# Ranking Algorithms for Named–Entity Extraction: Boosting and the Voted Perceptron

**Michael Collins**
AT&T Labs-Research, Florham Park, New Jersey.
mcollins@research.att.com

## Abstract

This paper describes algorithms which rerank the top N hypotheses from a maximum-entropy tagger, the application being the recovery of named-entity boundaries in a corpus of web data. The first approach uses a boosting algorithm for ranking problems. The second approach uses the voted perceptron algorithm. Both algorithms give comparable, significant improvements over the maximum-entropy baseline. The voted perceptron algorithm can be considerably more efficient to train, at some cost in computation on test examples.

## 1 Introduction

Recent work in statistical approaches to parsing and tagging has begun to consider methods which incorporate *global features* of candidate structures. Examples of such techniques are Markov Random Fields (Abney 1997; Della Pietra et al. 1997; Johnson et al. 1999), and boosting algorithms (Freund et al. 1998; Collins 2000; Walker et al. 2001). One appeal of these methods is their flexibility in incorporating features into a model: essentially any features which might be useful in discriminating good from bad structures can be included. A second appeal of these methods is that their training criterion is often *discriminative*, attempting to explicitly push the score or probability of the correct structure for each training sentence above the score of competing structures. This discriminative property is shared by the methods of (Johnson et al. 1999; Collins 2000), and also the Conditional Random Field methods of (Lafferty et al. 2001).

In a previous paper (Collins 2000), a boosting algorithm was used to rerank the output from an existing statistical parser, giving significant improvements in parsing accuracy on Wall Street Journal data. Similar boosting algorithms have been applied to natural language generation, with good results, in (Walker et al. 2001). In this paper we apply reranking methods to named-entity extraction. A state-of-the-art (maximum-entropy) tagger is used to generate 20 possible segmentations for each input sentence, along with their probabilities. We describe a number of additional global features of these candidate segmentations. These additional features are used as evidence in reranking the hypotheses from the max-ent tagger. We describe two learning algorithms: the boosting method of (Collins 2000), and a variant of the voted perceptron algorithm, which was initially described in (Freund & Schapire 1999). We applied the methods to a corpus of over one million words of tagged web data. The methods give significant improvements over the maximum-entropy tagger (a 17.7% relative reduction in error-rate for the voted perceptron, and a 15.6% relative improvement for the boosting method).

One contribution of this paper is to show that existing reranking methods are useful for a new domain, named-entity tagging, and to suggest global features which give improvements on this task. We should stress that another contribution is to show that a new algorithm, the voted perceptron, gives very credible results on a natural language task. It is an extremely simple algorithm to implement, and is very fast to train (the testing phase is slower, but by no means sluggish). It should be a viable alternative to methods such as the boosting or Markov Random Field algorithms described in previous work.

## 2 Background

### 2.1 The data

Over a period of a year or so we have had over one million words of named-entity data annotated. The data is drawn from web pages, the aim being to support a question-answering system over web data. A number of categories are annotated: the usual people, organization and location categories, as well as less frequent categories such as brand-names, scientific terms, event titles (such as concerts) and so on. From this data we created a training set of 53,609 sentences (1,047,491 words), and a test set of 14,717 sentences (291,898 words).

The task we consider is to recover named-entity boundaries. We leave the recovery of the categories of entities to a separate stage of processing.[1] We evaluate different methods on the task through precision and recall. If a method proposes $p$ entities on the test set, and $c$ of these are correct (i.e., an entity is marked by the annotator with exactly the same span as that proposed) then the precision of a method is $100\% * c/p$. Similarly, if $g$ is the total number of entities in the human annotated version of the test set, then the recall is $100\% * c/g$.

### 2.2 The baseline tagger

The problem can be framed as a tagging task – to tag each word as being either the start of an entity, a continuation of an entity, or not to be part of an entity at all (we will use the tags `S`, `C` and `N` respectively for these three cases). As a baseline model we used a maximum entropy tagger, very similar to the ones described in (Ratnaparkhi 1996; Borthwick et. al 1998; McCallum et al. 2000). Max-ent taggers have been shown to be highly competitive on a number of tagging tasks, such as part-of-speech tagging (Ratnaparkhi 1996), named-entity recognition (Borthwick et. al 1998), and information extraction tasks (McCallum et al. 2000). Thus the maximum-entropy tagger we used represents a serious baseline for the task. We used the following features (several of the features were inspired by the approach of (Bikel et. al 1999), an HMM model which gives excellent results on named entity extraction):

- The word being tagged, the previous word, and the next word.
- The previous tag, and the previous two tags (bigram and trigram features).
- A compound feature of three fields: (a) Is the word at the start of a sentence?; (b) does the word occur in a list of words which occur more frequently as lower case rather than upper case words in a large corpus of text? (c) the type of the first letter $x$ of the word, where $type(x)$ is defined as 'A' if $x$ is a capitalized letter, 'a' if $x$ is a lower-case letter, '0' if $x$ is a digit, and $x$ otherwise. For example, if the word *Animal* is seen at the start of a sentence, and it occurs in the list of frequent lower-cased words, then it would be mapped to the feature `1-1-A`.
- The word with each character mapped to its $type$. For example, *G.M.* would be mapped to `A.A.`, and *Animal* would be mapped to `Aaaaaa`.
- The word with each character mapped to its type, but repeated consecutive character types are not repeated in the mapped string. For example, *Animal* would be mapped to `Aa`, *G.M.* would again be mapped to `A.A.`.

The tagger was applied and trained in the same way as described in (Ratnaparkhi 1996). The feature templates described above are used to create a set of $m$ binary features $f_i(t, h)$, where $t$ is the tag, and $h$ is the "history", or context. An example is

$$f_{100}(t,h) = \begin{cases} 1 & \text{if t = S and the} \\ & \text{word being tagged = "Mr."} \\ 0 & \text{otherwise} \end{cases}$$

The parameters of the model are $\alpha_i$ for $i = 1 \ldots m$, defining a conditional distribution over the tags given a history $h$ as

$$P(t|h) = \frac{e^{\sum_i \alpha_i f_i(t,h)}}{\sum_{t'} e^{\sum_i \alpha_i f_i(t',h)}}$$

The parameters are trained using Generalized Iterative Scaling. Following (Ratnaparkhi 1996), we only include features which occur 5 times or more in training data. In decoding, we use a beam search to recover 20 candidate tag sequences for each sentence (the sentence is decoded from left to right, with the top 20 most probable hypotheses being stored at each point).

### 2.3 Applying the baseline tagger

As a baseline we trained a model on the full 53,609 sentences of training data, and decoded the 14,717 sentences of test data. This gave 20 candidates per

[1] In initial experiments, we found that forcing the tagger to recover categories as well as the segmentation, by exploding the number of tags, reduced performance on the segmentation task, presumably due to sparse data problems.

test sentence, along with their probabilities. The baseline method is to take the most probable candidate for each test data sentence, and then to calculate precision and recall figures. Our aim is to come up with strategies for reranking the test data candidates, in such a way that precision and recall is improved.

In developing a reranking strategy, the 53,609 sentences of training data were split into a 41,992 sentence training portion, and a 11,617 sentence development set. The training portion was split into 5 sections, and in each case the maximum-entropy tagger was trained on 4/5 of the data, then used to decode the remaining 1/5. The top 20 hypotheses under a beam search, together with their log probabilities, were recovered for each training sentence. In a similar way, a model trained on the 41,992 sentence set was used to produce 20 hypotheses for each sentence in the development set.

## 3 Global features

### 3.1 The global-feature generator

The module we describe in this section generates global features for each candidate tagged sequence. As input it takes a sentence, along with a proposed segmentation (i.e., an assignment of a tag for each word in the sentence). As output, it produces a set of feature strings. We will use the following tagged sentence as a running example in this section:

**Whether/**N **you/**N **'/**N **re/**N **an/**N **aging/**N **flower/**N **child/**N **or/**N **a/**N **clueless/**N **Gen/**S **Xer/**C **,/**N **"/**N **The/**S **Day/**C **They/**C **Shot/**C **John/**C **Lennon/**C **,/**N **"/**N **playing/**N **at/**N **the/**N **Dougherty/**S **Arts/**C **Center/**C **,/**N **entertains/**N **the/**N **imagination/**N **./**N

An example feature type is simply to list the full strings of entities that appear in the tagged input. In this example, this would give the three features

```
WE=Gen_Xer
WE=The_Day_They_Shot_John_Lennon
WE=Dougherty_Arts_Center
```

Here `WE` stands for "whole entity". Throughout this section, we will write the features in this format. The start of the feature string indicates the feature type (in this case `WE`), followed by =. Following the type, there are generally 1 or more words or other symbols, which we will separate with the symbol _.

A seperate module in our implementation takes the strings produced by the global-feature generator, and hashes them to integers. For example, suppose the three strings `WE=Gen_Xer`, `WE=The_Day_They_Shot_John_Lennon`, `WE=Dougherty_Arts_Center` were hashed to 100, 250, and 500 respectively. Conceptually, the candidate $x$ is represented by a large number of features $h_s(x)$ for $s = 1 \ldots m$ where $m$ is the number of distinct feature strings in training data. In this example, only $h_{100}(x)$, $h_{250}(x)$ and $h_{500}(x)$ take the value 1, all other features being zero.

### 3.2 Feature templates

We now introduce some notation with which to describe the full set of global features. First, we assume the following primitives of an input candidate:

- $t_i$ for $i = 1 \ldots n$ is the $i$'th tag in the tagged sequence.

- $w_i$ for $i = 1 \ldots n$ is the $i$'th word.

- $l_i$ for $i = 1 \ldots n$ is 1 if $w_i$ begins with a lowercase letter, 0 otherwise.

- $f_i$ for $i = 1 \ldots n$ is a transformation of $w_i$, where the transformation is applied in the same way as the final feature type in the maximum entropy tagger. Each character in the word is mapped to its $type$, but repeated consecutive character types are not repeated in the mapped string. For example, *Animal* would be mapped to `Aa` in this feature, *G.M.* would again be mapped to `A.A.`.

- $g_i$ for $i = 1 \ldots n$ is the same as $f_i$, but has an additional flag appended. The flag indicates whether or not the word appears in a dictionary of words which appeared more often lower-cased than capitalized in a large corpus of text. In our example, *Animal* appears in the lexicon, but *G.M.* does not, so the two values for $g_i$ would be `Aa1` and `A.A.0` respectively.

In addition, $t_i, w_i, f_i$ and $g_i$ are all defined to be `NULL` if $i < 1$ or $i > n$.

Most of the features we describe are anchored on entity boundaries in the candidate segmentation. We will use "feature templates" to describe the features that we used. As an example, suppose that an entity

| Description | Feature Template |
|---|---|
| The whole entity string | WE=$w_s\_w_{(s+1)}\_\ldots\_w_e$ |
| The $f_i$ features within the entity | FF=$f_s\_f_{(s+1)}\_\ldots\_f_e$ |
| The $g_i$ features within the entity | GF=$g_s\_g_{(s+1)}\_\ldots\_g_e$ |
| The last word in the entity | LW=$w_e$ |
| Indicates whether the last word is lower-cased | LWLC=$l_e$ |
| Bigram boundary features of the words before/after the start of the entity | BO00=$w_{(s-1)}\_w_s$ BO01=$w_{(s-1)}\_g_s$ BO10=$g_{(s-1)}\_w_s$<br>BO11=$g_{(s-1)}\_g_s$ |
| Bigram boundary features of the words before/after the end of the entity | BE00=$w_e\_w_{(e+1)}$ BE01=$w_e\_g_{(e+1)}$ BE10=$g_e\_w_{(e+1)}$<br>BE11=$g_e\_g_{(e+1)}$ |
| Trigram boundary features of the words before/after the start of the entity (16 features total, only 4 shown) | TO000=$w_{(s-2)}\_w_{(s-1)}\_w_s$ ... TO111=$g_{(s-2)}\_g_{(s-1)}\_g_s$<br>TO2000=$w_{(s-1)}\_w_s\_w_{(s+1)}$ ... TO2111=$g_{(s-1)}\_g_s\_g_{(s+1)}$ |
| Trigram boundary features of the words before/after the end of the entity (16 features total, only 4 shown) | TE000=$w_{(e-1)}\_w_e\_w_{(e+1)}$ ... TE111=$g_{(e-1)}\_g_e\_g_{(e+1)}$<br>TE2000=$w_{(e-2)}\_w_{(e-1)}\_w_e$ ... TE2111=$g_{(e-2)}\_g_{(e-1)}\_g_e$ |
| Prefix features | PF=$f_s$ PF2=$g_s$ PF=$f_s\_f_{(s+1)}$ PF2=$g_s\_g_{(s+1)}$<br>... PF=$f_s\_f_{(s+1)}\_\ldots\_f_e$ PF2=$g_s\_g_{(s+1)}\_\ldots\_g_e$ |
| Suffix features | SF=$f_e$ SF2=$g_e$ SF=$f_e\_f_{(e-1)}$ SF2=$g_e\_g_{(e-1)}$<br>... SF=$f_e\_f_{(e-1)}\_\ldots\_f_s$ SF2=$g_e\_g_{(e-1)}\_\ldots\_g_s$ |

Figure 1: The full set of entity-anchored feature templates. One of these features is generated for each entity seen in a candidate. We take the entity to span words $s \ldots e$ inclusive in the candidate.

is seen from words $s$ to $e$ inclusive in a segmentation. Then the `WE` feature described in the previous section can be generated by the template

`WE`=$w_s\_w_{s+1}\_\ldots\_w_e$

Applying this template to the three entities in the running example generates the three feature strings described in the previous section. As another example, consider the template `FF`=$f_s\_f_{s+1}\_\ldots\_f_e$. This will generate a feature string for each of the entities in a candidate, this time using the values $f_s \ldots f_e$ rather than $w_s \ldots w_e$. For the full set of feature templates that are anchored around entities, see figure 1.

A second set of feature templates is anchored around quotation marks. In our corpus, entities (typically with long names) are often seen surrounded by quotes. For example, "The Day They Shot John Lennon", the name of a band, appears in the running example. Define $s$ to be the index of any double quotation marks in the candidate, $e$ to be the index of the next (matching) double quotation marks if they appear in the candidate. Additionally, define $e2$ to be the index of the last word beginning with a lower case letter, upper case letter, or digit within the quotation marks. The first set of feature templates tracks the values of $g_i$ for the words within quotes:[2]

Q=$g_s\_t_s\_g_{(s+1)}\_t_{(s+1)}\_\ldots\_g_e\_t_e$
Q2=$g_{(s-1)}\_t_{(s-1)}\_g_s\_t_s\_\ldots\_g_{(e+1)}\_t_{(e+1)}$

[2]We only included these features if $(e - s) < 7$, to prevent an explosion in the length of feature strings.

The next set of feature templates are sensitive to whether the entire sequence between quotes is tagged as a named entity. Define $F2$ to be 1 if $t_{s+1}$ =`S`, and $t_i$=`C` for $i = s+2 \ldots e2$ (i.e., $F2 = 1$ if the sequence of words within the quotes is tagged as a single entity). Also define $U$ to be the number of upper cased words within the quotes, $L$ to be the number of lower case words, and $F$ to be 1 if $U \geq L$, 0 otherwise. Then two other templates are:

QF=$F2\_U\_L\_g_{(s+1)}\_g_{e2}$
QF2=$F2\_F\_g_{(s+1)}\_g_{e2}$

In the "The Day They Shot John Lennon" example we would have $F2 = 1$ provided that the entire sequence within quotes was tagged as an entity. Additionally, $U = 7$, $L = 0$, and $F = 1$. The values for $g_{(s+1)}$ and $g_{e2}$ would be $Aa1$ and $Aa0$ (these features are derived from *The* and *Lennon*, which respectively do and don't appear in the capitalization lexicon). This would give `QF`=1_7_0_$Aa1$_$Aa0$ and `QF2`=1_1_$Aa1$_$Aa0$.

At this point, we have fully described the representation used as input to the reranking algorithms. The maximum-entropy tagger gives 20 proposed segmentations for each input sentence. Each candidate $x$ is represented by the log probability $L(x)$ from the tagger, as well as the values of the global features $h_s(x)$ for $s = 1 \ldots m$. In the next section we describe algorithms which blend these two sources of information, the aim being to improve upon a strategy which just takes the candidate from the tagger with the highest score for $L(x)$.

## 4 Ranking Algorithms

### 4.1 Notation

This section introduces notation for the reranking task. The framework is derived by the transformation from ranking problems to a margin-based classification problem in (Freund et al. 1998). It is also related to the Markov Random Field methods for parsing suggested in (Johnson et al. 1999), and the boosting methods for parsing in (Collins 2000). We consider the following set-up:

- Training data is a set of example input/output pairs. In tagging we would have training examples $\{s_i, t_i\}$ where each $s_i$ is a sentence and each $t_i$ is the correct sequence of tags for that sentence.
- We assume some way of enumerating a set of candidates for a particular sentence. We use $x_{ij}$ to denote the $j$'th candidate for the $i$'th sentence in training data, and $\mathcal{C}(s_i) = \{x_{i1}, x_{i2} \ldots\}$ to denote the set of candidates for $s_i$. In this paper, the top $N$ outputs from a maximum entropy tagger are used as the set of candidates.
- Without loss of generality we take $x_{i1}$ to be the candidate for $s_i$ which has the most correct tags, i.e., is closest to being correct.[3]
- $Q(x_{i,j})$ is the probability that the base model assigns to $x_{i,j}$. We define $L(x_{i,j}) = \log Q(x_{i,j})$.
- We assume a set of $m$ additional features, $h_s(x)$ for $s = 1 \ldots m$. The features could be arbitrary functions of the candidates; our hope is to include features which help in discriminating good candidates from bad ones.
- Finally, the parameters of the model are a vector of $m + 1$ parameters, $\mathbf{w} = \{w_0, w_1 \ldots w_m\}$. The ranking function is defined as

$$F(x, \mathbf{w}) = w_0 L(x) + \sum_{s=1}^{m} w_s h_s(x)$$

This function assigns a real-valued number to a candidate $x$. It will be taken to be a measure of the plausibility of a candidate, higher scores meaning higher plausibility. As such, it assigns a ranking to different candidate structures for the same sentence, and in particular the output on a training or test example $s$ is $\arg\max_{x \in \mathcal{C}(s)} F(x, \mathbf{w})$. In this paper we take the features $h_s$ to be fixed, the learning problem being to choose a good setting for the parameters $\mathbf{w}$.

In some parts of this paper we will use vector notation. Define $\mathbf{h}(x)$ to be the vector $\{L(x), h_1(x) \ldots h_m(x)\}$. Then the ranking score can also be written as $F(x, \mathbf{w}) = \mathbf{w} \cdot \mathbf{h}(x)$ where $\mathbf{x} \cdot \mathbf{y}$ is the dot product between vectors $\mathbf{x}$ and $\mathbf{y}$.

### 4.2 The boosting algorithm

The first algorithm we consider is the boosting algorithm for ranking described in (Collins 2000). The algorithm is a modification of the method in (Freund et al. 1998). The method can be considered to be a greedy algorithm for finding the parameters $\mathbf{w}$ that minimize the loss function

$$Loss(\mathbf{w}) = \sum_i \sum_{j \geq 2} e^{F(x_{i,j}, \mathbf{w}) - F(x_{i,1}, \mathbf{w})}$$

where as before, $F(x, \mathbf{w}) = \mathbf{w} \cdot \mathbf{h}(x)$. The theoretical motivation for this algorithm goes back to the PAC model of learning. Intuitively, it is useful to note that this loss function is an upper bound on the number of "ranking errors", a ranking error being a case where an incorrect candidate gets a higher value for $F$ than a correct candidate. This follows because for all $x$, $e^{-x} \geq I[x]$, where we define $I[x]$ to be 1 for $x \leq 0$, and 0 otherwise. Hence

$$Loss(\mathbf{w}) \geq \sum_i \sum_{j \geq 2} I[M_{i,j}]$$

where $M_{i,j} = F(x_{i,1}, \mathbf{w}) - F(x_{i,j}, \mathbf{w})$. Note that the number of ranking errors is $\sum_i \sum_{j \geq 2} I[M_{i,j}]$.

As an initial step, $w_0$ is set to be

$$w_0 = \arg\min_w \sum_i \sum_{j \geq 2} e^{w(L(x_{i,j}) - L(x_{i,1}))}$$

and all other parameters $w_s$ for $s = 1 \ldots m$ are set to be zero. The algorithm then proceeds for $N$ iterations ($N$ is usually chosen by cross validation on a development set). At each iteration, a single feature is chosen, and its weight is updated. Suppose the current parameter values are $\mathbf{w}$, and a single feature $k$ is chosen, its weight being updated through an increment $\delta$, i.e., $w_k = w_k + \delta$. Then the new loss, after this parameter update, will be

$$L(k, \delta) = \sum_{i, j \geq 2} e^{-M_{i,j} + \delta(h_k(x_{i,j}) - h_k(x_{i,1}))}$$

[3] In the event that multiple candidates get the same, highest score, the candidate with the highest value of log-likelihood $L$ under the baseline model is taken as $x_{i,1}$.

where $M_{i,j} = F(x_{i,1}, \mathbf{w}) - F(x_{i,j}, \mathbf{w})$. The boosting algorithm chooses the feature/update pair $k^*, \delta^*$ which is optimal in terms of minimizing the loss function, i.e.,

$$(k^*, \delta^*) = \arg\min_{k,\delta} L(k, \delta) \tag{1}$$

and then makes the update $w_{k^*} = w_{k^*} + \delta^*$.

Figure 2 shows an algorithm which implements this greedy procedure. See (Collins 2000) for a full description of the method, including justification that the algorithm does in fact implement the update in Eq. 1 at each iteration.[4] The algorithm relies on the following arrays:

$$\begin{aligned}
A_k^+ &= \{(i,j) : [h_k(x_{i,1}) - h_k(x_{i,j})] = 1\} \\
A_k^- &= \{(i,j) : [h_k(x_{i,1}) - h_k(x_{i,j})] = -1\} \\
B_{i,j}^+ &= \{k : [h_k(x_{i,1}) - h_k(x_{i,j})] = 1\} \\
B_{i,j}^- &= \{k : [h_k(x_{i,1}) - h_k(x_{i,j})] = -1\}
\end{aligned}$$

Thus $A_k^+$ is an index from features to correct/incorrect candidate pairs where the $k$'th feature takes value 1 on the correct candidate, and value 0 on the incorrect candidate. The array $A_k^-$ is a similar index from features to examples. The arrays $B_{i,j}^+$ and $B_{i,j}^-$ are reverse indices from training examples to features.

### 4.3 The voted perceptron

Figure 3 shows the training phase of the perceptron algorithm, originally introduced in (Rosenblatt 1958). The algorithm maintains a parameter vector $\mathbf{w}$, which is initially set to be all zeros. The algorithm then makes a pass over the training set, at each training example storing a parameter vector $\mathbf{w}^i$ for $i = 1 \ldots n$. The parameter vector is only modified when a mistake is made on an example. In this case the update is very simple, involving adding the difference of the offending examples' representations ($\mathbf{w}^i = \mathbf{w}^{i-1} + \mathbf{h}(x_{i1}) - \mathbf{h}(x_{ij})$ in the figure). See (Cristianini and Shawe-Taylor 2000) chapter 2 for discussion of the perceptron algorithm, and theory justifying this method for setting the parameters.

In the most basic form of the perceptron, the parameter values $\mathbf{w}^n$ are taken as the final parameter settings, and the output on a new test example with $x_j$ for $j = 1 \ldots m$ is simply the highest

[4] Strictly speaking, this is only the case if the smoothing parameter $\epsilon$ is 0.

**Input**

- Examples $x_{i,j}$ with initial scores $L(x_{i,j})$
- Arrays $A_k^+$, $A_k^-$, $B_{i,j}^+$ and $B_{i,j}^-$ as described in section 4.2.
- Parameters are number of rounds of boosting $N$, a smoothing parameter $\epsilon$.

**Initialize**

- Set $w_0 = \arg\min_w \sum_{i,j} e^{w(L(x_{i,j}) - L(x_{i,1}))}$
- Set $\mathbf{w} = \{w_0, 0, 0, \ldots\}$
- For all $i, j$, set $M_{i,j} = w_0 \left[L(x_{i,1}) - L(x_{i,j})\right]$.
- Set $Z = \sum_i \sum_{j \geq 2} e^{-M_{i,j}}$
- For $k = 1 \ldots m$, calculate
  - $W_k^+ = \sum_{(i,j) \in A_k^+} e^{-M_{i,j}}$
  - $W_k^- = \sum_{(i,j) \in A_k^-} e^{-M_{i,j}}$
  - $BestLoss(k) = \left|\sqrt{W_k^+} - \sqrt{W_k^-}\right|$

**Repeat** for $t$ = 1 to $N$

- Choose $k^* = \arg\max_k BestLoss(k)$
- Set $\delta^* = \frac{1}{2} \log \frac{W_{k^*}^+ + \epsilon Z}{W_{k^*}^- + \epsilon Z}$
- Update one parameter, $w_{k^*} = w_{k^*} + \delta^*$
- for $(i,j) \in A_{k^*}^+$
  - $\Delta = e^{-M_{i,j} - \delta^*} - e^{-M_{i,j}}$
  - $M_{i,j} = M_{i,j} + \delta^*$
  - for $k \in B_{i,j}^+$, $\quad W_k^+ = W_k^+ + \Delta$
  - for $k \in B_{i,j}^-$, $\quad W_k^- = W_k^- + \Delta$
  - $Z = Z + \Delta$
- for $(i,j) \in A_{k^*}^-$
  - $\Delta = e^{-M_{i,j} + \delta^*} - e^{-M_{i,j}}$
  - $M_{i,j} = M_{i,j} - \delta^*$
  - for $k \in B_{i,j}^+$, $\quad W_k^+ = W_k^+ + \Delta$
  - for $k \in B_{i,j}^-$, $\quad W_k^- = W_k^- + \Delta$
  - $Z = Z + \Delta$
- For all features $k$ whose values of $W_k^+$ and/or $W_k^-$ have changed, recalculate $BestLoss(k) = \left|\sqrt{W_k^+} - \sqrt{W_k^-}\right|$

**Output** Final parameter setting $\mathbf{w}$

Figure 2: The boosting algorithm.

**Define:** $F(x, \mathbf{w}) = \mathbf{w} \cdot \mathbf{h}(x)$.
**Input:** Examples $x_{i,j}$ with feature vectors $\mathbf{h}(x_{i,j})$.
**Initialization:** Set parameters $\mathbf{w}^0 = 0$
**For** $i = 1 \ldots n$
$j = \text{argmax}_{j=1\ldots n_i} F(x_{ij}, \mathbf{w}^{i-1})$
**If** $(j = 1)$ **Then** $\mathbf{w}^i = \mathbf{w}^{i-1}$
**Else** $\mathbf{w}^i = \mathbf{w}^{i-1} + \mathbf{h}(x_{i1}) - \mathbf{h}(x_{ij})$
**Output:** Parameter vectors $\mathbf{w}^i$ for $i = 1 \ldots n$

Figure 3: The perceptron training algorithm for ranking problems.

**Define:** $F(x, \mathbf{w}) = \mathbf{w} \cdot \mathbf{h}(x)$.
**Input:** A set of candidates $x_j$ for $j = 1 \ldots m$,
A sequence of parameter vectors $\mathbf{w}^i$ for $i = 1 \ldots n$
**Initialization:** Set $V[j] = 0$ for $j = 1 \ldots m$
($V[j]$ stores the number of votes for $x_j$)
**For** $i = 1 \ldots n$
$j = \text{argmax}_{k=1\ldots m} F(x_k, \mathbf{w}^i)$
$V[j] = V[j] + 1$
**Output:** $x_j$ where $j = \arg\max_k V[k]$

Figure 4: Applying the voted perceptron to a test example.

scoring candidate under these parameter values, i.e., $x_k$ where $k = \arg\max_j \mathbf{w}^n \cdot \mathbf{h}(x_j)$.

(Freund & Schapire 1999) describe a refinement of the perceptron, the voted perceptron. The training phase is identical to that in figure 3. Note, however, that all parameter vectors $\mathbf{w}^i$ for $i = 1 \ldots n$ are stored. Thus the training phase can be thought of as a way of constructing $n$ different parameter settings. Each of these parameter settings will have its own highest ranking candidate, $x_k$ where $k = \arg\max_j \mathbf{w}^i \cdot \mathbf{h}(x_j)$. The idea behind the voted perceptron is to take each of the $n$ parameter settings to "vote" for a candidate, and the candidate which gets the most votes is returned as the most likely candidate. See figure 4 for the algorithm.[5]

## 5 Experiments

We applied the voted perceptron and boosting algorithms to the data described in section 2.3. Only features occurring on 5 or more distinct training sentences were included in the model. This resulted in 93,777 distinct features. The two methods were trained on the training portion (41,992 sentences) of the training set. We used the development set to pick the best values for tunable parameters in each algorithm. For boosting, the main parameter to pick is the number of rounds, $N$. We ran the algorithm for a total of 300,000 rounds, and found that the optimal value for F-measure on the development set occurred after 83,233 rounds. For the voted perceptron, the representation $\mathbf{h}(x)$ was taken to be a vector $\{\beta L(x), h_1(x) \ldots h_m(x)\}$ where $\beta$ is a parameter that influences the relative contribution of the log-likelihood term versus the other features. A value of $\beta = 0.5$ was found to give the best results on the development set. Figure 5 shows the results for the three methods on the test set. Both of the reranking algorithms show significant improvements over the baseline: a 15.6% relative reduction in error for boosting, and a 17.7% relative error reduction for the voted perceptron.

[5]Note that, for reasons of explication, the decoding algorithm we present is less efficient than necessary. For example, when $\mathbf{w}^i = \mathbf{w}^{i-1}$ it is preferable to use some book-keeping to avoid recalculation of $F(x, \mathbf{w}^i)$ and $\arg\max_j F(x_j, \mathbf{w}^i)$.

| | P | R | F |
|---|---|---|---|
| Max-Ent | 84.4 | 86.3 | 85.3 |
| Boosting | 87.3(18.6) | 87.9(11.6) | 87.6(15.6) |
| Voted Perceptron | 87.3(18.6) | 88.6(16.8) | 87.9(17.7) |

Figure 5: Results for the three tagging methods. $P$ = precision, $R$ = recall, $F$ = F-measure. Figures in parantheses are relative improvements in error rate over the maximum-entropy model. All figures are percentages.

In our experiments we found the voted perceptron algorithm to be considerably more efficient in training, at some cost in computation on test examples. Another attractive property of the voted perceptron is that it can be used with kernels, for example the kernels over parse trees described in (Collins and Duffy 2001; Collins and Duffy 2002). (Collins and Duffy 2002) describe the voted perceptron applied to the named-entity data in this paper, but using kernel-based features rather than the explicit features described in this paper. See (Collins 2002) for additional work using perceptron algorithms to train tagging models, and a more thorough description of the theory underlying the perceptron algorithm applied to ranking problems.

## 6 Discussion

A question regarding the approaches in this paper is whether the features we have described could be incorporated in a maximum-entropy tagger, giving similar improvements in accuracy. This section discusses why this is unlikely to be the case. The problem described here is closely related to the label bias problem described in (Lafferty et al. 2001).

One straightforward way to incorporate global features into the maximum-entropy model would be to introduce new features $f(h,t)$ which indicated whether the tagging decision $t$ in the history $h$ creates a particular global feature. For example, we could introduce a feature

$$f_{190}(t,h) = \begin{cases} 1 & \text{if t = } \mathtt{N} \text{ and this decision} \\ & \text{creates an } \textit{LWLC=1} \text{ feature} \\ 0 & \text{otherwise} \end{cases}$$

As an example, this would take the value 1 if *its* was tagged as N in the following context,

**She/**N **praised/**N **the/**N **University/**S **for/**C **its/**? **efforts to** . . .

because tagging *its* as N in this context would create an entity whose last word was not capitalized, i.e., *University for*. Similar features could be created for all of the global features introduced in this paper.

This example also illustrates why this approach is unlikely to improve the performance of the maximum-entropy tagger. The parameter $\alpha_{190}$ associated with this new feature can only affect the score for a proposed sequence by modifying $p(t|h)$ at the point at which $f_{190}(t,h) = 1$. In the example, this means that the *LWLC=1* feature can only lower the score for the segmentation by lowering the probability of tagging *its* as N. But *its* has almost probably 1 of not appearing as part of an entity, so $p(N|h)$ should be almost 1 whether $f_{190}$ is 1 or 0 in this context! The decision which effectively created the entity *University for* was the decision to tag *for* as C, and this has already been made. The independence assumptions in maximum-entropy taggers of this form often lead points of local ambiguity (in this example the tag for the word *for*) to create globally implausible structures with unreasonably high scores. See (Collins 1999) section 8.4.2 for a discussion of this problem in the context of parsing.

**Acknowledgements** Many thanks to Jack Minisi for annotating the named-entity data used in the experiments. Thanks also to Nigel Duffy, Rob Schapire and Yoram Singer for several useful discussions.

## References

Abney, S. 1997. Stochastic Attribute-Value Grammars. *Computational Linguistics,* 23(4):597-618.

Bikel, D., Schwartz, R., and Weischedel, R. (1999). An Algorithm that Learns What's in a Name. In *Machine Learning: Special Issue on Natural Language Learning*, 34(1-3).

Borthwick, A., Sterling, J., Agichtein, E., and Grishman, R. (1998). Exploiting Diverse Knowledge Sources via Maximum Entropy in Named Entity Recognition. *Proc. of the Sixth Workshop on Very Large Corpora*.

Collins, M. (1999). Head-Driven Statistical Models for Natural Language Parsing. *PhD Thesis, University of Pennsylvania*.

Collins, M. (2000). Discriminative Reranking for Natural Language Parsing. *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000).*

Collins, M., and Duffy, N. (2001). Convolution Kernels for Natural Language. In *Proceedings of NIPS 14*.

Collins, M., and Duffy, N. (2002). New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. In *Proceedings of ACL 2002*.

Collins, M. (2002). Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with the Perceptron Algorithm. In *Proceedings of EMNLP 2002*.

Cristianini, N., and Shawe-Tayor, J. (2000). *An introduction to Support Vector Machines and other kernel-based learning methods.* Cambridge University Press.

Della Pietra, S., Della Pietra, V., and Lafferty, J. (1997). Inducing Features of Random Fields. IEEE Transactions on Pattern Analysis and Machine Intelligence, 19(4), pp. 380-393.

Freund, Y. & Schapire, R. (1999). Large Margin Classification using the Perceptron Algorithm. In *Machine Learning*, 37(3):277–296.

Freund, Y., Iyer, R.,Schapire, R.E., & Singer, Y. (1998). An efficient boosting algorithm for combining preferences. In *Machine Learning: Proceedings of the Fifteenth International Conference*.

Johnson, M., Geman, S., Canon, S., Chi, Z. and Riezler, S. (1999). Estimators for Stochastic "Unification-based" Grammars. Proceedings of the ACL 1999.

Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of ICML 2001*.

McCallum, A., Freitag, D., and Pereira, F. (2000) Maximum entropy markov models for information extraction and segmentation. In *Proceedings of ICML 2000*.

Ratnaparkhi, A. (1996). A maximum entropy part-of-speech tagger. In *Proceedings of the empirical methods in natural language processing conference*.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65, 386–408. (Reprinted in *Neurocomputing* (MIT Press, 1998).)

Walker, M., Rambow, O., and Rogati, M. (2001). SPoT: a trainable sentence planner. In *Proceedings of the 2nd Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL 2001).*