A Practical Part-of-Speech Tagger

Doug Cutting and **Julian Kupiec** and **Jan Pedersen** and **Penelope Sibun**
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304, USA

Abstract

We present an implementation of a part-of-speech tagger based on a hidden Markov model. The methodology enables robust and accurate tagging with few resource requirements. Only a lexicon and some unlabeled training text are required. Accuracy exceeds 96%. We describe implementation strategies and optimizations which result in high-speed operation. Three applications for tagging are described: phrase recognition; word sense disambiguation; and grammatical function assignment.

1 Desiderata

Many words are ambiguous in their part of speech. For example, "tag" can be a noun or a verb. However, when a word appears in the context of other words, the ambiguity is often reduced: in "a tag is a part-of-speech label," the word "tag" can only be a noun. A *part-of-speech tagger* is a system that uses context to assign parts of speech to words.

Automatic text tagging is an important first step in discovering the linguistic structure of large text corpora. Part-of-speech information facilitates higher-level analysis, such as recognizing noun phrases and other patterns in text.

For a tagger to function as a practical component in a language processing system, we believe that a tagger must be:

Robust Text corpora contain ungrammatical constructions, isolated phrases (such as titles), and nonlinguistic data (such as tables). Corpora are also likely to contain words that are unknown to the tagger. It is desirable that a tagger deal gracefully with these situations.

Efficient If a tagger is to be used to analyze arbitrarily large corpora, it must be efficient—performing in time linear in the number of words tagged. Any training required should also be fast, enabling rapid turnaround with new corpora and new text genres.

Accurate A tagger should attempt to assign the correct part-of-speech tag to every word encountered.

Tunable A tagger should be able to take advantage of linguistic insights. One should be able to correct systematic errors by supplying appropriate *a priori* "hints." It should be possible to give different hints for different corpora.

Reusable The effort required to retarget a tagger to new corpora, new tagsets, and new languages should be minimal.

2 Methodology

2.1 Background

Several different approaches have been used for building text taggers. Greene and Rubin used a rule-based approach in the TAGGIT program [Greene and Rubin, 1971], which was an aid in tagging the Brown corpus [Francis and Kučera, 1982]. TAGGIT disambiguated 77% of the corpus; the rest was done manually over a period of several years. More recently, Koskenniemi also used a rule-based approach implemented with finite-state machines [Koskenniemi, 1990].

Statistical methods have also been used (e.g., [DeRose, 1988], [Garside *et al.*, 1987]). These provide the capability of resolving ambiguity on the basis of most likely interpretation. A form of Markov model has been widely used that assumes that a word depends probabilistically on just its part-of-speech category, which in turn depends solely on the categories of the preceding two words.

Two types of training (i.e., parameter estimation) have been used with this model. The first makes use of a tagged training corpus. Derouault and Merialdo use a bootstrap method for training [Derouault and Merialdo, 1986]. At first, a relatively small amount of text is manually tagged and used to train a partially accurate model. The model is then used to tag more text, and the tags are manually corrected and then used to retrain the model. Church uses the tagged Brown corpus for training [Church, 1988]. These models involve probabilities for each word in the lexicon, so large tagged corpora are required for reliable estimation.

The second method of training does not require a tagged training corpus. In this situation the Baum-Welch algorithm (also known as the forward-backward algorithm) can be used [Baum, 1972]. Under this regime the model is called a *hidden Markov model* (HMM), as state transitions (i.e., part-of-speech categories) are assumed to be unobservable. Jelinek has used this method for training a text tagger [Jelinek, 1985]. Parameter smoothing can be conveniently achieved using the method of *deleted interpolation* in which weighted estimates are taken from second- and first-order models and a uniform probability distribution [Jelinek and Mercer, 1980]. Kupiec used word equivalence classes (referred to here as *ambiguity classes*) based on parts of speech, to pool data from individual words [Kupiec, 1989b]. The most common words are still represented individually, as sufficient data exist for robust estimation.

However all other words are represented according to the set of possible categories they can assume. In this manner, the vocabulary of 50,000 words in the Brown corpus can be reduced to approximately 400 distinct ambiguity classes [Kupiec, 1992]. To further reduce the number of parameters, a first-order model can be employed (this assumes that a word's category depends only on the immediately preceding word's category). In [Kupiec, 1989a], networks are used to selectively augment the context in a basic first-order model, rather than using uniformly second-order dependencies.

2.2 Our approach

We next describe how our choice of techniques satisfies the criteria listed in section 1. The use of an HMM permits complete flexibility in the choice of training corpora. Text from any desired domain can be used, and a tagger can be tailored for use with a particular text database by training on a portion of that database. Lexicons containing alternative tag sets can be easily accommodated without any need for re-labeling the training corpus, affording further flexibility in the use of specialized tags. As the resources required are simply a lexicon and a suitably large sample of ordinary text, taggers can be built with minimal effort, even for other languages, such as French (e.g., [Kupiec, 1992]). The use of ambiguity classes and a first-order model reduces the number of parameters to be estimated without significant reduction in accuracy (discussed in section 5). This also enables a tagger to be reliably trained using only moderate amounts of text. We have produced reasonable results training on as few as 3,000 sentences. Fewer parameters also reduce the time required for training. Relatively few ambiguity classes are sufficient for wide coverage, so it is unlikely that adding new words to the lexicon requires retraining, as their ambiguity classes are already accommodated. Vocabulary independence is achieved by predicting categories for words not in the lexicon, using both context and suffix information. Probabilities corresponding to category sequences that never occurred in the training data are assigned small, non-zero values, ensuring that the model will accept any sequence of tokens, while still providing the most likely tagging. By using the fact that words are typically associated with only a few part-of-speech categories, and carefully ordering the computation, the algorithms have linear complexity (section 3.3).

3 Hidden Markov Modeling

The hidden Markov modeling component of our tagger is implemented as an independent module following the specification given in [Levinson *et al.*, 1983], with special attention to space and time efficiency issues. Only first-order modeling is addressed and will be presumed for the remainder of this discussion.

3.1 Formalism

In brief, an HMM is a doubly stochastic process that generates sequence of symbols

$$S = \{S_1, S_2, \ldots, S_T\},\ S_i \in W \ \ 1 \leq i \leq T,$$

where W is some finite set of possible symbols, by composing an underlying Markov process with a state-dependent symbol generator (i.e., a Markov process with noise).[1] The Markov process captures the notion of sequence dependency and is described by a set of N states, a matrix of transition probabilities $A = \{a_{ij}\}$ $1 \leq i, j \leq N$ where a_{ij} is the probability of moving from state i to state j, and a vector of initial probabilities $\Pi = \{\pi_i\}$ $1 \leq i \leq N$ where π_i is the probability of starting in state i. The symbol generator is a state-dependent measure on V described by a matrix of symbol probabilities $B = \{b_{jk}\}$ $1 \leq j \leq N$ and $1 \leq k \leq M$ where $M = |W|$ and b_{jk} is the probability of generating symbol s_k given that the Markov process is in state j.[2]

In part-of-speech tagging, we will model word order dependency through an underlying Markov process that operates in terms of lexical tags, yet we will only be able to observe the sets of tags, or ambiguity classes, that are possible for individual words. The ambiguity class of each word is the set of its permitted parts of speech, only one of which is correct in context. Given the parameters A, B and Π, hidden Markov modeling allows us to compute the most probable sequence of state transitions, and hence the mostly likely sequence of lexical tags, corresponding to a sequence of ambiguity classes. In the following, N can be identified with the number of possible tags, and W with the set of all ambiguity classes.

Applying an HMM consists of two tasks: estimating the model parameters A, B and Π from a training set; and computing the most likely sequence of underlying state transitions given new observations. Maximum likelihood estimates (that is, estimates that maximize the probability of the training set) can be found through application of alternating expectation in a procedure known as the Baum-Welch, or forward-backward, algorithm [Baum, 1972]. It proceeds by recursively defining two sets of probabilities: the forward probabilities,

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\right] b_j(S_{t+1}) \qquad 1 \leq t \leq T-1, \quad (1)$$

where $\alpha_1(i) = \pi_i b_i(S_1)$ for all i; and the backward probabilities,

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(S_{t+1}) \beta_{t+1}(j) \qquad T-1 \leq t \leq 1, \quad (2)$$

where $\beta_T(j) = 1$ for all j. The forward probability $\alpha_t(i)$ is the joint probability of the sequence up to time t, $\{S_1, S_2, \ldots, S_t\}$, and the event that the Markov process is in state i at time t. Similarly, the backward probability $\beta_t(j)$ is the probability of seeing the sequence $\{S_{t+1}, S_{t+2}, \ldots, S_T\}$ given that the Markov process is at state i at time t. It follows that the probability of the entire sequence is

$$P = \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_t(i) a_{ij} b_j(S_{t+1}) \beta_{t+1}(j)$$

for any t in the range $1 \leq t \leq T-1$.[3]

[1] For an introduction to hidden Markov modeling see [Rabiner and Juang, 1986].

[2] In the following we will write $b_j(S_t)$ for b_{jk} if $S_t = s_k$.

[3] This is most conveniently evaluated at $t = T-1$, in which case $P = \sum_{i=1}^{N} \alpha_T(i)$

Given an initial choice for the parameters A, B, and Π the expected number of transitions, γ_{ij}, from state i to state j conditioned on the observation sequence S may be computed as follows:

$$\gamma_{ij} = \frac{1}{P}\sum_{t=1}^{T-1} \alpha_t(i) a_{ij} b_j(S_{t+1}) \beta_{t+1}(j).$$

Hence we can estimate a_{ij} by:

$$\hat{a}_{ij} = \frac{\gamma_{ij}}{\sum_{j=1}^{N} \gamma_{ij}} = \frac{\sum_{t=1}^{T-1} \alpha_t(i) a_{ij} b_j(S_{t+1}) \beta_{t+1}(j)}{\sum_{t=1}^{T-1} \alpha_t(i)\beta_t(i)}. \tag{3}$$

Similarly, b_{jk} and π_i can be estimated as follows:

$$\hat{b}_{jk} = \frac{\sum_{t \ni S_t = s_k} \alpha_t(j)\beta_t(j)}{\sum_{t=1}^{T} \alpha_t(j)\beta_t(j)} \tag{4}$$

and

$$\hat{\pi}_i = \frac{1}{P}\alpha_1(i)\beta_1(i). \tag{5}$$

In summary, to find maximum likelihood estimates for A, B, and Π, via the Baum-Welch algorithm, one chooses some starting values, applies equations 3–5 to compute new values, and then iterates until convergence. It can be shown that this algorithm will converge, although possibly to a non-global maximum [Baum, 1972].

Once a model has been estimated, selecting the most likely underlying sequence of state transitions corresponding to an observation S can be thought of as a maximization over all sequences that might generate S. An efficient dynamic programming procedure, known as the Viterbi algorithm [Viterbi, 1967], arranges for this computation to proceed in time proportional to T. Suppose $V = \{v(t)\}$ $1 \le t \le T$ is a state sequence that generates S, then the probability that V generates S is,

$$P(v) = \pi_{v(1)} b_{v(1)}(S_1) \prod_{t=2}^{T} a_{v(t-1)v(t)} b_{v(t)}(S_t).$$

To find the most probable such sequence we start by defining $\phi_1(i) = \pi_i b_i(S_1)$ for $1 \le i \le N$ and then perform the recursion

$$\phi_t(j) = \max_{1 \le i \le N} [\phi_{t-1}(i) a_{ij}] b_j(S_t) \tag{6}$$

and

$$\psi_t(j) = \max_{1 \le i \le N}{}^{-1} \phi_{t-1}(i)$$

for $2 \le t \le T$ and $1 \le j \le N$. The crucial observation is that for each time t and each state i one need only consider the most probable sequence arriving at state i at time t. The probability of the most probable sequence is $\max_{1 \le i \le N}[\phi_T(i)]$ while the sequence itself can be reconstructed by defining $v(T) = \max^{-1}_{1 \le i \le N} \phi_T(i)$ and $v(t-1) = \psi_t(q_t)$ for $T \ge t \ge 2$.

3.2 Numerical Stability

The Baum-Welch algorithm (equations 1–5) and the Viterbi algorithm (equation 6) involve operations on products of numbers constrained to be between 0 and 1. Since these products can easily underflow, measures must be taken to rescale. One approach premultiplies the α and β probabilities with an accumulating product depending on t [Levinson *et al.*, 1983]. Let $\tilde{\alpha}_1(i) = \alpha_1(i)$ and define

$$c_t = \left[\sum_{i=1}^{N} \tilde{\alpha}_t(i)\right]^{-1} \qquad 1 \le t \le T.$$

Now define $\hat{\alpha}_t(i) = c_t \tilde{\alpha}_t(i)$ and use $\hat{\alpha}$ in place of α in equation 1 to define $\tilde{\alpha}$ for the next iteration:

$$\tilde{\alpha}_{t+1}(j) = \left[\sum_{i=1}^{N} \hat{\alpha}_t(i) a_{ij}\right] b_j(S_{t+1}) \qquad 1 \le t \le T-1.$$

Note that $\sum_{i=1}^{n} \hat{\alpha}_t(i) = 1$ for $1 \le t \le T$. Similarly, let $\hat{\beta}_T(i) = \beta_T(i)$ and define $\tilde{\beta}_t(i) = c_t \hat{\beta}_t(i)$ for $T \ge t \ge 1$ where

$$\hat{\beta}_t(i) = \sum_{j=1}^{N} a_{ij} b_j(S_{t+1}) \tilde{\beta}_{t+1}(j) \qquad T-1 \le t \le 1.$$

The scaled backward and forward probabilities, $\hat{\alpha}$ and $\hat{\beta}$, can be exchanged for the unscaled probabilities in equations 3–5 without affecting the value of the ratios. To see this, note that $\hat{\alpha}_t(i) = C_1^t \alpha_t(i)$ and $\hat{\beta}_t(i) = \beta_t(i) C_{t+1}^T$ where

$$C_i^j = \prod_{t=i}^{j} c_t.$$

Now, in terms of the scaled probabilities, equation 5, for example, can be seen to be unchanged:

$$\frac{\hat{\alpha}_1(i)\hat{\beta}_1(i)}{\sum_{i=1}^{N} \hat{\alpha}_T(i)} = \frac{C_1^1 \alpha_1(i)\beta_1(i) C_2^T}{\sum_{i=1}^{N} C_1^T \alpha_T(i)} = \hat{\pi}_i.$$

A slight difficulty occurs in equation 3 that can be cured by the addition of a new term, c_{t+1}, in each product of the upper sum:

$$\frac{\sum_{t=1}^{T-1} \hat{\alpha}_t(i) a_{ij} b_j(S_{t+1}) \hat{\beta}_{t+1}(j) c_{t+1}}{\sum_{t=1}^{T-1} \hat{\alpha}_t(i)\hat{\beta}_t(i)} = \hat{a}_{ij}.$$

Numerical instability in the Viterbi algorithm can be ameliorated by operating on a logarithmic scale [Levinson *et al.*, 1983]. That is, one maximizes the log probability of each sequence of state transitions,

$$\begin{aligned}\log(P(v)) \quad &= \quad \log(\pi_{v(1)}) + \log(b_{v(1)}(S_1)) + \\ &\sum_{t=2}^{T} \log(a_{v(t-1)v(t)}) + \log(b_{v(t)}(S_t)).\end{aligned}$$

Hence, equation 6 is replaced by

$$\phi_t(j) = \max_{1 \le i \le N} [\phi_{t-1}(i) + \log(a_{ij})] + \log b_j(S_t).$$

Care must be taken with zero probabilities. However, this can be elegantly handled through the use of IEEE negative infinity [P754, 1981].

3.3 Reducing Time Complexity

As can be seen from equations 1–5, the time cost of training is $O(TN^2)$. Similarly, as given in equation 6, the Viterbi algorithm is also $O(TN^2)$. However, in part-of-speech tagging, the problem structure dictates that the matrix of symbol probabilities B is sparsely populated. That is, $b_{ij} \neq 0$ iff the ambiguity class corresponding to symbol j includes the part-of-speech tag associated with state i. In practice, the degree of overlap between ambiguity classes is relatively low; some tokens are assigned unique tags, and hence have only one non-zero symbol probability.

The sparseness of B leads one to consider restructuring equations 1–6 so a check for zero symbol probability can obviate the need for further computation. Equation 1 is already conveniently factored so that the dependence on $b_j(S_{t+1})$ is outside the inner sum. Hence, if k is the average number of non-zero entries in each row of B, the cost of computing equation 1 can be reduced to $O(kTN)$.

Equations 2–4 can be similarly reduced by switching the order of iteration. For example, in equation 2, rather than for a given t computing $\beta_t(i)$ for each i one at a time, one can accumulate terms for all i in parallel. The net effect of this rewriting is to place a $b_j(S_{t+1}) = 0$ check outside the innermost iteration. Equations 3 and 4 submit to a similar approach. Equation 5 is already only $O(N)$. Hence, the overall cost of training can be reduced to $O(kTN)$, which, in our experience, amounts to an order of magnitude speedup.[4]

The time complexity of the Viterbi algorithm can also be reduced to $O(kTN)$ by noting that $b_j(S_t)$ can be factored out of the maximization of equation 6.

3.4 Controlling Space Complexity

Adding up the sizes of the probability matrices A, B, and Π, it is easy to see that the storage cost for directly representing one model is proportional to $N(N + M + 1)$. Running the Baum-Welch algorithm requires storage for the sequence of observations, the α and β probabilities, the vector $\{c_i\}$, and copies of the A and B matrices (since the originals cannot be overwritten until the end of each iteration). Hence, the grand total of space required for training is proportional to $T + 2N(T + N + M + 1)$.

Since N and M are fixed by the model, the only parameter that can be varied to reduce storage costs is T. Now, adequate training requires processing from tens of thousands to hundreds of thousands of tokens [Kupiec, 1989a]. The training set can be considered one long sequence, it which case T is very large indeed, or it can be broken up into a number of smaller sequences at convenient boundaries. In first-order hidden Markov modeling, the stochastic process effectively restarts at unambiguous tokens, such as sentence and paragraph markers, hence these tokens are convenient points at which to break the training set. If the Baum-Welch algorithm is run separately (from the same starting point) on each piece, the resulting trained models must be recombined in some way. One obvious approach is simply to average. However, this fails if any two states are indistinguishable (in the sense that they had the same transition probabilities and the same symbol probabilities at start), because states are then not matched across trained models. It is therefore important that each state have a distinguished role, which is relatively easy to achieve in part-of-speech tagging.

Our implementation of the Baum-Welch algorithm breaks up the input into fixed-sized pieces of training text. The Baum-Welch algorithm is then run separately on each piece and the results are averaged together.

Running the Viterbi algorithm requires storage for the sequence of observations, a vector of current maxes, a scratch array of the same size, and a matrix of ψ indices, for a total proportional to $T + N(2 + T)$ and a grand total (including the model) of $T + N(N + M + T + 3)$. Again, N and M are fixed. However, T need not be longer than a single sentence, since, as was observed above, the HMM, and hence the Viterbi algorithm, restarts at sentence boundaries.

3.5 Model Tuning

An HMM for part-of-speech tagging can be tuned in a variety of ways. First, the choice of tagset and lexicon determines the initial model. Second, empirical and *a priori* information can influence the choice of starting values for the Baum-Welch algorithm. For example, counting instances of ambiguity classes in running text allows one to assign non-uniform starting probabilities in A for a particular tag's realization as a particular ambiguity class. Alternatively, one can state *a priori* that a particular ambiguity class is most likely to be the reflection of some subset of its component tags. For example, if an ambiguity class consisting of the open class tags is used for unknown words, one may encode the fact that most unknown words are nouns or proper nouns by biasing the initial probabilities in B.

Another biasing of starting values can arises from noting that some tags are unlikely to be followed by others. For example, the lexical item "to" maps to an ambiguity class containing two tags, *infinitive-marker* and *to-as-preposition*, neither of which occurs in any other ambiguity class. If nothing more were stated, the HMM would have two states which were indistinguishable. This can be remedied by setting the initial transition probabilities from *infinitive-marker* to strongly favor transitions to such states as *verb-uninflected* and *adverb*.

Our implementation allows for two sorts of biasing of starting values: ambiguity classes can be annotated with favored tags; and states can be annotated with favored transitions. These biases may be specified either as sets or as set complements. Biases are implemented by replacing the disfavored probabilities with a small constant (machine epsilon) and redistributing mass to the other possibilities. This has the effect of disfavoring the indicated outcomes without disallowing them; sufficient converse data can rehabilitate these values.

4 Architecture

In support of this and other work, we have developed a system architecture for text access [Cutting *et al.*, 1991]. This architecture defines five components for such systems:

[4] An equivalent approach maintains a mapping from states i to non-zero symbol probabilities and simply avoids, in the inner iteration, computing products which must be zero [Kupiec, 1992].

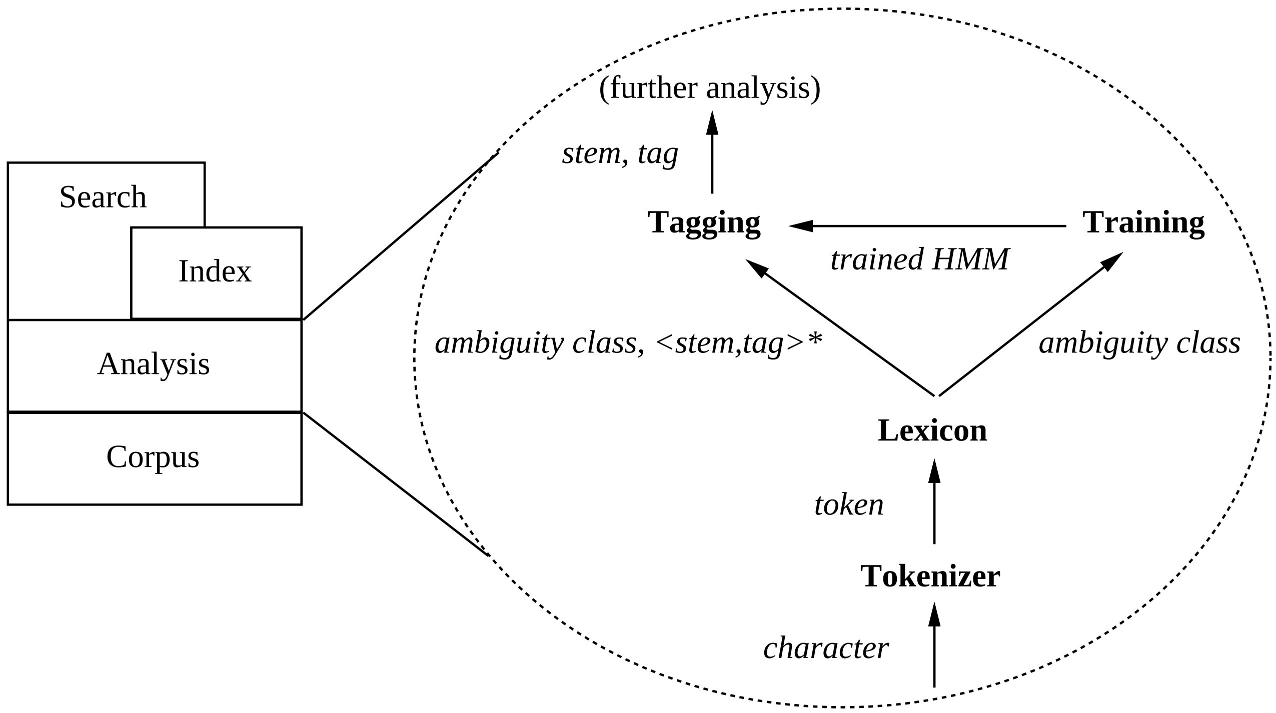

Figure 1: Tagger Modules in System Context

corpus, which provides text in a generic manner; *analysis*, which extracts *terms* from the text; *index* which stores term occurrence statistics; and *search*, which utilizes these statistics to resolve queries.

The part-of-speech tagger described here is implemented as an analysis module. Figure 1 illustrates the overall architecture, showing the tagger analysis implementation in detail. The tagger itself has a modular architecture, isolating behind standard protocols those elements which may vary, enabling easy substitution of alternate implementations.

Also illustrated here are the data types which flow between tagger components. As an analysis implementation, the tagger must generate terms from text. In this context, a term is a word stem annotated with part of speech.

Text enters the analysis sub-system where the first processing module it encounters is the *tokenizer*, whose duty is to convert text (a sequence of characters) into a sequence of *tokens*. Sentence boundaries are also identified by the tokenizer and are passed as reserved tokens.

The tokenizer subsequently passes tokens to the *lexicon*. Here tokens are converted into a set of stems, each annotated with a part-of-speech tag. The set of tags identifies an *ambiguity class*. The identification of these classes is also the responsibility of the lexicon. Thus the lexicon delivers a set of stems paired with tags, and an ambiguity class.

The *training* module takes long sequences of ambiguity classes as input. It uses the Baum-Welch algorithm to produce a *trained HMM*, an input to the tagging module. Training is typically performed on a sample of the corpus at hand, with the trained HMM being saved for subsequent use on the corpus at large.

The *tagging* module buffers sequences of ambiguity classes between sentence boundaries. These sequences are disambiguated by computing the maximal path through the HMM with the Viterbi algorithm. Operating at sentence granularity provides fast throughput without loss of accuracy, as sentence boundaries are unambiguous. The resulting sequence of tags is used to select the appropriate stems. Pairs of stems and tags are subsequently emitted.

The tagger may function as a complete analysis component, providing tagged text to search and indexing components, or as a sub-system of a more elaborate analysis, such as phrase recognition.

4.1 Tokenizer Implementation

The problem of tokenization has been well addressed by much work in compilation of programming languages. The accepted approach is to specify token classes with regular expressions. These may be compiled into a single deterministic finite state automaton which partitions character streams into labeled tokens [Aho *et al.*, 1986, Lesk, 1975].

In the context of tagging, we require at least two token classes: sentence boundary and word. Other classes may include numbers, paragraph boundaries and various sorts of punctuation (e.g., braces of various types, commas). However, for simplicity, we will henceforth assume only words and sentence boundaries are extracted.

Just as with programming languages, with text it is not always possible to unambiguously specify the required token classes with regular expressions. However the addition of a simple *lookahead* mechanism which allows specification of right context ameliorates this [Aho *et al.*, 1986, Lesk, 1975]. For example, a sentence boundary in English text might be identified by a period, followed by whitespace, followed by an uppercase letter. However the uppercase letter must not be consumed, as it is the first component of the next token. A lookahead mechanism allows us to specify in the sentence-boundary regular expression that the final character matched should not be considered a part of the token.

This method meets our stated goals for the overall system. It is efficient, requiring that each character be examined only once (modulo lookahead). It is easily parameterizable, providing the expressive power to concisely define accurate and robust token classes.

4.2 Lexicon Implementation

The lexicon module is responsible for enumerating parts of speech and their associated stems for each word it is given. For the English word "does," the lexicon might return "do, *verb*" and "doe, *plural-noun*." It is also responsible for identifying ambiguity classes based upon sets of tags.

We have employed a three-stage implementation:

First, we consult a manually-constructed lexicon to find stems and parts of speech. Exhaustive lexicons of this sort are expensive, if not impossible, to produce. Fortunately, a small set of words accounts for the vast majority of word occurences. Thus high coverage can be obtained without prohibitive effort.

Words not found in the manually constructed lexicon are generally both open class and regularly inflected. As a second stage, a language-specific method can be employed to guess ambiguity classes for unknown words. For many languages (e.g., English and French), word suffixes provide strong cues to words' possible categories. Probabalistic predictions of a word's category can be made by analyzing suffixes in untagged text [Kupiec, 1992, Meteer *et al.*, 1991].

As a final stage, if a word is not in the manually constructed lexicon, and its suffix is not recognized, a default ambiguity class is used. This class typically contains all the open class categories in the language.

Dictionaries and suffix tables are both efficiently implementable as letter trees, or *tries* [Knuth, 1973], which require that each character of a word be examined only once during a lookup.

5 Performance

In this section, we detail how our tagger meets the desiderata that we outlined in section 1.

5.1 Efficient

The system is implemented in Common Lisp [Steele, 1990]. All timings reported are for a Sun SPARCStation2. The English lexicon used contains 38 tags ($M = 38$) and 174 ambiguity classes ($N = 174$).

Training was performed on 25,000 words in articles selected randomly from Grolier's Encyclopedia. Five iterations of training were performed in a total time of 115 CPU seconds. Following is a time breakdown by component:

Training: average μseconds per token

tokenizer	lexicon	1 iteration	5 iterations	total
640	400	680	3400	4600

Tagging was performed on 115,822 words in a collection of articles by the journalist Dave Barry. This required a total of of 143 CPU seconds. The time breakdown for this was as follows:

Tagging: average μseconds per token

tokenizer	lexicon	Viterbi	total
604	388	233	1235

It can be seen from these figures that training on a new corpus may be accomplished in a matter of minutes, and that tens of megabytes of text may then be tagged per hour.

5.2 Accurate and Robust

When using a lexicon and tagset built from the tagged text of the Brown corpus [Francis and Kučera, 1982], training on one half of the corpus (about 500,000 words) and tagging the other, 96% of word instances were assigned the correct tag. Eight iterations of training were used. This level of accuracy is comparable to the best achieved by other taggers [Church, 1988, Merialdo, 1991].

The Brown Corpus contains fragments and ungrammaticalities, thus providing a good demonstration of robustness.

5.3 Tunable and Reusable

A tagger should be tunable, so that systematic tagging errors and anomalies can be addressed. Similarly, it is important that it be fast and easy to target the tagger to new genres and languages, and to experiment with different tagsets reflecting different insights into the linguistic phenomena found in text. In section 3.5, we describe how the HMM implementation itself supports tuning. In addition, our implementation supports a number of explicit parameters to facilitate tuning and reuse, including specification of lexicon and training corpus. There is also support for a flexible tagset. For example, if we want to collapse distinctions in the lexicon, such as those between positive, comparative, and superlative adjectives, we only have to make a small change in the mapping from lexicon to tagset. Similarly, if we wish to make finer grain distinctions than those available in the lexicon, such as case marking on pronouns, there is a simple way to note such exceptions.

6 Applications

We have used the tagger in a number of applications. We describe three applications here: phrase recognition; word sense disambiguation; and grammatical function assignment. These projects are part of a research effort to use shallow analysis techniques to extract content from unrestricted text.

6.1 Phrase Recognition

We have constructed a system that recognizes simple phrases when given as input the sequence of tags for a sentence. There are recognizers for noun phrases, verb groups, adverbial phrases, and prepositional phrases. Each of these phrases comprises a contiguous sequence of tags that satisfies a simple grammar. For example, a noun phrase can be a unary sequence containing a pronoun tag or an arbitrarily long sequence of noun and adjective tags, possibly preceded by a determiner tag and possibly with an embedded possessive marker. The longest possible sequence is found (e.g., "the program committee" but not "the program").

Conjunctions are not recognized as part of any phrase; for example, in the fragment "the cats and dogs," "the cats" and "dogs" will be recognized as two noun phrases. Prepositional phrase attachment is not performed at this stage of processing. This approach to phrase recognition in some cases captures only parts of some phrases; however, our approach minimizes false positives, so that we can rely on the recognizers' results.

6.2 Word Sense Disambiguation

Part-of-speech tagging in and of itself is a useful tool in lexical disambiguation; for example, knowing that "dig" is being used as a noun rather than as a verb indicates the word's appropriate meaning. But many words have multiple meanings even while occupying the same part of speech. To this end, the tagger has been used in the implementation of an experimental noun homograph disambiguation algorithm [Hearst, 1991]. The algorithm (known as CatchWord) performs supervised training over a large text corpus, gathering lexical, orthographic, and simple syntactic evidence for each sense of the ambiguous noun. After a period of training, CatchWord classifies new instances of the noun by checking its context against that of previously observed instances and choosing the sense for which the most evidence is found. Because the sense distinctions made are coarse, the disambiguation can be accomplished without the expense of knowledge bases or inference mechanisms. Initial tests resulted in accuracies of around 90% for nouns with strongly distinct senses.

This algorithm uses the tagger in two ways: (i) to determine the part of speech of the target word (filtering out the non-noun usages) and (ii) as a step in the phrase recognition analysis of the context surrounding the noun.

6.3 Grammatical Function Assignment

The phrase recognizers also provide input to a system, Sopa [Sibun, 1991], which recognizes nominal arguments of verbs, specifically, Subject, Object, and Predicative Arguments. Sopa does not rely on information (such as arity or voice) specific to the particular verbs involved. The first step in assigning grammatical functions is to partition the tag sequence of each sentence into phrases. The phrase types include those mentioned in section 6.1, additional types to account for conjunctions, complementizers, and indicators of sentence boundaries, and an "unknown" type. After a sentence has been partitioned, each simple noun phrase is examined in the context of the phrase to its left and the phrase to its right. On the basis of this local context and a set of rules, the noun phrase is marked as a syntactic Subject, Object, Predicative, or is not marked at all. A label of Predicative is assigned only if it can be determined that the governing verb group is a form of a predicating verb (e.g., a form of "be"). Because this cannot always be determined, some Predicatives are labeled Objects. If a noun phrase is labeled, it is also annotated as to whether the governing verb is the closest verb group to the right or to the left. The algorithm has an accuracy of approximately 80% in assigning grammatical functions.

Acknowledgments

We would like to thank Marti Hearst for her contributions to this paper, Lauri Karttunen and Annie Zaenen for their work on lexicons, and Kris Halvorsen for supporting this project.

References

[Aho *et al.*, 1986] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Baum, 1972] L. E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process. *Inequalities*, 3:1–8, 1972.

[Church, 1988] K. W. Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the Second Conference on Applied Natural Language Processing (ACL)*, pages 136–143, 1988.

[Cutting *et al.*, 1991] D.R. Cutting, J. Pedersen, and P.-K. Halvorsen. An object-oriented architecture for text retrieval. In *Conference Proceedings of RIAO'91, Intelligent Text and Image Handling, Barcelona, Spain*, pages 285–298, April 1991. Also available as Xerox PARC technical report SSL-90-83.

[DeRose, 1988] S. DeRose. Grammatical category disambiguation by statistical optimization. *Computational Linguistics*, 14:31–39, 1988.

[Derouault and Merialdo, 1986] A. M. Derouault and B. Merialdo. Natural language modeling for phoneme-to-text transcription. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8:742–749, 1986.

[Francis and Kučera, 1982] W. N. Francis and F. Kučera. *Frequency Analysis of English Usage*. Houghton Mifflin, 1982.

[Garside *et al.*, 1987] R. Garside, G. Leech, and G. Sampson. *The Computational Analysis of English*. Longman, 1987.

[Greene and Rubin, 1971] B. B. Greene and G. M. Rubin. Automatic grammatical tagging of English. Technical report, Department of Linguistics, Brown University, Providence, Rhode Island, 1971.

[Hearst, 1991] M. A. Hearst. Noun homograph disambiguation using local context in large text corpora. In *The Proceedings of the 7th New OED Conference on Using Corpora*, pages 1–22, Oxford, 1991.

[Jelinek and Mercer, 1980] F. Jelinek and R. L. Mercer. Interpolated estimation of markov source parameters from sparse data. In *Proceedings of the Workshop Pattern Recognition in Practice*, pages 381–397, Amsterdam, 1980. North-Holland.

[Jelinek, 1985] F. Jelinek. Markov source modeling of text generation. In J. K. Skwirzinski, editor, *Impact of Processing Techniques on Communication*. Nijhoff, Dordrecht, 1985.

[Knuth, 1973] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[Koskenniemi, 1990] K. Koskenniemi. Finte-state parsing and disambiguation. In H. Karlgren, editor, *COLING-90*, pages 229–232, Helsinki University, 1990.

[Kupiec, 1989a] J. M. Kupiec. Augmenting a hidden Markov model for phrase-dependent word tagging. In *Proceedings of the DARPA Speech and Natural Language Workshop*, pages 92–98, Cape Cod, MA, 1989. Morgan Kaufmann.

[Kupiec, 1989b] J. M. Kupiec. Probabilistic models of short and long distance word dependencies in running text. In *Proceedings of the 1989 DARPA Speech and Natural Language Workshop*, pages 290–295, Philadelphia, 1989. Morgan Kaufmann.

[Kupiec, 1992] J. M. Kupiec. Robust part-of-speech tagging using a hidden markov model. submitted to Computer Speech and Language, 1992.

[Lesk, 1975] M. E. Lesk. LEX — a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[Levinson *et al.*, 1983] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi. An introduction to the application of the theory of probabilistic functions of a Markov process to automatic speech recognition. *Bell System Technical Journal*, 62:1035–1074, 1983.

[Merialdo, 1991] B. Merialdo. Tagging text with a probablistic model. In *Proceedings of ICASSP-91*, pages 809–812, Toronto, Canada, 1991.

[Meteer *et al.*, 1991] M. W. Meteer, R. Schwartz, and R. Weischedel. POST: Using probabilities in language processing. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 960–965, 1991.

[P754, 1981] IEEE Task P754. A proposed standard for binary floating-point arithmetic. *Computer*, 14(3):51–62, March 1981.

[Rabiner and Juang, 1986] L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, January 1986.

[Sibun, 1991] P. Sibun. Grammatical function assignment in unrestricted text. internal report, Xerox Palo Alto Research Center, 1991.

[Steele, 1990] G. L. Steele, Jr. *Common Lisp, The Language.* Digital Press, second edition, 1990.

[Viterbi, 1967] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. In *IEEE Transactions on Information Theory*, pages 260–269, April 1967.