where the first DFA left off. If no input is matched, then the next token is taken to be all input between the current character (inclusive) and the first character of the first string matched successfully by either of the two DFAs.

Both of these DFAs are generated from regular expressions (REs) read from the *token* data file (see section A.3 for a complete specification). The second DFA identifies each matched token as belonging to one of a set of token types defined in the data file.

Each token type may have an associated tag set which is assigned to any token of that type. Punctuation marks and special forms such as numbers, which can be described succinctly by REs, are good candidates for tagging at this step. Since this is the fastest method of assigning a tag set, certain frequently occurring words

such as articles could also be considered for inclusion in the regular set.

Token types may also have a transform associated with them. Transforms are specifications for transforming strings in standard ways, as described in detail in section A.3. If a transform is associated with a token type, it will be applied to all tokens of that type. The application of a transform at this step is different from applications at later steps in that it is permanent: the original token cannot be recovered after the transformation has taken place. The purpose of this is to allow for the transformation of certain forms which are known not to be recognizable by later stages of the algorithm. For example, words which are split over two lines (ie, with a hyphen) could be joined together, or words containing capitals could be converted to lower case (if the dictionary contains only lowercase words).

A.2.2 Dictionary Look-up

The second step is to look up the token in a dictionary. This takes place in two stages: the first is a straightforward search for the token; the second is a check to see if the token is an inflected form of some word.

The search stage consists of looking up the token in a hash table which contains the entries defined in the *dict* data file (section A.3). If the token is found, it is assigned the tag set associated with the entry.

The inflection stage depends on the set of inflection rules defined in the *inflect* data file, which are interpreted and applied according to lex's model of a word. This is an abstract entity characterized by a set of attributes, each of which may take on a range of values (including a null value). Each vector in this attribute space is denoted by a unique tag; in theory, all combinatorially possible tags could be used. In practice however, the tag set is limited because some attribute values combine only with null values of other attributes (eg when part of speech = noun, tense = null). Generally, a word will have a set of applicable tags, generated by fixing the value of some attribute(s) (usually part of speech) and letting the others vary. There is an inflected form of a word for each applicable tag, but these forms are not necessarily lexically distinct.

The way in which a word inflects is modelled by assigning it an inflection class,

which specifies a mapping between some canonical form—which is stored in the dictionary—and all other possible inflected forms of the word. This mapping removes a fixed suffix from the canonical form and, for each tag (ie, valid combination of attributes), replaces it with another fixed suffix to create the inflected form corresponding to the tag.

Although the model is straightforward when used to describe lexical generation as above, there are multiple ambiguities which affect recognition. First, a token must be tested for matches with a set of suffixes, each of which could be common to many different inflection classes. (Note that this applies to canonical forms as well, since they could be inflected forms of other words.) For each inflection class, the matching suffix must be replaced by the canonical suffix for the class and the resulting form looked up in the dictionary. If it is found, and if the class matches one of the inflection classes listed for the entry (there could be several, since the dictionary form could be common to several different words), then the set of tags associated with the class can be added to that for the current token. The reason that this is a set of tags rather than a single tag is that, for any inflection class, several inflected forms could be lexically identical; hence a single lexical form can have multiple tags.

The only reason for having inflection rules is to save space, since exactly the same set of words could be specified by listing them all explicitly in the dictionary. In fact, this is a tradeoff, since the suffix matching step takes considerably longer than a single look-up. Three measures are used to speed the process somewhat.

First, rules are chained so that only a single suffix match is ever performed on a token. Each rule has a list of rules having suffixes which are contained in its own. The longest possible suffix is always chosen to ensure that all applicable rules are in the list of the matching rule.

The second measure is applied to inflection rules with a null suffix and null replacement strings. These rules perform the function of associating some fixed tag set with every word in an inflection class without having to duplicate the tag set in the dictionary entries for all words in the class. The standard match sequence is redundant in this case, since it will apply to every word. Instead, a table of default

tag sets for each inflection class is maintained. After the first stage look-up, each retrieved inflection class for a word is checked against this table and the token's tag set is augmented with the results.

Finally, tag sets for previously encountered tokens are stored explicitly, so that inflection rules do not have to be used more than once. Since about 98% of all tokens occur more than once in a typical corpus, this saves a significant amount of time. It also adds space, but for a reasonably extensive dictionary and a moderate sized corpus, this will be less space than would be required to store the dictionary in explicit form in the first place.

A.2.3 Transformation

This step attempts to identify a variant form of a token such as one containing capital letters, apostrophes, hyphens, periods, etc, and transform it into a standard form which may be found in the dictionary. It uses a set of transformation rules contained in the *rules* data file, each of which consists of a triggering RE pattern and a token transform.

The rules are applied in the sequence in which they are defined in the data file. Each rule is applied to a list of transformed tokens (to start with, this contains only the original token) created by its predecessors, beginning with the most recent and working backwards. If the rule's triggering RE matches any token in its entirety, it is applied to create a new token. If the new token is found in the dictionary, the search terminates; if not, it is added to the end of the list.

Some rules can split the token into a prefix and a suffix. Although subsequent rules operate only on the prefix, the suffix is retained (and possibly added to by later rules of the same type). If a look-up succeeds on a prefix, the corresponding suffix is pushed back onto the input, to be read again as the next token. This capability should be taken into account when defining token types for step 1: since there is no provision for performing the opposite operation of concatenating successive tokens, an attempt should be made to recognize the longest possible token from the input.

It should be noted that the algorithm for this step has the potential to grow exponentially in the number of rules applied. In practice, this will not happen,

since the list of transformed tokens is limited in length (to 20), but care should nonetheless be taken in the formulation of triggering patterns and the selection of rule order.

A.2.4 Guess

As its name implies, this is the last resort. The original token is matched against a set of RE patterns defined in the *rules* data file. If a match occurs, a corresponding token set is inferred.

If this step does not succeed, lex assigns the error tag "???" to the token. This can, of course, be replaced by any other tag or set of tags by associating the preferred tag(s) with a wild-card pattern which matches all possible tokens.

A.3 Data Files

The program uses four external data files, as mentioned in the previous section. They are:

token defines token types

dict the dictionary

inflect contains inflection rules

rules contains transform and guess rules

A.3.1 Lexical Conventions

All data files share common lexical conventions for data items, which may be simple or compound. Simple items are character strings, delineated by whitespace or surrounded by double quotes. Compound items are lists of items (either simple or compound), delineated by parentheses. A null simple item is a pair of double quotes; a null compound item is a pair of empty parentheses.

Certain characters have special interpretations. They are:

; begin a comment which ends at the next newline

^ interpret any immediately following digits as decimal character codes

(begin a compound item

) end a compound item

" Begin and end a quoted string. To quote a double quote, use its code: ^34.

A.3.2 Syntax

An Extended Backus Normal Form (EBNF) syntax convention, somewhat modified, is used in the descriptions of the data files which follow. The use of meta characters is illustrated in the following table:

form	meaning
a = b	b is the definition of a
[item]	0 or 1 instance of item
{item}	0 or more instances of item
+{item}	1 or more instances of item
item1 | item2	either item1 or item2
lowercase letters	a symbol
UPPERCASE letters	a literal

It should be noted that parentheses () are used as literals throughout.

A.3.3 Token File

This file contains definitions for token types and for the whitespace between tokens. Its syntax is as follows:

token-file = (({re-class-list}) whitespace-def +{token-def})
token-def = (re ({tag}) [transform])
transform= (affix | map | pat-rep)
affix = `A` pfx
map = `M` applic from to
from, to = char-char | char
pat-rep = `P` applic pat rep

applic = `Lft` | `All` | `Rgt`

where:

re-class-list is a character class definition which is active in every subsequent regular expression (RE) in the file; the syntax for class definitions and REs is described in re.doc. To prevent special characters from being intercepted by the data file reader, class definitions and REs should be enclosed in quotes.

whitespace-def is an RE which defines the set of patterns which are to be skipped when searching for tokens, as described in section 2.

token-def is a single token type definition

re is an RE which defines the set of tokens corresponding to the token type as all strings which match a prefix of the input after whitespace has been skipped.

tag is a character string which identifies a tag; it is a single element in the tag set for the token. Any future instance of the same string will be considered to denote the same tag.

transform is a string transformation

affix is a transformation which splits the string into a prefix (the part which is matched by the RE pfx) and a suffix (the remainder of the string).

map performs a character mapping on the string, taking any character in the sequence *from* to the character at the same position in the sequence *to*. These sequences must be of the same length. Applic specifies that the mapping be applied to the leftmost character only, to all characters, or to all but the leftmost character (Lft, All, Rgt respectively).

from, to designate sequences of characters. The form *char-char* designates a range of characters; the form *char* designates a single character. For example, `ad-fz` designates the sequence "a,d,e,f,z".

pat-rep replaces portions of the string which match the pat RE with the string rep; applic specifies that only the leftmost, all, or only the rightmost matching

portions are to be replaced (Lft, All, Rgt respectively). Note that a rightmost transform can have unintuitive results due to the algorithm employed: work backwards over the string, trying to match the pattern (in a forward direction) starting at each character.

A.3.4 Dict File

This file contains the dictionary. Its syntax is as follows:

dict-file = ({entry})
entry = (word infl-class (+{tag}))

where:

entry is an entry for each word

word is a single word form. It may appear in multiple entries.

infl-class is an inflection class label for the word. It may be any string—subsequent occurrences of the same string denote the same inflection class.

tag is a character string which identifies a tag; it is a single element in the tag set for the word. Any future instance of the same string will be considered to denote the same tag.

Note that duplicate entries in this file (ie those whose word, inflection class, and tag set are exactly the same as some previous entry) are harmlessly discarded.

A.3.5 Inflect File

This file contains the set of inflection rules used to match inflected forms of words in the dictionary. Its syntax is as follows:

inflect-file = ({i-rule})
i-rule = (sfx +{replacement})
replacement = (rep-string +{tag-pair})
tag-pair = (infl-class (+{tag})

where:

i-rule is an inflection rule

sfx is a suffix which identifies words to which the rule applies

rep-string is a string which replaces the suffix for the dictionary look-up

infl-class is an inflection class label for the word

tag is one of a set of tags which is inferred for a word if the word matches sfx and the form created by replacing sfx with rep-string is found in the dictionary with an inflection class which matches infl-class

A.3.6 Rules File

This file contains the set of transform and guess rules which are applied to words not found in the dictionary. Its syntax is as follows:

rules-file = (+{re-class-def} ({t-rule}) ({g-rule}))
t-rule = (trigger [prev-cond] transform)
g-rule = (trigger [prev-cond] (+{tag}))
prev-cond= tag | !tag

where:

re-class-def is a character class definition which is active in every subsequent regular expression (RE) in the file; the syntax for class definitions and REs is described in re.doc. To prevent special characters from being intercepted by the data file reader, class definitions and REs should be enclosed in quotes.

t-rule is a transform rule

transform is a transform specification as described in section 3.1

g-rule is a guess rule

trigger in an RE which must match a token to trigger a rule

prev-cond stipulates that a previous tag must match the given tag—or must not match if the *!tag* form is used—in order for the rule to be applied. The test always fails if the previous token had more than one tag.

tag is one of a set of tags which is inferred for the current word if the g-rule is triggered

A.4 Improvements

Speed is the main problem. Two causes for this can be identified. First, it takes time to parse and read in the four data files. Adding the capability of compiling C versions of all data files (currently possible only with the dictionary) would help, although some initialization time would probably still be required (to make entries into hash tables, etc) unless this was done very cleverly. Another improvement would be to allow multiple input files to be analyzed following a single data file read.

Second, the process of applying inflection rules is very inefficient under the current implementation, as each new form requires a separate look-up in the hash table. A better way would perhaps be to store the dictionary as a special trie: once the prefix of a word has been found, a list of alternate suffixes could be quickly tested by working forward from that point in the trie.

Appendix B

Lexical Analysis of French

B.1 Introduction

This is a description of a French lexical analyzer built around lex, a language-independent lexical analysis program, and the Dictionnaire Morphologique du Française (DMF) [35]. The program reads a French text, identifies tokens (words or punctuation marks) and assigns to each token a set of tags. Each tag corresponds to a unique grammatical characterization; in a specific grammatical context, only one of the tags associated with a word is appropriate.

The algorithm used by lex to find a set of tags for a token consists of four steps, executed in sequence until one succeeds: tokenization, dictionary lookup, token transformation and guess. External data files contain language-specific information for each step: *token* for tokenization; *dict* and *inflect* for lookup; and *rules* for transformation and guess. A detailed description of the algorithm and the format of the data files is given in the program documentation for lex.

Most of what follows—sections 2 through 4—is a description of the design of the data files. This was guided by two principles: compatibility with the DMF, which forms the core of the language-specific knowledge in the analyzer; and reliance on a small number of general rules, rather than a large number of specific ones, to describe various lexical phenomena. The analyzer was tested on two large corpora of French text (Hansard transcripts and CRTC hearing proceedings); the final section describes the results.

B.2 Token Data File

The token file enumerates token types, each of which is described by a regular expression. Some token types may be assigned tag sets in this file; tokens of these types are tagged by lex as soon as they are matched. This is appropriate for types comprising a single token or those whose tokens are all expected to display similar grammatical behaviour.

Token types are categorized as words, numeric expressions, or punctuation, based on regular expressions over classes of characters. Four classes partition the ISO character set: letters (upper and lower case and all accented forms), decimal digits, whitespace and special characters (all remaining characters). A fifth class, special punctuation, is a subset of the special character class and consists of the hyphen, period, and apostrophe characters. The categories of token types are described in the following sections, with reference to this system of character classification. It should be noted that the whitespace class is of special significance to lex, which uses it to find the beginning of the next token; this does not however, preclude its inclusion in some tokens.

B.2.1 Words

Most words are defined but not tagged in the token file; with a few exceptions described below, tagging is left to later stages in the algorithm. Since later stages have the ability to split a token if necessary, but not the ability to join successive tokens, it is essential that word definitions in the token file include the longest strings which have the potential of being recognized as words. The following sections identify three main types of words.

B.2.1.1 Standard Words

Words come in a number of variant forms which can make it difficult to distinguish between a word and surrounding punctuation. As it is hard to anticipate every possible valid variant, standard words are modelled by a simple regular expression which is correct in the majority of cases. This defines a word as any string made up

of letters and special punctuation characters which contains at least one letter and does not contain two adjacent special punctuation characters. Thus, *chef-d'oeuvre* is a word, but *chef—d'oeuvre* would initially be tokenized as *chef-*. This example illustrates the main reason for proscribing adjacent special punctuation characters within words: it prevents multi-character punctuation marks such as dash, ellipsis and doubled single quote from being erroneously included in words. In the example, since *chef* exists in the dictionary, the transformation rule stage (see section 4) would push the hyphen back on the input and the final tokenization would be two words separated by a dash: *chef*, —, and *d'oeuvre*.

One possible problem with the model is that it does not allow potentially valid combinations of special punctuation characters such as ".-", which do not correspond to a multi-character punctuation mark, to occur within a word. However, these seem to be very rare in general text (they are non-existent in the Hansard corpus). In any case, none of the nine possible combinations occurs in the DMF, so these would not be recognized without an augmented word list.

Another problem is that any special punctuation character which is contiguous with a word will always be included in its token, whether it was intended to be part of the word or not. This is in keeping with the goal, described above, of always finding the longest possible token which is potentially a word, but it leads to a consistently wrong initial tokenization for some common combinations, such as words followed by a sentence-ending period or surrounded by single quotes. For most strings, this problem—although detrimental to performance—does not result in errors, as the transformation step eventually separates the word from the punctuation. However, a few strings will always be tokenized erroneously. For example, an 'l' would be analyzed as a quote followed by an ell apostrophe since the latter token is in the dictionary, rather than a quoted ell as would probably have been intended. Errors of this kind may be an inevitable consequence of the attempt to provide for forms such as abbreviations which end with a period or words which begin or end with an apostrophe—it is difficult to see how these could be eliminated on the basis of lexical knowledge alone.

B.2.1.2 Broken Words

These are words which are broken over two lines, with a hyphen to mark the break. More precisely, they consist of two standard words as defined in the previous section, separated by any amount of whitespace containing at least one newline character, and of which the first word ends in a hyphen. They are concatenated into a single token, with the intervening whitespace removed. Hyphens are left in because their removal would be irreversible and would cause true hyphenated words to be missed. The transformation stage strips hyphens when required.

B.2.1.3 Expressions

An expression is a sequence of words which can be sometimes be treated as a unit for grammatical purposes. Some special expressions, such as *quant à* or *parce que*, always form such a unit: any tags which are normally assigned to their constituent words are always irrelevant within these expressions. Although general expressions require more complex treatment, members of this special class can be tagged exactly as if they were words. Since lex provides no mechanism for assigning a single tag to a sequence of tokens, it was considered appropriate to identify this type of expression as a single token. Since each must be explicitly described by a regular expression, it is most efficient to assign tags at this step as well, although it would be possible to do this during the lookup stage, as is normal for other word tokens.

The DMF contains numerous expressions; the problem of determining which ones form invariant units is not a trivial one. However, it is obvious that any expression which contains a word having no separate entry of its own—as do the two listed above—must be a member of this class. All such expressions were extracted from the DMF and nine of the most common ones were included in the token file. (More could have been, but lex requires considerable space to store, and time to parse, a large regular expression.) The definitions for these unit expressions allow for variation in the amount of whitespace between words, the case of the initial letter and the elision of the final vowel (eg *parce que* or *Parce qu'*). Provision was not made for variant forms resulting from inflection or the insertion of extra words (eg *notamment* in *afin notamment de*), as this would have lengthened the regular

expressions considerably. Because of this, some words such as *parce* will occasionally go unrecognized.

B.2.2 Numeric Expressions

This category includes any string of non-whitespace characters which contains one or more decimal digits and begins and ends in a letter or a decimal digit. These make up about 1% of the tokens in the Hansard corpus, and take on a variety of grammatical roles which include quantifier, cardinal, ordinal, and proper noun. It is not clear that this list is exhaustive; such things as dates, for example, could behave in grammatically unique ways. Because of this ambiguity, and the fact that it is difficult to assign any subset of these roles to any subset of tokens in the category, it was decided to assign the single tag NN to all numeric expressions.

The stipulation that a numeric expression be bounded by a letter or decimal digit is made to prevent valid punctuation marks which immediately border the expression from being included in its token. Since numeric expressions—unlike words—are tagged solely during the tokenization stage, there is no opportunity for these marks to be pushed back onto the input later.

B.2.3 Punctuation

This includes all standard marks as well as all unknown characters. Most are single characters: the exceptions are dash, which comprises one, two or three contiguous hyphens; ellipsis, which comprises two or three contiguous periods; doubled single quote, which comprises two contiguous single quotes; and doubled back quote, which comprises two contiguous back quotes. Tag labels for most single-character marks consist of the character itself; those for the multi-character marks are "—", "..." and a double quote, respectively.

Certain marks are considered close enough in grammatical usage to warrant being grouped under a single tag. These include all varieties of quotation mark, which are tagged with a double quote (no distinction is made between left or right); left brace, bracket and parenthesis, which are tagged with a left parenthesis; and right brace, bracket and parenthesis, which are tagged with a right parenthesis.

Characters in the special punctuation class require special consideration because they can begin or end words. They should only be considered punctuation when followed by whitespace (this is a sufficient condition because lex tokenizes left to right). There is no need to explicitly impose this condition, however, due to the fact that lex always matches the longest possible prefix of the remaining input. For example, the string - *able* can only be tokenized as a dash followed by the word *able*. The string *-able*, on the other hand, is ambiguous because it can be tokenized as above or as a single word. Since the word *-able* is a longer prefix of the input than the dash '-' however, the string will always be tokenized—as intended—as a single word. This of course applies to the initial tokenization only; if no dictionary entry existed for *-able*, the transformation stage would strip the hyphen.

The final class included under this heading is that of unknown characters: any special characters which are not considered punctuation. It is important that these be tagged during tokenization to avoid their being assigned a default set of tags, intended for words, by the guess stage. They are tagged as X.

B.3 Dict and Inflect Data Files

These files contain the dictionary and a list of inflection rules which are used to recognize inflected forms of the words in the dictionary. They were created from the DMF's main dictionary and inflection tables with the help of a set of conversion programs. This process was mechanical but quite involved; it is described in detail in the following sections.

B.3.1 Character Set

The first issue concerns the method of handling the ISO character set in which the DMF is encoded. French corpora which do not have any representation for accents cannot be analyzed properly using the DMF in its current form—a seven-bit ASCII version is required. To create one, and to facilitate working with non-ISO compatible editors, an ISO-to-ASCII conversion program (iso) was written. It has three different modes of output: a viewing mode, which creates a facsimile of

each accented character using two or more ASCII characters; a strip mode, which removes accents from characters; and a code mode, which represents ISO characters by decimal codes, in a format compatible with the lex data file reader.

Since the Hansard corpus is bilingual, and since useful information is lost when accents are removed, the version of lex described here uses ISO data files.

B.3.2 Mnemonic Labels

DMF mnemonic labels for inflection class, POS, and some attributes were used. Labels for inflection classes were used verbatim; others were mapped into (possibly) different strings for compactness and to allow greater control of lex's tagset. All DMF labels used by lex as well as their mappings to lex labels are defined in the file *wordtagset.c*; to ensure consistency, this data was used by all of the conversion programs described below. Future versions of the DMF should be checked against *wordtagset.c* to ensure continuing compatibility.

B.3.3 Dictionary Conversion

B.3.3.1 Conversion of dmfp24.unx

The conversion of the DMF dictionary file (*dmfp24.unx*) into a lex dict file was performed by the program *dmf2dict*, which reads each entry in the DMF and writes a corresponding entry in dict format. Words are copied verbatim from the DMF; tags and inflection class labels are created from the DMF's mnemonic labels as described below.

Dmf2dict scans the labels associated with a word in the DMF entry left to right, looking for a POS, an inflection class and any known attribute labels. Exactly one POS and at most one inflection class label must be found, or an error is signalled; if no inflection class label is found, a default label is used. When the entire entry has been scanned, a tag is created by appending the strings into which the attribute labels were mapped to that into which the POS was mapped. A dict entry consisting of the word, the DMF inflection class label (unchanged), and the tag is then written. Note that the order of the mapped attribute strings within the tag is the same order in which the attribute labels were listed in the DMF entry. This means that

and quantifier entries by replacing the initial 'n' with 'a', 'o', and 'q', respectively. Changes to the inflection class tables were also required, as described below.

B.3.3.4 Omitted and Added Forms

Dmf2dict has the capability of omitting entries which are associated with certain labels. The only such label was *loc*, which identifies expression entries. These were omitted because the sequence of words in the expression does not necessarily always act as an expression and hence should not automatically be tagged as such. Some exceptions to this rule are tagged during tokenization, as described in section B.2.1.3.

Some other special forms were manually deleted from the dict file created by dmf2dict. These were the unmarked contractions *au*, *aux*, *du* and *des*; they were omitted to allow them to be handled specially during the rules phase, as described below.

Some very common words which are not in the DMF were manually added to the dict file. These included *d'*, *j'*, *l'*, *qu'*, *jusqu'*, and *lorsqu'*.

B.3.4 Inflection Rule Conversion

This stage involved the conversion of DMF inflection tables to lex inflection rules. Of the four inflection tables, two—the one for pronouns, and that for determiners—were ignored. These tables do not add any new word forms to the DMF's lexicon, although they do add additional tags to some words. In most cases, the additional tags represent fairly fine distinctions and were not considered to be worth the extra ambiguities which they induced.

The verb inflection table was converted by the program *verb* into an intermediate format which contains many duplicate suffixes, each associated with a replacement string, an inflection class and a single tag.

The program *noun* is used to do the same thing for noun-format inflection tables. Due to the inflection class problem mentioned in the previous section, it was necessary to create separate tables for each of the parts of speech formerly assigned the 'n' inflection class. For quantifiers and ordinals, the tables are small, since they

used only a few of the 'n' inflection labels (11 and 3 respectively). Since this is not the case for adjectives, the entire noun inflection table was duplicated, with 'n' class labels changed to 'a' class labels. The noun program takes a string as input, which it uses as the main POS label and places at the beginning of each tag which it writes. For the four new inflection tables, the strings NC, AQ, OD and QN were used.

The final stage in inflection rule conversion was performed by the program *mrgrules*, which merges identical elements in successive rules, beginning with suffixes and ending with tags. Mrgrules requires that its input be sorted so that rules in the intermediate form which have identical elements are adjacent. This was accomplished by concatenating the five intermediate inflection rule files and using the unix sort program on the concatenation. The final output from mrgrules is thus an inflect data file with rules in the proper format which includes all inflection classes.

B.4 Rules Data File

This file is divided into two parts, the first containing token transformation rules and the second containing guess rules. Since the two types of rules operate independently, they are discussed in separate sections below.

B.4.1 Transformation Rules

Transformation rules are transformations which are applied to tokens which could not be found in the dictionary. Each consists of a triggering regular expression which defines the set of tokens to which it applies, and an action, which can be a character mapping, a pattern replacement, or a suffix removal (the latter with the intention that, if the rule succeeds, the stripped suffix is pushed back onto the input to be recognized as the next token). Rules are applied sequentially in order of occurrence in the file to all results of all previous rules (including the original token), working backwards from the most recent, until one of the transformed tokens is found in the dictionary.

Due to the design of the token file (see section 2), the only tokens which will be

encountered at the transformation stage are words. These may be divided into five classes of interest, defined by the presence of capital letters, periods, apostrophes, hyphens and unmarked contractions. Although these classes are not disjoint, it is convenient to discuss them separately because a distinct set of transformation rules applies to each. The classes and their rules are described in the following sections. A discussion of the optimum ordering of the four rule sets is postponed until the final section.

B.4.1.1 Capitalized Words

Most capitalized words in any corpus will require conversion to lowercase, since very few entries in the DMF are capitalized. To avoid interfering with other words—included in the token because they are attached with an apostrophe or hyphen—before they have been looked up in their original form, lowercase conversion is performed only on the first letter of a word. Words consisting solely of capitals can easily be converted prior to lexical analysis if they are frequent enough to present a problem.

There are four usual reasons why a word is capitalized: it is part of a title, it is capitalized by convention, it begins a sentence, or it is a proper noun. Words in the first three categories should be converted to lowercase and looked up in the dictionary; proper nouns should not, as there is a possibility that they have a homonym whose tags they will erroneously be assigned. There is clearly no way to identify proper nouns, but sentence-initial capitals are easy to pick out. This observation suggests a strategy of converting only those words which begin a sentence, thereby hopefully circumventing the homonym problem. If it is assumed that there is a guess rule which tags all capitalized words as proper nouns, the performance of the sentence-initial strategy (B) compared to the default strategy of making all words lowercase (A) will depend on the frequencies of capitalized words as follows:

$$\text{error proportion for A} = P \times P_H$$

$$\text{error proportion for B} = P_F \times P_H + W - W_F$$

where

P is the proportion of proper nouns in the corpus

P_H is the proportion of proper nouns with homonyms in the dictionary

P_F is the proportion of proper nouns in the corpus which immediately follow a period

W is the proportion of capitalized non proper nouns in the corpus

W_F is the proportion of capitalized non proper nouns in the corpus which immediately follow a period

For the Hansard corpus, these proportions were estimated as follows: $P = 3\%$, $P_F = 1\%$, $W = 6\%$, $W_F = 3\%$. This makes A the better choice, regardless of the value of P_H, so it was the strategy selected. However, it is not clear that the Hansard corpus is representative—the value for W in particular seems suspiciously high—so strategy B should perhaps be considered for other corpora.

Another idea would be to add a proper noun tag to all capitalized words which do not follow a period. Unfortunately, this is not currently possible with lex. In any case, it would benefit only $P_H \times (P - P_F)$ of the words and would unnecessarily increase the ambiguity of $W - W_F$ of the words (assuming strategy A is used). If P_H is estimated at 20%, the relevant proportions are .2% and 3% respectively.

B.4.1.2 Words Containing Periods

These include acronyms, abbreviations and tokens to which a sentence-ending period has been erroneously appended. Tokens in the first two categories which have not been found in their original form in the dictionary will probably not benefit from a transformation. In an attempt to recognize tokens in the latter category, a suffix removal rule was used to strip trailing periods.

The problem is to distinguish between tokens to which this rule can be usefully applied and those to which it cannot. Although it is obviously not foolproof, a fairly good heuristic is to take the presence of periods within the token as indicative of an acronym—accordingly, such tokens were not included in the triggering pattern for the period stripping rule.

B.4.1.3 Words Containing Apostrophes

For contractions marked with apostrophes, a suffix rule was used to split the token after the first apostrophe. For example, *j'ai* becomes *j'* and *ai*. This works because prefixes for most common contractions are in the DMF.

Apostrophes occurring at the beginning or end of a word can be single quotes which have been erroneously tokenized with the word. Suffix rules were used to separate the quote mark from the word in both of these cases. To make the rule work when a quote precedes a word, an entry for the single quote character was added to the dictionary. Thus the token *'octobre* would be first split into *'* and *octobre*. Since the single quote is in the dictionary, the rule would succeed and *octobre* would be pushed back onto the input.

The rule for apostrophes and those for single quotes can interact. If the rule for apostrophes comes first, any tokens to which it has been unsuccessfully applied will match the triggering regular expression for the quote rule which strips a trailing single quote. For example, the token *aaa'bbb* would be split into *aaa'* and *bbb* by the apostrophe rule. If *aaa'* is not in the dictionary, it will be converted into *aaa* and *'* by the quote rule; this will probably result in an error, as apostrophes embedded within words are not usually intended as quotes. To avoid this problem, the quote rules were defined before the apostrophe rule, thereby precluding any possibility of interaction between the two.

B.4.1.4 Words Containing Hyphens

Before considering true hyphenated words, there are two artificial forms, created during tokenization, which must be dealt with: words which were contiguous with a dash and to which the first hyphen in the dash was appended, and broken words which were tokenized with an extra hyphen inserted. The first of these forms requires a suffix rule to push a trailing hyphen back onto the input; the second a replacement rule to strip all hyphens. Rule order for the two must be as indicated, otherwise trailing hyphens would be deleted rather than pushed back to form part of the dash. Both rules have potential side-effects for normal tokens, but neither is harmful. The first will cause any valid words with a trailing hyphen to be inter-

preted as being followed by a dash, which is probably as likely an interpretation as any. The second will allow words with variant hyphenated forms to be recognized.

True hyphenated words to which neither of the previous two rules apply are likely to be either a series of words joined with hyphens, or standard affixes joined to words. In either case, it is reasonable to split the token at the hyphen(s) in an attempt to recognize its component parts. Two rules were used to accomplish this: the first is a suffix rule which splits the token after the first hyphen and the second is a pattern rule which removes a trailing hyphen. The idea is that the first rule tests for the presence of a known prefix (which are listed with trailing hyphens in the DMF) and the second will test for an ordinary word; rule order must obviously be preserved.

There are two deficiencies in these rules: they do not test for known suffixes, and they stop at the first unrecognizable component of a hyphenated form. Although it would require an awkward rule set, both of these problems could be corrected; the main reason for not doing so is that it would be detrimental to speed. The reason for choosing to recognize prefixes but not suffixes is that there are more of them listed in the DMF (741 versus 295), and because they seem more likely to be attached with a hyphen (ie, instead of being concatenated directly).

B.4.1.5 Unmarked Contractions

There are four very common contracted forms in which two words are merged without explicit markings: *au*, *aux*, *du* and *des*. These forms correspond to a preposition followed by a determiner (eg *du* = *de* + *le*), and it is most accurate to tag them as such. To do this, a pattern rule was first applied to split each form into two dummy constituents, separated by a period; eg, *du* becomes *de.le*. An affix rule was then used to push the second constituent, beginning with a period, back onto the input. The final tokenization is therefore a preposition, eg *de*, followed by a determiner marked with a period to indicate that it is an artificial token, eg *.le*. Entries for *.le* and *.les*, tagged as determiners, were added to the dictionary to make this work.

B.4.1.6 Ordering for Correctness

The order of application of transformation rules can affect both the correctness and speed of the analysis. Although the ordering of each of the four rule sets (for words containing capital letters, periods, apostrophes and hyphens) must be as specified in the previous sections, the order of the sets themselves can be varied and should be chosen to optimize performance.

The comparative analysis of alternate sequences of rules is simplified by the fact that each of the rule sets defined in the previous sections operates on words with a single distinguishing feature. Rules can only interact on words which are characterized by two or more of these features. Most interaction occurs on words with exactly two features, as those with three or more are rare. This suggests that an analysis which considers only pairs of rules (rather than triples or quadruples) is appropriate. Moreover, since the last set of rules described in the previous section (those for unmarked contractions) operate only on four specific tokens, they will not interact with any other rule set except the one for capital letters so the correctness requirement in that case can obviously be satisfied by placing the contraction rules after the capital rules. The discussion which follows therefore applies to only the first three rule sets.

If A and B are any two rules which apply to some token x, call xA the string resulting from A acting on x, xB the string resulting from B acting on x, and xAB the string resulting from B acting on xA. The string sequence resulting from the application of AB to x is then xA, xAB, xB and that resulting from the application of BA to x is xB, xBA, xA. Both of these sequences will be generated only up to the first string which is found in the dictionary. Because the rules defined above are commutative, xBA = xBA for all x to which both A and B apply. There is therefore no string generated by a particular sequence of two rules which is not eventually generated by the inverse sequence. This does not mean that order is irrelevant however. Consider the sequence AB above and suppose that xB is in the dictionary and is a valid interpretation of x. If xA is also in the dictionary, but is invalid for x, then AB will make a mistake because it will generate only xA. BA will not make a mistake because it will generate only xB.

This is illustrated by the token *Avant-corps*, to which the capital and hyphen rules apply. Ignoring stripped suffixes, the strings potentially generated by the capital/hyphen sequence and its inverse are, respectively:

avant-corps, avant-, avant, Avant-, Avant

Avant-, Avant, avant-, avant, avant-corps

The capital/hyphen sequence will therefore stop after generating the correct interpretation, *avant-corps*, since this word is in the dictionary. The inverse sequence will stop after generating the incorrect interpretation, *avant*, since it is the first word in the dictionary.

The goodness of any rule sequence AB can be characterized by the probability that it finds a valid interpretation for some token x to which both A and B apply:

$$\Pr(AB|x) = \Pr(xA|x) \times \mathrm{Prd}(xA) + \Pr(xAB|x) \times \mathrm{Prd}(xAB) \times !\mathrm{Prd}(xA)$$
$$+ \Pr(xB|x) \times \mathrm{Prd}(xB) \times !\mathrm{Prd}(xA) \times !\mathrm{Prd}(xAB)$$

where Pr(y|x) is the probability that the transformed string y is the correct interpretation of x, Prd(y) is the probability that string y will be found in the dictionary, and !Prd(y) = 1 - Prd(y). The three terms in the expression are the probabilities for each of the three strings generated by AB; the negated factors in the second and third terms are the probabilities that the preceding strings will not be found in the dictionary.

It is possible to estimate the individual probabilities in this expression. Conditional probabilities can be obtained from a sample of tokens to which both rules apply by (hand) evaluating each of the three alternate interpretations and counting the proportion of correctness for each. Dictionary probabilities can be obtained by selecting a sample of strings of a given form from a corpus and counting the proportion of successful lookups. (It is tempting to count forms in the dictionary itself, but this would not take token frequencies into account.)

The tedium of making estimations is obviated by some observations. First, any rule pair for which the three forms of generated strings are disjoint with respected to membership in the DMF is order independent. For example, if AB generates xA, xAB, and xB, but it is known that at most one of these strings can be in the DMF

for any x, then BA will always yield the same result. The rule pairs capital/period, capital/apostrophe and apostrophe/period can be eliminated from consideration on this ground.

The remaining three pairs of rules all contain the hyphen rule set as a member. For the pairs hyphen/capital and hyphen/period, placing the hyphen rule last corresponds to trying the entire hyphenated token (with the period stripped or an initial capital made lowercase) before trying a hyphenated prefix. It seems obvious that this is the best order, because it gives precedence to hyphenated words which are in the DMF over any prefixes of the same words which are also in the DMF.

The situation is less clear, and the interaction more complex, for the final pair of rules, those for apostrophes and hyphens. These combinations are too rare, however—they occur in only 11 tokens out of almost 800,000 in the CRTC corpus—to justify an exhaustive analysis.

B.4.1.7 Ordering for Efficiency

The simplest ordering for efficiency is to put those rules which are used most often first. Rule use in the CRTC corpus was as follows:

capitals: 80326 applications
apostrophes: 33969 applications
periods: 17550 applications
hyphens: 7399 applications

This order meets the constraints discussed in the previous section. For the sake of clarity in the final version of the rules file, the period rule was placed before the apostrophe rule; this did not significantly affect performance.

B.4.2 Guess Rules

Guess rules are applied during the final stage of lex's algorithm if all transformation rules have failed. Each consists of a regular expression and a tag set; if a token matches the regular expression, it is assigned the corresponding tag set.

The most obvious use for a guess rule is to tag all tokens which begin with an uppercase letter as proper nouns, as discussed in section B.4.1.1. The only other

orthographic characteristic which is clearly associated with a tag is the presence of leading or trailing hyphens, which are identified as affixes, in keeping with the DMF's convention.

The tokens which remain unidentified after these rules are applied are not numerous (about .3% of the CRTC corpus). Many seem to be spelling errors or English words, so it is doubtful whether derivational morphology would be useful. Since they obviously play some grammatical role, however, it was decided to tag them with generic tags for the three most common open word categories: common noun, verb and adjective.

B.4.3 Results

Lex analyzed the CRTC corpus, containing 792,856 tokens, at a rate of 1720 tokens per second.

A random sample of 200 tokens was examined and found to 98.5% correct. Assuming that the distribution of the sample proportion is approximately Normal, this gives a 95% lower confidence bound of about 97%.

notation	interpretation
$V = \{v_1, \ldots, v_M\}$	word set containing M words v_i
$C = \{c_1, \ldots, c_L\}$	tag set containing L tags c_i
$O = o_1 \ldots o_T$	corpus containing T tokens o_t, where each $o_t =$ some v_i

Table C.1: Nomenclature

a training corpus—one from which HMM parameters are inferred; lexed corpora can also serve in the role of a test corpus—one which is used to test the tagging performance of the system.

A lexed corpus consists of a series of tokens (occurrences of words or punctuation), one per line, each of which is followed by at least one tab character and a space-separated list of potential tags for the token. Lists of tags are assumed to be identical for all tokens of the same word. No restrictions apart from those which are implicit in this definition govern the orthography of tokens or tags. Neither tokens nor tags have any intrinsic significance to ytag: the system is language independent, and any file in the proper format will work.

A tagged corpus has exactly the same format, except that only one tag (presumably the correct one) is listed for each token. Because tags depend on context, different tokens of the same word may be associated with different tags.

Corpora and vocabulary are formally represented as shown in table C.1.

C.1.3 Hidden Markov Models

An HMM consists of a set of states, a set of output symbols, and three probability distributions. It may be viewed as a source which enters successive states and generates one output symbol per state. The initial state, transitions between states, and the association between states and symbols are all random processes governed by the HMM's probability distributions. The formal components of an HMM are given in table C.2.

The coincidence between the notation for words and HMM output symbols, and corpora and symbol sequences is intentional, as they are identical for ytag's purposes. To emphasize this, HMM output probabilities will hereafter be referred

component	interpretation
$V = \{v_1, \ldots, v_M\}$	symbol set
$Q = \{q_1, \ldots, q_N\}$	state set
$\{\Pr(q_i),\ i = 1 \ldots N\}$	initial probabilities
$\{\Pr(q_k \mid q_j),\ j,k = 1 \ldots N\}$	transition probabilities
$\{\Pr(v_h \mid q_l),\ h = 1 \ldots M, l = 1 \ldots N\}$	output probabilities
$O = o_1 \ldots o_T$	symbol sequence produced by the HMM
$I = i_1 \ldots i_T$	state path for O

Table C.2: HMM nomenclature

Stage	Program	Input File(s)	Output File	Report Pgm
lexical analysis	retag	lexed	lexed	—
statistics collection	coll	lexed	stats	rstats
estimation	estm	stats	HMM	valhmm
reestimation	reestm	lexed, HMM	HMM	valhmm
tagging	tag	lexed, HMM	tagged	perf

Table C.3: Ytag stages

to as lexical probabilities.

C.2 Overview

Ytag comprises a set of programs which interact through files (or pipes) which they read and write. There is no overall control program or menu; the programs must be manually invoked as Unix commands.

Figure x illustrates the operation of the system. Its main function is to create an HMM from a training corpus and use it to tag a test corpus. This is accomplished in a series of stages: lexical analysis; statistics collection; estimation; reestimation; and tagging. Each stage involves an intermediate file, a program to generate it and in most cases another program to summarize its contents. Table C.3 lists the normal sequence of events in ytag, with the files and programs associated with each.

C.2.1 Lexical Analysis

The first step in creating an HMM is to lexically analyze a text file to produce a lexed file in the format described above; this task is not performed by ytag. However, ytag provides a tool—the program *retag*—which may be used to map any tag set into another one of equivalent or lesser specificity.

C.2.2 Statistics Collection

The next step is to gather statistics from a training corpus. This is performed by the program *coll* and results in a *stats* file whose contents may be summarized by the program *rstats*. The purpose of the stats file is to retain the information necessary for the estimation of HMM parameters—frequencies of words, word/ tag combinations, tags, and tag sequences of length 2 (tag-pairs) and 3 (tag-triples)—in compact form. This saves space and makes it possible to try different ways of generating an HMM from the same training corpus without having to make repeated (lengthy) passes through it.

Either a lexed or a tagged corpus can serve as a source of statistics for coll. There are some differences in the way coll operates on each type of corpus which affect the estimation process.

C.2.2.1 Constructing Valid Tag Sets

The first concerns the way in which the set of valid tags for each word is constructed. For a lexed corpus, this set is assigned the first time a word is encountered and is not altered thereafter. For a tagged corpus, the (single) tag associated with each token is checked against a list for the corresponding word and added if it is not already present. This implies that the list of tags for a word will be incomplete if some valid combination never occurs in a tagged training corpus. To remedy this, coll provides a special *vocabulary acquisition* mode which learns words and tags from a lexed corpus without recording any frequencies. (There is also an analogous *tag acquisition* mode, in which only tags are of interest.) A stats file created in this way can be designated as an initialization file for coll to ensure that the sets of tags for all known words are complete, no matter how small a training corpus is

statistic	increment
word frequency	1
word/tag frequency	$1/L$ for each tag
tag frequency	$1/L$ for each tag
tag-pair frequency	0
tag-triple frequency	0

Table C.4: Statistics from an ambiguous word

subsequently used. This precaution is only necessary if a tagged training corpus is to be used, since it is assumed that the lists of tags provided in a lexed corpus are complete.

C.2.2.2 Using Ambiguous Words

Another issue is the method of treating ambiguous words—those with more than one possible tag—within a lexed corpus. This varies depending on the statistic being collected: word frequencies are counted normally but word/tag and tag frequencies are incremented fractionally, and tag-pair and tag-triple frequencies are not counted. The latter are ignored because there is no simple way to determine the contribution which ambiguous words should make; their counts therefore reflect only the "naturally" unambiguous words in a lexed corpus. The following table summarizes the contribution made to each statistic by an ambiguous word with L possible tags[1].

C.2.3 Estimation

In this step, the program *estm* uses the frequency statistics in a stats file to estimate probabilities for an HMM, then writes the resulting model to an "HMM" file. The program *valhmm* may be used to report on the contents of an HMM file and check

[1]It should be noted that the fractional increments described here are not actually performed when coll counts frequencies, so they will not be reflected in the report generated by rstats. However, the stats file contains all information needed to make these increments and they are always performed prior to estimation.

probability	HMM prob	order = 2	order = 3
initial	$\Pr(q_i)$	$\Pr(c_i)$	$\Pr(c_l, c_m)$
transition	$\Pr(q_j \mid q_i)$	$\Pr(c_j \mid c_i)$	$\Pr(c_n \mid c_l, c_m)$
lexical	$\Pr(v_h \mid q_i)$	$\Pr(v_h \mid c_i)$	$\Pr(v_h \mid c_m)$

Table C.5: HMM $\Leftrightarrow$ word/tag probabilities for orders 2 and 3

its validity by ensuring that certain sets of probabilities sum to one (this is chiefly useful for debugging purposes). The estimation process has three main parameters which may be controlled by the user: the order of the model, the method used to modify observed frequencies, and the method used to compute probabilities from frequencies.

C.2.3.1 Model Order

The first parameter specifies the order of the HMM. This determines the length of the sequence of tags which constitutes an HMM state. In general, the higher the order, the more powerful (and expensive) the model. Ytag provides a 2nd order model in which states correspond to tags and a 3rd order model in which states correspond to tag-pairs. The mapping from word and tag probabilities to HMM probabilities depends on the order of the model. This is summarized in table C.5, for orders 2 and 3[2].

C.2.3.2 Frequency Modification

The second estimation parameter specifies the method of frequency modification to be used. If the observed frequencies from the stats file were used to make direct maximum likelihood estimates (see below), they would assign a probability of zero to each event (word, word/tag pair, tag, tag-pair or tag-triple) which did not occur in the training corpus. For any training corpus of finite size, this would be

[2]Note that the lexical probability for order 3 listed in the table is an approximation to $\Pr(v_h \mid c_l, c_m)$. It is customary to make this approximation, which is motivated by considerations of efficiency and justified by the intuition that conditioning word probabilities on the previous two tags does not add information which is not already inherent in the transition probabilities.

inaccurate, so estm corrects all zero frequencies to some small non-zero value and adjusts all other frequencies to compensate before using them to estimate probabilities. Two methods of modifying frequencies are provided: augmented corpus and Good-Turing.

In order to modify the frequencies for non-occurring events, it is necessary to decide exactly what these events are. For tags, tag-pairs and tag-triples, this is not a problem, as it is reasonable to assume that the universe of tags is exactly the set contained in the stats file. (For small training corpora, it is possible that rare tags will not occur—in these cases, coll can be used with an init file which contains all desired tags.) A similar assumption cannot be made for words or word/tag pairs, as no corpus can be expected to contain the entire vocabulary of a language. Estm therefore adopts the strategy of mapping all unknown words into a single special word whose list of tags contains all possible tags. A probability estimate is made for this word and for each word/tag pair in which it participates. These are used during tagging (see below) to make predictions for all words which are unknown to the HMM. (In this context, it should be noted that estm discards any words in the stats file which have zero frequency. Such words would have been placed in the stats file by coll's vocabulary mode and would not have occurred in the training corpus.)

Augmented Corpus Frequency Modification

The simplest way to avoid zero probabilities is to assume that the observed training corpus has been augmented by another in which every possible event occurs exactly once, and to modify the observed frequencies in accordance with this assumption. For words, word/tag pairs, and tags, the augmented corpus is also assumed to be ambiguous and the modification is performed so as to be consistent with the method used by coll to count events in an ambiguous corpus.

Good-Turing Frequency Modification

A more sophisticated method of modifying frequencies is to use the Good-Turing formula:

$$f' = (f+1)N_{f+1}/N_f$$

where

f' is the modified frequency

f is the observed frequency

N_f is the number of events with frequency f

The use of this formula is critically dependent upon the method for smoothing the N_f, which tend to be very noisy and sparse, especially for large values of f. The smoothing method used by estm involves a two step procedure which first averages each N_f by the zero values (if any) which surround it, then fits a 3rd order polynomial to the $\log N_f, \log f$ averaged data points.

Another requirement of the Good-Turing formula is that N_0—the number of events which occur with zero frequency—be known. For tags, tag-pairs and tag-triples this presents no problem, because the number of tags is assumed to be fixed, as described above. For word/tag pairs, N_0 is estimated by assuming a fixed (large) vocabulary size and a number of valid tags per word equal to the average number of tags per word in the training corpus.

C.2.3.3 Probability Estimation

The final estimation parameter specifies the method of estimating HMM probabilities from the corresponding (modified) frequencies. Estm provides three estimators—equal probability, maximum likelihood and combined order—each of which is valid for any combination of model order and frequency modifier, and each of which can be used independently for each set of HMM probabilities (initial, transition and lexical).

Equal Estimation

The simplest estimator ignores frequencies and makes all probabilities within each distribution equal. This effectively removes the source of knowledge embodied in the distribution from the system and is useful in allowing its contribution to be assessed. (Note that, for lexical probabilities, there is an apparent problem with

probability	HMM Prob	order 2 estimate	order 3 estimate
initial	$\Pr(q_i)$	$f(c_i)/T$	$f(c_l, c_m)/T$
transition	$\Pr(q_j \mid q_i)$	$f(c_i, c_j)/f(c_i)$	$f(c_l, c_m, c_n)/f(c_l, c_m)$
lexical	$\Pr(v_h \mid q_i)$	$f(v_h, c_i)/f(c_i)$	$f(v_h, c_l)/f(c_l)$

Table C.6: Maximum Likelihood estimates for orders 2 and 3

this method in that the sums reported by valhmm do not add to one. This is due to an implementation detail and does not affect performance.)

Maximum Likelihood Estimation

The standard method for estimating probabilities is to use the ratio of the frequency of an event of interest to the total frequency of events in its class; in statistical terminology, this is maximum likelihood estimation. Table C.6 gives the maximum likelihood estimates for 2nd and 3rd order models. where:

$f(e)$ denotes the frequency of event e

T is the number of tokens in the training corpus

Q, V, C are state, word and tag sets as usual

v_h, c_j denotes the joint occurrence of word v_h with tag c_j (the comma is normally used to indicate joint events; where no conflict of meaning arises, it is also used to indicate sequence, as in c_i, c_j)

The estimates listed for the conditional probabilities in table C.6 are via Bayes' law expansions, with the denominators cancelled, as they are all very close approximations to T. For a lexed training corpus, this approximation breaks down because the numbers of "naturally" unambiguous 2 and 3 token sequences in the corpus will not generally be equal, either to each other or to the number of tokens. Due to this, it is necessary to use marginal sums for the denominators in the estimates. For example, the third order transition probabilities are actually estimated by:

$$\Pr(q_j \mid q_i) = f(c_l, c_m, c_n) / \sum_{n=1}^{L} f(c_l, c_m, c_n)$$

Combined Order Estimation

As the order of an estimator is increased, the potential accuracy of its predictions increases, but it becomes less reliable. This is due to the fact that for a given training corpus size there are proportionally fewer $(n+1)$-tag sequence types represented than n-tag sequence types; an estimate made from an $(n+1)$-tag sequence will in general involve a lower frequency than one made from an n-tag sequence and hence will be less apt to be representative. The problem is exacerbated by the use of a lexed corpus, since, for example, tag-triples are only collected when a stretch of at least three naturally unambiguous words occurs.

One solution to this problem is to combine estimators of different orders in some appropriate way so that the reliability of the lower order estimators complements the precision of the higher order estimators. Ytag uses a simple, non-linear method of combination, in which only one order is ever used for any particular estimate. In this scheme, the highest order estimator (beginning with the value selected for the order parameter: 2 or 3) whose frequency is above a threshold value (which may be set by the user) is always used. If no estimator has this property, a first order estimate is used[3].

C.2.4 Reestimation

The next step is an optional one which involves refining an HMM with respect to some training corpus by means of the reestimation algorithm. This is performed by the program *reestm*, which reads a lexed corpus and an HMM file and creates a new HMM file.

C.2.4.1 The Reestimation Algorithm

Reestimation is an iterative algorithm which modifies HMM parameters using a training corpus. It is guaranteed to converge to a local maximum of the probability which the HMM assigns to the training corpus as a whole ($\Pr(O)$), in the notation presented above). The idea is that, if a large and representative corpus is used,

[3]This has the useful side effect of allowing the user to force the system to make first order estimates by specifying a very high value for the threshold frequency.

prob	HMM prob	reestimate
init	$\Pr(q_i)$	$\sum_{t=1}^{T} \Pr(O, i_t = q_i) / \Pr(O)$
trans	$\Pr(q_j \mid q_i)$	$\sum_{t=1}^{T-1} \Pr(O, i_t = q_i, i_{t+1} = q_j) / \sum_{t=1}^{T} \Pr(O, i_t = q_i)$
lex	$\Pr(v_h \mid q_i)$	$\sum_{t=1}^{T} \Pr(O, i_t = q_i, o_t = v_h) / \sum_{t=1}^{T} \Pr(O, i_t = q_i)$

Table C.7: Reestimates

maximizing this probability will yield an HMM which reflects the language well; in the current context, this means an HMM which can tag well. Reestimation is particularly useful for tagging because, unlike the estimators described above, it uses all of the information contained in the ambiguous words of a training corpus.

Each pass of the algorithm uses the HMM to compute joint state, symbol sequence probabilities (symbol sequence = training corpus) at each word position in the corpus. These are summed to compute maximum likelihood estimates for the parameters of a new HMM as shown in table C.7[4].

C.2.4.2 Reestimation Parameters

There are three basic parameters for reestimation: the starting point, the convergence criterion and the probability sets to be reestimated.

Starting Point

The first choice to make is that of a training corpus. To an even greater extent than is true of estimation, larger reestimation training corpora produce better models but incur more cost.

Once a training corpus has been chosen, estm must be used to generate an initial HMM from it. Although reestm does not enforce the requirement that the initial HMM be generated from the training corpus, this is the usual convention. For efficiency, reestm has no mechanism for handling words which occur in the training

[4]An alternate formula for initial probabilities is based on the probability with which each state appears at the beginning of the sequence, $\Pr(O, i_1 = q_i)$. For the current problem, this would yield very arbitrary estimates which depend solely upon which word happens to head the training corpus. Ytag therefore uses the absolute probability of each state, which is a much more reliable source of information.

corpus but not in the initial HMM; hence if the convention is to be departed from, care must be taken to ensure that the vocabulary of the initial HMM includes all words and tags in the training corpus.

The order of the initial HMM is the most important factor affecting reestimation, as this determines the order of the reestimated HMM. The other estimation parameters are also important because the probability maximum found by reestimation is a local one and therefore depends on the starting point. In general, the better the initial HMM, the better the results, although once a certain level is reached, the difference can be expected to become insignificant.

Convergence Criteria

Reestm computes two quantities which may be used to determine the number of iterations required for convergence: error and perplexity. Error is a measure of the difference between the initial and reestimated HMMs for a single iteration (where initial means the HMM with which the iteration begins and not necessarily the initial HMM for the program). It is defined as the average absolute change made to each probability, taken over all probabilities which were changed. The error generally decreases monotonically, but its rate of decrease depends on the initial HMM, as well as the size and nature of the training corpus. Reestm has the capability of terminating automatically when the error falls below a specified threshold.

Perplexity is a more direct measure of the quantity which reestimation is supposed to maximize: $\Pr(O)$. It is defined as

$$1/\Pr(O)^{1/T}$$

ie, the reciprocal of the Tth root of the probability assigned to the corpus by the HMM, where T denotes the size of the training corpus as usual. Perplexity is a measure of both the inherent complexity of the training corpus—independent of corpus size—and the degree to which the HMM "fits" it: higher values mean more complexity and/or less fit. Any determination of convergence based on perplexity must be manually applied, as reestm does not provide this capability. It should be noted that, due to efficiency considerations, the perplexity values reported by

reestm at each iteration are those for the HMM with which the iteration begins, rather than that which it produces.

Probability Sets

A nice property of reestimation is that it can be applied to some of the HMM probability distributions while other(s) are held fixed. This allows, for example, lexical probabilities to be reestimated from a large corpus while estimates for initial and transition probabilities from some other source are not changed. Reestm provides the capability of fixing any combination of probability sets in this way.

Another property of reestimation is the fact that, as a consequence of its convergence to a maximum likelihood estimate, it will assign a probability of 0 to unseen events (eg lexical combinations, transitions, etc, which do not occur in the training corpus). This is obviously undesirable, as it is precisely the condition which the frequency modification techniques used by estm are designed to avoid. Some balance must therefore be struck between the HMMs yielded by estimation—which model common events poorly but uncommon events relatively well—and those yielded by reestimation, which model common events better but uncommon events poorly. An ad hoc way of doing this is to stop iterating before the point at which reestimation drives the probabilities of unseen events to zero. Another is to set some a priori lower bound on all reestimated probabilities. This capability is provided by reestm, which allows the specification of a *floor threshold* for each set of probabilities.

C.2.5 Tagging

The final stage is performed by the program *tag*, which reads a lexed file, uses the Markov model contained in an HMM file to disambiguate it, and writes a corresponding tagged file. By default, tag writes a header to the tagged file containing information about the training file, estimator, and tagging method used to create it. The program *perf* creates a tagging performance report which includes this information as well as statistics on the number of correct tags (for which it requires a correctly tagged *benchmark* version of the original lexed file).

When tag reads a lexed file, it is dependent on the vocabulary known by the

HMM. Its behaviour on encountering a tag or word not in this vocabulary is consistent with the assumptions made by estm when generating the HMM: unknown tags cause immediate errors; unknown words are mapped onto a special unknown word in the model. The lexical probability of an unknown word given any tag with which it is listed is taken from the lexical probability of the special unknown word for the same tag. Tag is silent if it encounters a known word with a tag list which differs from the tag list in the HMM, but the latter is always used. (In other words, tag only uses the tag lists in a lexed file for unknown words).

To disambiguate, tag uses an integrated tagging and self-error detection system which depends on five main user-controlled parameters, one of which specifies the tagging method and four of which specify the error detection (flagging) method.

C.2.5.1 Tagging Methods

Tag offers four different methods of tagging, each of which corresponds to a distinct method of picking an HMM state path (and therefore a sequence of tags) for a sequence of words. Sequences of words submitted to the tagging algorithm are chosen as the shortest possible sequences of ambiguous words which begin and end with an unambiguous word (although this may not be the case at the beginning and end of the lexed corpus).

The first path (ML) is the maximum likelihood path, which consists of the state for each word which maximizes the joint state/word sequence probability. The second path (NL) consists of the next most likely state for each word. The third path (P1) is the one for which the joint path, word sequence probability is highest. The fourth path (P2) is the one for which the joint path, word sequence probability is next highest. Formally:

ML: $I = i_1 \ldots i_T$, where $i_t = \arg\max_{q_j, j=1\ldots N} \Pr(O, i_t = q_j)$

NL: $I = i_1 \ldots i_T$, where $i_t = \text{2nd}\arg\max_{q_j, j=1\ldots N} \Pr(O, i_t = q_j)$

P1: $\arg\max_I \Pr(O, I)$

P2: $\text{2nd}\arg\max_I \Pr(O, I)$

It should be noted that the four paths are not necessarily disjoint; in particular, ML, P1 and P2 often coincide. NL will always usually differ from ML at each ambiguous word (it will always differ for second order models).

The most useful paths are ML and P1: if the HMM's estimates are very accurate, P1 tends to give better performance because it reflects the plausibility of the path as a whole; if not, ML is the better choice. The two other paths are chiefly useful for comparison purposes and for error flagging, as described below.

C.2.5.2 Error Flagging Methods

When error flagging is selected, tag attempts to identify the tag assignments about which it is uncertain. These are marked with an error flag in the output file. (No attempt is made to correct the assignments or propose alternatives.)

The algorithm used by tag for error flagging involves comparing the path used for tagging (the tagging path) with one other (the comparison path). If the ratio of the tagging path to the comparison path (defined in an appropriate sense, described below) does not meet a given threshold, the tagging path is deemed to contain potential errors at certain points. At each such point, the state from the tagging path is compared to the state from a third path (the alternate path); if the two differ, an error flag is set.

This algorithm has five parameters which are under user control: the method of making comparisons between the tagging and comparison paths, the threshold value used for comparisons, the tagging path, the comparison path, and the alternate path. The tagging path is fixed when the tagging method is selected, as described above; the other four parameters are described in the following sections.

Comparison Method and Threshold Value

Two methods of comparing paths are provided: path based and word based.

Path based comparison involves taking the ratio of the path probability for the tagging path to that of the comparison path:

$$\Pr(O, I_{\text{tag}}) / \Pr(O, I_{\text{compare}})$$

If this is less than the threshold value, each state on the tagging path is deemed to be a potential error.

Word based comparison involves taking the ratio of the maximum likelihood probability of each state on the tagging path to that of the corresponding state on the comparison path:

$$\Pr(O, i_t) / \Pr(O, j_t)$$

where

$$I_{\text{tag}} = i_1 \ldots i_T$$

$$I_{\text{compare}} = j_1 \ldots j_T$$

For each t where the ratio is less than the threshold value, the state i_t is deemed to be a potential error.

The useful range and the sensitivity of the threshold value depend on the tagging and comparison paths selected—both quantities can vary widely. In all cases however, the sense is the same: increasing the value increases the number of errors flagged, until the saturation point is reached (usually when all ambiguous tokens have been flagged as errors).

Comparison Path

The path used for comparison can be any of the four paths—ML, NL, P1 and P2—available for tagging (although it makes little sense to choose the same path for tagging and comparison). In addition, one other "path" is available: the total probability, $\Pr(O)$, which the HMM assigns to the sequence as a whole. For path based comparisons with this option, $\Pr(O)$ is substituted for the comparison path probability; for word based comparisons, $\Pr(O)$ is substituted for the maximum likelihood probability of each state on the comparison path.

Alternate Path

As described above, the alternate path is used as corroboration for points on the tagging path deemed to be potential errors by the comparison process: errors are flagged only if the alternate and tagging paths differ at such points. The alternate

path may be any of the four standard paths, plus one other—the PX path—which is guaranteed to differ from the tagging path (no matter which one has been chosen) at each ambiguous word.

The behaviour of the alternate path comparison is dependent on the order of the model. Because states do not correspond directly to tags in a third order model, state paths which differ at some point can map to tag paths which do not differ at that point. This may cause confusion when the alternate path is being relied upon to differ from the tagging path at each point. In such cases, it is safe to use the PX path, which is not affected by order.

C.3 Program Descriptions

This section describes the individual programs which comprise ytag. The descriptions are in alphabetical order by program name in the sections which follow.

There are two switches—undocumented in what follows—that are common to all programs:

-h Print a help message with the syntax, options and a brief description of the program's function.

-d Activate debugging, if any.

Four types of file are used throughout ytag:

lexed A lexically analyzed file. Each line must begin with a token, followed by any number of tab characters, followed by a space-separated list of tags, of which there must be at least one. The list of tags must be identical for each occurrence of a token.

tagged A tagged file. Each line must begin with a token, followed by any number of tab characters, followed by a single tag. Different occurrences of a token may be associated with different tags.

stats A statistics file, generated by the program coll. This contains lists of words and tags from a lexed file, along with frequency statistics for words, word/tag pairs, tags and tag sequences of length two and three.

HMM An HMM file, generated by the programs estm or reestm. This is a binary file representation of an HMM, along with all information necessary to use the HMM for tagging.

C.3.1 Coll

Collect statistics on a lexed or tagged file and write them to a stats file.

C.3.1.1 Syntax

`coll [-v|-t] [-i` *init-file*`] [-r` *rep-file*`] [`*src* `[`*dst*`]]`

C.3.1.2 Parameters

src Input lexed or tagged file (or stdin).

dst Output stats file (or stdout).

`-v` Vocabulary mode. Coll collects words name, tag names, and word/tag lists only; no frequencies are recorded. The purpose is to create an *init-file* in which tag lists for each word are complete. This option is not designed to work with tagged *src* files.

`-t` Tag mode. This is the same as vocabulary mode, except that only tag names are collected. The purpose is to ensure that the tag set contained in a stats file is complete.

`-i` Designate stats file *init-file* as an initialization file. The contents of this file are read prior to reading *src*; all statistics from *src* are added to those from *init-file*. If no *init-file* is specified, the environment variable COLLINIT is checked. If it is active, its contents specify the name of the initialization file; otherwise none is used.

`-r` Write a statistics report to *rep-file*. This is the same report generated by rstats; its format is described below.

C.3.2 Estm

Estimate a tagging HMM from the word and tag statistics in a tagged file.

C.3.2.1 Syntax

`estm [-e [o]/[f]/[i]/[t]/[w]] [-r th] [-z] [src [dst]]`

C.3.2.2 Parameters

src Input stats file (or stdin).

dst Output HMM file (or stdout).

`-e` Specify the estimator to use (the default is `2T/AC/ML/ML/ML`):

o is order of the model: `2T` or `3T`

f is the frequency modifier: `AC` or `GT`

i is the initial probability estimator: `EQ`, `ML`, or `CO`

t is the transition probability estimator: `EQ`, `ML`, or `CO`

i is the lexical probability estimator: `EQ`, `ML`, or `CO`

where:

`2T` means 2nd order; HMM states correspond to tags.

`3T` means 3rd order; HMM states correspond to tag pairs.

`AC` means augmented corpus; each type in each statistic (words, word/tag pairs, tags, tag-pairs, tag-triples) is assumed to have occurred once—frequencies in the stats file are modified accordingly.

`GT` means Good-Turing; each frequency in the stats file is modified to $(f+1)N_{f+1}/N_f$, where f is the original observed frequency and N_f the number of types with frequency f.

`EQ` means set the probabilities of all types equal.

`ML` means maximum likelihood; estimate probabilities from relative frequencies.

`CO` means combined order; use the ML estimate of the highest order (3, 2 or 1) whose frequency count is above some threshold. The default threshold of 1.0 can be changed with the `-r` option. Currently, CO estimates are only available for transition probabilities; selecting CO for any other probability set causes ML estimates to be used.

`-r` Set the threshold frequency for CO estimators to *th*.

`-z` Keep all words which have zero frequency in *src*, but modify their frequency to one. The default is to discard such words. This option is only intended for use with reestimation, when the initial model is estimated from a smaller corpus than that on which reestimation is to performed.

C.3.3 Perf

Make a status and performance report on a tagged file.

C.3.3.1 Syntax

`perf [-b` *bmk*`] [`*src* `[`*dst*`]]`

C.3.3.2 Parameters

src Input tagged file (or stdin). This file must have a header (ie, have been generated by running tag without the `-n` switch). For maximum information in the report, it should also be in list format, with all applicable tags listed after the selected tag for each token (ie have been generated by running tag with the `-l` switch).

dst Output report file (or stdout). This is in two parts: the first is a copy of the header written by tag and contains a summary of the files and methods used to create the tagged file. This includes:

Lexed file the name of the lexed file which was tagged.

Tagged file the name of the tagged file.

Statistics file the name of the stats file used to create the HMM.

Tagging Method a coded description of the parameters to tag, in the following format:

f/t/d/a/r

Where *f* is the flagging method, *t* is the tagging path, *d* the comparison path and a the alternate path, as described in section C.3.7. *r* is the error flagging threshold, as described in the same section.

The second part of the report contains statistics on the tagging and flagging performance. The following items are included (those which are followed by a "-l" in parentheses are not valid unless the -l switch was selected for tag):

- average tags per token—the average number of tags per token in the lexed file (-l).
- average tags per ambig token—the average number of tags per ambiguous token (those with more than one tag) in the lexed file (-l).
- tokens—the total number of tokens in the lexed file (and in the tagged file).
- ambig tokens—the number of ambiguous tokens; also expressed as a percentage of the total number of tokens (-l).
- tokens correctly tagged—the number of tokens correctly tagged (by comparison to those in the benchmark file); also expressed as a percentage of the total number of tokens.
- ambig tokens correctly tagged—the number of ambiguous tokens correctly tagged; also expressed as a percentage of the total number of ambiguous tokens (-l).
- ambig tokens tagged by chance—the percentage of ambiguous tokens which would be correctly tagged by picking tags at random, estimated by the number of ambiguous tokens divided by the total number of tags associated with all ambiguous tokens (-l).
- error flags—the number of error flags (ie, the number of tokens flagged as errors by tag); also expressed as a percentage of the total number of tokens.

- valid error flags—the number of error flags which are valid; also expressed as a percentage of the total number of error flags.
- actual errors flagged—the number of tagging errors which were flagged; also expressed as a percentage of the total number of tagging errors.
- errors flagged by chance—the percentage of tagging errors which would be flagged by assigning this number of error flags to ambiguous tokens at random, estimated via the joint independent error and flag probabilities. (-l)

`-b` Designate *bmk* as a benchmark file, that is, a correctly tagged version of *src* for comparison. If this parameter is not specified, the benchmark file name is taken from the environment variable TAGBENCHMARK. If TAGBENCHMARK is not active, or the benchmark file cannot be opened, perf aborts.

C.3.4 Reestm

Reestimate an HMM using a lexed training corpus.

C.3.4.1 Syntax

`reestm [-v] [-s` *s*`] [-i` *i*`] [-t` *t*`] [-f` *f*`] [-r` *r*`]` *trg hmm*`|+ [`*dst*`]`

C.3.4.2 Parameters

trg Lexed training corpus to use for reestimation. If `+` is specified, this is taken from stdin.

hmm HMM file containing the HMM to be reestimated, which may be one produced by a previous reestimation. Reestm does not make allowance for unknown words in *trg*, so *hmm* should originally have been generated from *trg* or a superset thereof.

dst Output file for reestimated HMM (or stdout). This may be the same as *hmm*.

`-v` Select verbose mode; issue status reports. The first report echoes reestm's parameter settings; subsequent reports are issued for each iteration and include

the following items:

error The difference between the initial and final HMM's, computed as the average change in probabilities taken over all changed probabilities.

perplexity This is the reciprocal of a normalized version of the probability, $\Pr(O)$, assigned to the word sequence O (ie, *trg*) by the initial HMM. Specifically, perplexity $= 1/\Pr(O)^{1/T}$, where T is the number of tokens in O.

`-s` Set the number of segments used to *s* (the default is 1). Using more segments causes reestm to use less memory. Normally, the number of segments are adjusted automatically, but there are occasions when the automatic algorithm fails. If reestm returns a "can't alloc" error message, setting the number of segments to some small value greater than 1 will normally correct the problem.

`-i` Set the maximum number of reestimation iterations over *trg* to *i* (the default is 10). (Note: reestimating for a fixed number of iterations n produces exactly the same results as running reestm n times with *hmm* set to the *dst* file produced on the previous run. The former method will be faster because it avoids the overhead of reading in *trg* for each iteration–for large corpora, this is not a trivial consideration!)

`-t` Set the tolerance for convergence to *t* (the default is 0.001). At the end of each iteration, the reestimation error (described above) is compared to *t*; if it is less than *t*, reestm stops.

`-f` Fix (do not reestimate) the specified set of probabilities. *f* may be any of `I`, `T` or `L`, for output, transition and lexical (if all three are selected, no change is made to the original HMM).

`-r` Set a floor threshold for each set of probabilities. *r* must be of the form: *fi/ft/fl*, where *fi*, *ft* and *fl* are the thresholds for initial, transition and lexical probabilities respectively (the default is `0/0/0`). If a reestimated probability is less than or equal to the floor threshold, it is not used to update the corresponding HMM probability.

C.3.5 Retag

Retag a lexed file using a different tag set.

C.3.5.1 Syntax

`retag [-r]` *map-file* `[`*src* `[`*dst*`]]`

C.3.5.2 Parameters

src Input lexed file (or stdin)

dst Output lexed file (or stdout)

map-file File containing mappings from tags in the old set to tags in the new set. Each line in this file is of the form:

old-name new-name

and specifies that all occurrences of the tag *old-name* in *src* are to be replaced by *new-name* in *dst*. Any tags in *src* with no entry in *map-file* are copied verbatim to *dst*. Any duplicate tag entries in the tag lists written to *dst* are removed (whether or not they were caused by the mapping); if *map-file* is empty, this is the only action performed by retag.

`-r` Reverse the order in which the new tag lists are written.

C.3.6 Rstats

Make a summary of the statistics in a stats file.

C.3.6.1 Syntax

`rstats [`*src* `[`*dst*`]]`

C.3.6.2 Parameters

src Input stats file (or stdin).

dst Output file (or stdout) for the statistics report. This report includes the following items:

Vocabulary the number of words and tags in the total vocabulary contained in *src*. This reflects the contents of the initialization file (if any) and the lexed file which were used by coll to create *src*.

Sample the number of words, tags and tokens in the lexed file used to create *src*. The discrepancy between these numbers and those for the "Vocabulary" entry is due to the contribution made by the initialization file. (Note: it is possible that the number of tags listed under "Sample" will be less than that listed under "Vocabulary", even if no initialization file has been used to generate *src*. This is because the "Sample" counts do not include tags with frequency zero—those which never occur in the lexed file as the sole tag for a token.)

Table of statistics the next item is a table containing six statistics of interest from the lexed file used to create *src* (no contribution is made by the *init-file*). The following list describes the entry in each column of the table (beginning with "Types") for each statistic:

words number of words; number of word tokens; min, max, mean, and standard deviation of tokens per word.

ambig words number of ambiguous words; number of ambiguous tokens; min, max, mean and standard deviation of tokens per ambiguous word.

tags per word number of words over which the tags per word statistic was taken; total number of tags for all words; min, max, mean and sdev of tags per word.

tags number of tag types; number of tag tokens for unambiguous words; min, max, mean and sdev of tag tokens per tag.

bi-tag sequence number of distinct bi-tag sequence types (taken over unambiguous words) which occur at least once; total number of bi-tag sequence tokens collected; min, max, mean and sdev of bi-tag

tokens per bi-tag type.

tri-tag sequence number of distinct tri-tag sequence types (taken over unambiguous words) which occur at least once; total number of tri-tag sequence tokens collected; min, max, mean and sdev of tri-tag tokens per tri-tag type.

C.3.7 Tag

Tag a lexically analyzed file, using the data contained in an HMM file.

C.3.7.1 Syntax

`tag [-n] [-l|-L] [-t [f]/[t]/[d]/[a]] [-r th] hmm|+ [src [dst]]`

C.3.7.2 Parameters

src Input lexed file (or stdin).

dst Output tagged file (or stdout). This is a disambiguated version of *src*, with an optional header prepended (see section C.3.3 for a description of the header's contents). There are two formats for *dst*: *single tag*, in which only the chosen tag is listed for each token; and *list*, in which the chosen tag heads a list of all applicable tags for the token (the remainder of the list is in no special order). Error flags may also be present: these are circumflex characters (^) placed one space before the first tag.

hmm HMM file to use for tagging. If + is given instead of a file name, the HMM will be read from the standard input. If the input lexed file is also to be read from standard input (ie, *src* has been omitted), it must follow the HMM on this stream.

`-n` Suppress the header which is normally written to a tagged file.

`-l` Write *dst* in list format, as described above; the default is to use single tag format. The list of tags supplied for each token is taken from the vocabulary in *hmm*, not from the tags listed in *src*.

-L Write *dst* in list format, but perform no tagging. This option is useful to recover a lexed file from a tagged file, using the vocabulary and tag sets in *hmm*.

-t Specify the tagging method (the default is `FN/ML/NL/NL`):

f is the type of error flagging: `FN`, `FP`, or `FS`

t is the tagging path: `ML`, `NL`, `P1` or `P2`. This is the path used to assign tags.

d is the comparison path: `ML`, `NL`, `P1`, `P2` or `TT`. This path is compared to the tagging path to determine if it contains errors.

a the alternate path: `ML`, `NL`, `P1`, `P2` or `PX`. This is the path used to propose replacement tags for those on the tagging path which have been identified as potential errors.

where:

`FN` specifies that no error flagging is to be attempted.

`FP` selects path based flagging: path probabilities are compared to determine if a path contains errors.

`FS` selects word based flagging: maximum likelihood tag probabilities (taken in the context of the entire word sequence) are compared to determine if each tag assignment is erroneous.

`ML` is the path of maximum likelihood tags for each word.

`NL` is the path of next maximum likelihood tags for each word.

`P1` is the path with the highest path probability.

`P2` is the path with the next highest path probability.

`TT` is the total sequence probability

`PX` is a path which differs from the tagging path at every ambiguous word.

-r Set the error flagging threshold to *th*. Default is 1.0. This value is compared with the ratio of the tagging to the comparison paths (either the ratio of path or tag probabilities, depending on the method of error flagging); if the ratio is less than *th*, a potential error is deemed to have occurred (although no error

will be flagged unless the alternate path differs from the tagging path at that point).

C.3.8 Valhmm

Create a report on the contents of an HMM and validate it by computing sums of probabilities.

C.3.8.1 Syntax

`valhmm [-s] [`*src* `[`*dst*`]]`

C.3.8.2 Parameters

src Input HMM file (or stdin).

dst Output report file (or stdout). The report consists of the following items:

Statistics file The statistics file used by estm to create the HMM.

Estimator a coded description of the estimator used to create the HMM. The first part of the code is a representation of the parameters to estm, in the following format:

o/f/i/t/l/r

where *o*, *f*, *i*, *t* and *l* are the selected order, frequency modifier, and initial, transition and lexical probability estimators as described in section C.3.2.2. *r* is the value of the combined order threshold, also described in that section.

The second part of the code describes any reestimation passes made on the original HMM. There may be any number of segments to this part, each of which represents one run of the program reestm, in the following format:

+itx,i/t/l,err,ppx

where *it* is the number of iterations made; *i*, *t* and *l* describe the reestimation for initial, transition and lexical probabilities—this is either the

corresponding floor threshold value, or `fix`, to indicate that no reestimation was applied; *err* is the reestimation error; and *ppx* is the sequence perplexity. See section C.3.4 for a description of these parameters.

Number of tags The number of tags in the vocabulary.

Number of words The number of words in the vocabulary.

Number of states The number of states for the HMM.

Number of symbols The number of symbols for the HMM.

Initial state The sum over states of the initial probability for each state (this should equal 1).

State transition and output sums A list of states, with two sums pertaining to each state:

state transition sum the sum over all states of the probability of a transition from the current state (this should equal 1).

output sum the sum over all symbols of the output probability from the current state (should also equal 1).

`-s` Limit the report to a brief summary of the HMM's contents, which excludes the probability sums.

C.4 Design

Ytag is programmed in C and is based on a modular design. Modules are of three types: program modules, of which there is one for each program in the system; utility modules, which provide some basic service and may be used by several program modules; and library modules, which implement an abstract data type such as a list. The latter are not specific to ytag and are documented elsewhere.

Program modules correspond directly to source files with a *.c* extension; there is one for each program described above: coll.c, estm.c, perf.c, reestm.c, retag.c, rstats.c, tag.c and valhmm.c.

Each utility module has two parts: a *.c* file which contains type, function and variable definitions and constitutes the implementation portion of the module; and

a *.h* file which contains declarations of exported types, functions and variables and constitutes the interface portion of the module. Modules which use a utility module must include its *.h* file. The interface for each utility module is extensively commented to allow it to be used without the need for the user to be familiar with the details of the implementation. The utility modules in ytag are the following:

collect Representation of frequency statistics from a lexed file and routines for counting, reading and writing them.

hmm Representation of a generic HMM with provision for efficient non-ergodic models; all basic operations on HMMs are supported, as well as routines for reading and writing them.

hmmt Representation of a tagging HMM. This module is an interface between the generic routines in *hmm*, which deal with states and symbols, and the programs in ytag which deal with tags and words.

lexio Routines to read and write lexed and tagged corpora.

There is also one other *.h* file which is not part of a module. This is the file *params.h* which contains various size parameters and can be used to customize ytag for a specific tag set and vocabulary.

Appendix D

Category Sets

D.1 Categories for the Hans Corpus

There are 31 categories in this set, which includes the following punctuation tags:

```
! " $ % & ( ) * + , -- . ... / : ; < ?
```

and the following grammatical tags:

```
AJ - adjective
AV - adverb
AX - affix
CC - coordinating conjunction
CS - subordinating conjunction
DT - determiner
IJ - interjection
NC - common noun
NN - numeral
NP - proper noun
OD - ordinal
PN - pronoun
PP - preposition
QN - quantifier
VB - verb
```

X - unknown

D.2 Categories for the LOB Corpus

The following 131 categories were used for the LOB. See [38] for a description of these tags.

```
! &FO &FW ( ) *' **' *- , . ... : ; ? ABL ABN ABX AP AP" AP$
APS AT ATI BE BED BEDZ BEG BEM BEN BER BEZ CC CC" CD CD-CD CD1
CD1$ CD1S CDS CS CS" DO DOD DOZ DT DTI DTS DTX EX HV HVD HVG
HVN HVZ IN IN" JJ JJB JJR JJT JNP MD NC NN NN$ NNP NNP$ NNPS
NNS NNS$ NNU NNU" NNUS NP NP$ NPL NPL$ NPLS NPS NPS$ NPT NPT$
NPTS NR NR$ NRS OD PN PN" PN$ PP$ PP$$ PP1A PP1AS PP10 PP10S
PP2 PP3 PP3A PP3AS PP30 PP30S PPL PPLS PPLS" QL QLP RB RB" RBR
RBT RI RN RP TO TO" UH VB VBD VBG VBN VBZ WDT WDTR WP WP$R
WPOR WPR WRB XNOT ZZ
```

D.3 Categories for the LOB/s Corpus

Each tag in the LOB corpus was mapped to one of the 41 tags in the following set to create the LOB/s corpus. In most cases, the mapping was accomplished by simply truncating the LOB tag after the first two characters.

```
! &FO &FW ( ) *' **' *- , . ... : ; ? AB AP AT BE CC CD CS DO
DT EX HV IN JJ MD NC NN OD PN PP QL RB TO UH VB WH XN ZZ
```

Bibliography

[1] Eric Steven Atwell. Grammatical analysis of english by statistical pattern recognition. In *Proceedings of the 4th International Conference on Pattern Recognition*, pages 626–635, Cambridge, UK, 1988. Springer-Verlag.

[2] L. Bahl, Brown P., de Souza P., and Mercer R. A tree-based statistical language model for natural language speech recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-37:1001–1008, July 1989.

[3] Lalit R. Bahl, Frederick Jelinek, and Robert L. Mercer. A maximum likelihood approach to continuous speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2):179–191, March 1983.

[4] L.R. Bahl, J.K Baker, F. Jelinek, and R.L Mercer. Perplexity: a measure of difficulty of speech recognition tasks. In *94th Meeting of the Acoustical Society of America*, Miami, December 1977.

[5] L.E. Baum, T. Petrie, Soles G., and Weiss N. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *Annals of Mathematical Statistics*, 41:164–171, 1970.

[6] E. Black, J. Cocke, T Fujisaki, and J. Jelinek. Probabilistic parsing method for sentence disambiguation. In *Proceedings of The International Workshop on Parsing Technologies.* Carnegie Mellon University, 1989.

[7] Leonard Bolc, editor. *Natural Language Parsing Systems.* Springer-Verlag, 1987.

[8] Peter F. Brown, John Cocke, Stephan A. Della Pietra, Vincent J. Della Pietra, Frederick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roosin. A statistical approach to machine translation. *Computational Linguistics*, 16(2):79–85, June 1990.

[9] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. Word sense disambiguation using statistical methods. *???*, 1991.

[10] Peter F. Brown, Vincent J. Della Pietra, Peter V. deSouza, and Jenifer C. Lai. Class-based n-gram models of natural language. *???*, December 1990.

[11] Peter F. Brown, Jennifer C. Lai, and Robert L. Mercer. Aligning sentences in parallel corpora. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, Berkeley, CA, 1991.

[12] Roy J. Byrd and Evelyne Tzoukermann. Adapting an english morphological analyzer for french. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*, Buffalo, NY, June 1988.

[13] K. Church. A stochastic parts program and noun phrase parser for unrestricted text. In *2nd Conference on Applied Natural Language Processing*, Austin, Texas, 1988.

[14] K. Church. Text analysis. In Mellish, editor, *Encyclopedia of Language and Linguistics*. Pergamon Press, Aberdeen University Press, 1991.

[15] K. Church and W. Gale. A comparison of the enhanced good-turing and deleted estimation methods for estimating probabilities of english bigrams. *Computer Speech and Language*, 5(1), 1991.

[16] K. Church and W. Gale. Probability scoring for spelling correction. *Statistics and Computing*, 1991.

[17] K. Church, P. Hanks, D. Hindle, and W. Gale. Using statistics in lexical analysis. In Zernik, editor, *Lexical Acquisition: Using on-line Resources to Build a Lexicon*. Lawrence Erlbaum, 1991.

[18] K. Church, P. Hanks, D. Hindle, W. Gale, and R. Moon. Substitutability. In Atkins and Zampolli, editors, *Computational Approaches to the Lexicon: Automating the Lexicon II Schema.* Oxford University Press, 1991.

[19] Kenneth W. Church and William A. Gale. Identifying word correspondences in parallel texts. In *Proceedings of the DARPA Workshop*, 1991.

[20] Kenneth W. Church and William A. Gale. A program for aligning sentences in bilingual corpora. Draft, 1991.

[21] K.W Church and W. A Gale. *Estimation Procedures for Language Context: Poor Estimates are Worse than None*, pages 69–74. Physica-Verlag, Heidelberg, 1990.

[22] R.A. Code, editor. *Perception and Production of Fluent Speech.* Erlbaum, 1980.

[23] A. Corazza, R. DeMori, R. Gretter, and G. Satta. Computation of probabilities for an island-driven parser. Technical Report SOCS 90.19, McGill University, Montreal, January 1991.

[24] David Crystal. *Dictionary of Linguistics and Phonetics.* Basil Blackwell Ltd, Oxford, 1985.

[25] F.J. Damerau, E. Mays, and R.L. Mercer. Context based spelling correction. In *Proceedings of the IBM Natural Language ITL*, Paris, France, March 1990. IBM.

[26] de Marcken. Parsing the lob corpus. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, pages 243–251. ACL, 1990.

[27] Renato De Mori and Roland Kuhn. A cache-based natural language model for speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):570–583, June 1990.

[28] S. DeRose. Grammatical category disambiguation by statistical optimization. *Computational Linguistics*, 14(1), 1988.

[29] A-M. Derouault and B. Merialdo. Natural language modeling for phoneme-to-text transcription. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):742–743, November 1986.

[30] Anne-Marie Derouault and Marc Elbeze. A morphological model for large vocabulary speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 577–580, Albuquerque, 1990.

[31] P. Dumouchel, V. Gupta, M Lennig, and P. Mermelstein. Three probabilistic language models for a large vocabulary speech recognizer. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 513–516, New York, 1988.

[32] I.J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3):237–264, 1953.

[33] B.B Greene and G.M Rubin. *Automatic Grammatical Tagging of English.* Brown University, Providence, R.I., 1971.

[34] D. Hindle. Acquiring disambiguation rules from text. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 118–125, Vancouver, 1989. ACL.

[35] PROGICIELS Bourbeau-Pinard Inc. Dictionnaire morphologique du français.

[36] Hajic J., G. Russell, and Warwick S. Searching on tagged corpora: Linguistically motivated concordance analysis. In *Proceedings of the 6th Annual Conference of the UW Centre for the New Oxford English Dictionary*, pages 10–18, Waterloo, 1990.

[37] F. Jelinek and R.L. Mercer. Interpolated estimation of markov source parameters from sparse data. In E.S. Gelsema and L.N. Kanal, editors, *Pattern Recognition in Practice.* North-Holland, Amsterdam, 1980.

[38] Stig Johansson. *The Tagged LOB Corpus User's Manual.* Knut Hofland, Bergen, 1986.

[39] Church K. and Hanks P. Word asssociation norms, mutual information and lexicography. *Computational Linguistics*, 16(1):22–30, March 1990.

[40] Slava M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-35(3):400–401, March 1987.

[41] S. Kline and R.F. Simmons. A computational approach to the grammatical coding of english. *Journal of the ACM*, 10:334–347, 1963.

[42] Julian Kupiec. Probabilistic models of short and long distance word dependencies in running text. In *Proceedings, Speech and Natural Language Workshop*, pages 290–295, Philadelphia, February 1989. DARPA.

[43] W.A. Lea, editor. *Trends in Speech Recognition.* Prentice Hall, 1980.

[44] B. Merialdo. Multilevel decoding for very-large-size dictionary speech recognition. *IBM Journal of Research and Development*, 32(2):227–237, March 1988.

[45] B. Merialdo. Tagging text with a probabilistic model. In *Proceedings of the IBM Natural Language ITL*, pages 161–172, Paris, France, March 1990. IBM.

[46] Arthur Nádas. A decision theoretic formulation of a training problm in speech recognition and a comparison of training by unconditional versus conditional maximum likelihood. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-31(4):814–817, August 1983.

[47] Arthur Nádas. Estimation of probabilities in the language model of the ibm speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-32(4):859–861, August 1984.

[48] Arthur Nádas. On turing's formula for word probabilities. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-33(6):1415–1417, December 1985.

[49] Garside R., Leech G., and Sampson J., editors. *The Computational Analysis of English.* Longman, 1987.

[50] L.R. Rabiner and B.H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, pages 4–16, January 1986.

[51] S. Seneff. Tina: A probabilistic syntactic parser for speech understanding systems. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 711–715, Albuquerque, 1989.

[52] K. Shikano. Improvement of word recognition results by trigram model. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 1987.

[53] A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, IT-13(4):260–269, April 1967.