Fast High-Accuracy Part-of-Speech Tagging by Independent Classifiers

Robert C. Moore
Google Inc.
bobmoore@google.com

Abstract

Part-of-speech (POS) taggers can be quite accurate, but for practical use, accuracy often has to be sacrificed for speed. For example, the maintainers of the Stanford tagger (Toutanova et al., 2003; Manning, 2011) recommend tagging with a model whose per tag error rate is 17% higher, relatively, than their most accurate model, to gain a factor of 10 or more in speed. In this paper, we treat POS tagging as a single-token independent multiclass classification task. We show that by using a rich feature set we can obtain high tagging accuracy within this framework, and by employing some novel feature-weight-combination and hypothesis-pruning techniques we can also get very fast tagging with this model. A prototype tagger implemented in Perl is tested and found to be at least 8 times faster than any publicly available tagger reported to have comparable accuracy on the standard Penn Treebank Wall Street Journal test set.

1 Introduction

Part-of-speech (POS) tagging remains an important basic task in natural-language processing, often being used as an initial step in addressing more complex problems such as parsing (e.g., McDonald et al., 2005) or named-entity recognition (e.g., Florian et al., 2003). State-of-the-art-taggers typically employ discriminatively-trained models with hidden tag-sequence features. These models include features of the observable input sequence, plus hidden features consisting of tag sequences up to some fixed length.

With a tag-sequence model, the highest scoring tagging for an input sentence can be found by the Viterbi algorithm, but exact search can be slow with a large tag set. If tri-tag features are used, the full search space is $O(|T|^3 n)$, where $|T|$ is the size of the tag set and n is the length of the sentence. For the English Penn Treebank (Marcus et al., 1993) , $|T| = 45$, hence $|T|^3 = 91125$. For efficiency, some form of approximate search is normally used. For example, both Shen et al. (2007) and Huang et al. (2012) use approximate search in both training and tagging. Shen et al. use a specialized bi-directional beam search in which the search order is learned at training time and applied at tagging time, along with the model. Huang et al. use a more conventional left-to-right beam search, but they explore various special variants of the perceptron algorithm to cope with search errors during model training. These two taggers represent the current state of the art on the Penn Treebank Wall Street Journal (WSJ) corpus, for models trained using no additional resources, as measured on the standard training/development/test data split introduced by Collins (2002a): 2.67% per tag error for Shen et al., and 2.65% for Huang et al.

Alternatively, one may omit hidden tag-sequence features, enrich the set of observable features, and treat tagging each token as an independent multi-class classification problem. Toutanova et al. (2003) were the first to note that such models could achieve fairly high accuracy for POS tagging, reporting per-tag error of 3.43% on the standard WSJ development set. Liang et al. (2008) report 3.2% error on the standard WSJ test set (using a slightly smaller than standard training set), which as far as we know is the current state of the art for WSJ POS tagging by independent classifiers. The independent classifier approach has the advantage of a simple model structure with a search space for tagging of $O(|T|n)$. On the other hand, while Liang et al.'s result would have been state-of-the-art before Collins (2002a), today it represents an error rate about 20% higher than Huang et al.'s best result for tri-tag-based POS tagging, under similar training conditions.

In the first part of this paper, we introduce new features for tagging by independent classifiers. We introduce case-insensitive versions of several standard types of features, which enables our models to generalize over different casings of the same underlying word. We also cluster the vocabulary of the annotated training set, preserving as much information as possible about the tag probabilities for each word, and use sequences of the resulting classes to approximate the contextual information provided by hidden tri-tag features. With the further addition of another set of word-class features based on distributional similarity over a large corpus of unnanotated data, we obtain a model with a WSJ test set error of 2.66% (97.34% accuracy).

In the remainder of the paper, we show how to perform fast tagging with this model. Even with the simple structure of an independent multiclass classifer, tagging can be slow with a rich model and a large tag set, simply because feature extraction and model scoring take so much time. We address this in two ways. First we effectively reduce the number of features that have to be considered for a given token by combining the feature weights for more general features into those for more specific features. For example, if a word is in the training set vocabulary, none of its sublexical features need to be extracted or scored, if the weights of those features have already been combined into the weights for the corresponding "whole word" feature. Second, we limit the number of tags considered for each token by a pruning method that refines Ratnaparkhi's (1996) tag dictionary, employing a Kneser-Ney-smoothed probability distribution over the possible tags for each word, and applying a threshold tuned to reduce the number of tags considered while minimizing loss of accuracy. We have implemented a prototype tagger in Perl using these methods, which we find to be at least 8 times faster than any of the publicly available taggers reported to have comparable accuracy on the standard WSJ test set.

2 Models for Tagging by Independent Classifiers

We formulate the POS-tagging task as a linear multiclass classification problem defined by a set of tags $\mathcal{T}$ and a set of indicator features $\mathcal{F}$. Each training example consists of a set of features $\mathbf{f} \subseteq \mathcal{F}$ present in that example and a correct tag $t \in \mathcal{T}$. The feature set $\mathbf{f}$ for a particular example consists of observable properties of the token to be tagged and the tokens surrounding it. A model is a vector $\mathbf{w} \in \Re^{|\mathcal{T}|\times|\mathcal{F}|}$ indexed by feature-tag pairs. We refer to the coordinates $w_{(f,t)}$ of $\mathbf{w}$ as *feature weights*. A model $\mathbf{w}$ maximizes the sum of relevant feature weights to predict a tag $t(\mathbf{f}, \mathbf{w})$:

$$t(\mathbf{f}, \mathbf{w}) = \arg\max_{t \in \mathcal{T}} \sum_{f \in \mathbf{f}} w_{(f,t)} \tag{1}$$

In the remainder of this section we explain the feature sets we use and our method of training feature weights, and we evaluate the accuracy of the resulting models on the usual Wall Street Journal corpus from Penn Treebank III (Marcus et al., 1993).

2.1 Lexical Features

As noted above, the current state of the art for tagging by independent classifiers seems to be the results presented by Liang et al. (2008). Their best model uses the following set of base features for each word:

Whether the first character of the word is a capital letter
Prefixes of the word up to three characters
Suffixes of the word up to three characters
Two "shape" features described below
The full word

For each base feature, Liang et al. define three expanded features: whether the token being tagged has the base feature, whether the preceding token has the base feature, and whether the following token has the base feature. The shape features were first introduced by Collins (2002b) for named-entity recognition. What we will call the "Shape 1" feature is a generalization of the spelling of the word with all capital letters treated as equivalent, all lower-case letters treated as equivalent, and all digits treated as equivalent. All other characters are treated as distinct. In the "Shape 2" feature, all sequences of capital letters, all sequences of lower case letters, and all sequences of digits are treated as equivalent, regardless of the length of the sequence or the identity of the upper case letters, lower case letters, or digits.

With this feature set as our starting point, and partially drawing from the feature sets of Ratnaparkhi (1996) and Collins (2002a), we settled on the following set of base features through experimentation on the WSJ development set:

Whether the word contains a capital letter
Whether the word contains a digit
Whether the word contains a hyphen
Lower-cased prefixes of the word up to four characters
Lower-cased suffixes of the word up to four characters
The Shape 1 feature for the word
The Shape 2 feature for the word
The full lower-cased word
The full word
A distributional-similarity-based class for the full word

In *all* these features we ignore distinctions among digits (rather than just in the shape features, as Liang et al. do). For the last feature, we used 256 word classes derived by unsupervised clustering for the most frequent 999996 distinct tokens (ignoring distinctions among digits) in 121.6 billion tokens of English-language newswire, using the method of Uszkoreit and Brants (2008). A 257th class was added for tokens not found in this set. We use Liang et al.'s mapping of all base features into expanded features for the token being tagged, the preceding token, and the following token. For the first token of a sentence we include a beginning-of-sentence feature in place of the preceding-token features, and for the last token of a sentence we include an end-of-sentence feature in place of the following-token features.

2.2 Word-Class-Sequence Features

In a hidden tri-tag model, the prediction for a particular tag t_i is linked to the predictions for the preceding tag t_{i-1}, the following tag t_{i+1}, the preceding tag pair $\langle t_{i-2}, t_{i-1}\rangle$, the following tag pair $\langle t_{i+1}, t_{i+2}\rangle$, and the surrounding tag pair $\langle t_{i-1}, t_{i+1}\rangle$. In tagging by independent classifiers, we do not have access to information regarding predictions for these nearby tags and tag combinations.

To substitute for these missing features, we carry out supervised clustering of the distinct words in the training set (again ignoring distinctions among digits) into 50 classes, attempting to maximize the information carried by each class regarding the tag probabilities for the words in the class. From these classes, we construct the features

$c(w_{i-1})$
$c(w_{i+1})$
$\langle c(w_{i-2}), c(w_{i-1})\rangle$
$\langle c(w_{i+1}), c(w_{i+2})\rangle$
$\langle c(w_{i-1}), c(w_{i+1})\rangle$

The type of clustering we use here differs from the unsupervised clustering described previously. In assigning each word to a cluster, the unsupervised clustering algorithm looks only at adjacent words in unannoted data, while the supervised clustering algorithm looks only at the tags the word receives in the annotated data. The unsupervised clustering tells us what known words a large number of unknown words are simliar to, but the supervised clustering carries much more information about what tags the known words are likely to receive.

2.2.1 Clustering Algorithm

Our supervised clustering algorithm is based on the method presented by Dhillon et al. (2003). This is similar to the well-known Lloyd algorithm for k-means clustering, but uses KL-divergence between probability distributions, instead of Euclidian distance, to assign items to clusters. In our application of this algorithm, we simply keep moving each word to the cluster that has the most similar probability distribution over tags, and then re-estimating the tag probability distributions for the clusters, until the clustering converges. At a high-level, our algorithm is:

- For each unique word w in the training set, estimate a smoothed probability distribution $p(T|w)$ over tags given w.
- Select k seed words, and initialize k clusters for clustering 0, with one seed word per cluster.[1]
- Set $i = 0$.
- Repeat until the assignment of words to clusters in clustering i is the same as in clustering $i - 1$, returning clustering i:
  - For each cluster c in clustering i, compute a probability distribution $p(T|c)$ over tags given c, such that
  $$p(t|c) = \sum_{w \in c} p(w|c)p(t|w)$$
  - For each word w, find the cluster c that minimizes the KL-divergence $D_{KL}(p(T|w)||p(T|c))$, and assign w to cluster c in clustering $i + 1$.
  - Set $i = i + 1$.

As indicated, the probability distributions $p(T|c)$ over tags for a given cluster are computed as the word-frequency-weighted mean of probability distributions $p(T|w)$ over tags given the words in the cluster. The $p(T|w)$ distributions are estimated based on the relative frequencies of each tag for a given word, smoothed using the interpolated Kneser-Ney method (Chen and Goodman, 1999) widely used in statistical language modeling. (See Section 3.2 for more discussion of this smoothing method applied to POS tag prediction.)

2.2.2 Cluster Initialization

Our clustering algorithm is identical to that of Dhillon et al., except for the method of initializing the clusters. Their initialization method would assign all words with the same most likely tag to the same initial cluster. Instead, we initialize the clusters using a set of seed words with the property that conflating any two of them would result in a large loss of information about tag probabilities.

We define the distance between a pair of words (w_1, w_2) as the total decrease resulting from treating w_1 and w_2 as indistinguishable, in the estimated log probability, based on $p(T|W)$, of the reference tagging of the training data. Letting n_1 be the number of occurrences in the training data of w_1, and similarly for n_2 and w_2, we compute the distance between w_1 and w_2 as

$$n_1 D_{KL}(p(T|w_1)||p_{w_1 w_2}) + n_2 D_{KL}(p(T|w_2)||p_{w_1 w_2})$$

where $p_{w_1 w_2} = p(T|w_1 \vee w_2)$, computed as

$$p_{w_1 w_2}(t) = \frac{n_1}{n_1 + n_2} p(t|w_1) + \frac{n_2}{n_1 + n_2} p(t|w_2)$$

We select a set S of k seed words as follows:

- Choose a maximal subset V of the training data vocabulary, such that every word in V has a different distribution of observed POS tags.
- Choose a random ordering of V.
- Initialize S to contain the first k words of V.

[1]Note that most words in the training set are not assigned to any initial cluster.

```
• Find the minimum distance $d$ between any two words in $S$.
• Taking each remaining word $w$ of $V$ in order:
  – Find the minimum distance $d'$ between $w$ and any word in $S$.
  – If $d' > d$,
    ∗ Select from $S$ a pair of words $(w', w'')$ separated by $d$.
    ∗ Find the minimum distance $d'_2$ between $w'$ and any word in $S$ other than $w''$.
    ∗ Find the minimum distance $d''_2$ between $w''$ and any word in $S$ other than $w'$.
    ∗ If $d'_2 < d''_2$, remove $w'$ from $S$, otherwise remove $w''$ from $S$.
    ∗ Add $w$ to $S$.
    ∗ Recompute the minimum distance $d$ between any two words in $S$.
```

2.2.3 Random restarts

The clustering we find depends on the set of seed words, which in turn depends on the order in which the words in V are enumerated to select the seed words. To ensure that we find a good clustering, we try multiple runs of the algorithm based on different random enumerations of V, returning the clustering yielding the lowest entropy for predicting the training set tags from the clusters.

We noticed in preliminary experiments that a poor clustering on the first iteration of the algorithm seldom leads to a good final clustering, so we save the training set tag entropy for the first iteration of the best clustering found so far, and we abandon a run of the algorithm if it results in higher training set tag entropy on its first iteration than the best previously observed final clustering had on its first iteration. We continue trying different random enumerations until a fixed number of runs has passed since the current best clustering was found.

2.2.4 Classes for unknown words

Note that this clustering method assigns classes only to words observed in the training data. All words (ignoring distinctions among digits) not seen in the training data are assigned to an additional class. In training the tagging model, however, we treat each word that has a single occurrence in the training data as a member of this unknown-word class, so that features based on that class will be seen in training; but at tagging time, we give all words seen in the training data the class they are assigned by the clustering algorithm, and apply the unknown-word class only to words not seen in the training data.

2.3 Feature Weight Training

Our models are trained by optimizing the multiclass SVM hinge loss objective (Crammer and Singer, 2001) using stochastic subgradient descent as described by Zhang (2004). We use a small, constant learning rate of 2^{-8}, which early in our experiments we found generally to be a good value, given the size of our training set and the sorts of feature sets we were using. We did not re-optimize the learning rate as we experimented with different feature sets. We do not use a numerical regularizer (such as L_1 or L_2), but we avoid over-fitting by using early stopping, and averaging as Collins (2002a) does with the averaged perceptron. To determine the stopping point, we evaluate the model on the development set after each pass through the training data. We continue iterating until we have made 10 consecutive passes through the training data without reducing the development set error, and we return the model from the iteration with the lowest error.

2.4 Evaluation of Tagging Accuracy

We evaluate the tagging accuracy of three models: our new model with all the features discussed above, our new model minus the unsupervised distributional clustering features (to give a "no additional resources" measurement), and the Liang et al. model that was our starting point. Our data is the ususal Wall Street Journal corpus from Penn Treebank III (Marcus et al., 1993), split into standard training (sections 0–18), development (sections 19–21), and test (sections 22-24) sets.

Table 1 shows WSJ development and test set error rates for all tokens and for unknown-word (OOV) tokens for all three models. Our full model has an overall test set tag error rate of 2.66%, or 97.34%

Tagging Model	Dev Set All Tag Error %	Dev Set OOV Tag Error %	Test Set All Tag Error %	Test Set OOV Tag Error %
Our full feature set	2.69	9.40	2.66	8.93
Our features minus unsupervised classes	2.83	10.45	2.77	10.14
Liang et al. feature set	3.23	12.47	3.17	11.92

Table 1: WSJ development and test set error rates for different feature sets

accuracy. Omitting unsupervised word-class features results in a relative increase in the error rate of 4.1% overall and 13.5% on unknown words. The model trained on the Liang et al. feature set gives results consistent with their reported 3.2% test set error, but the error is 19.2% higher than the model using our full feature set, and 14.4% higher than our model without unsupervised word-class features.

3 Efficient Tag Inference

Although the complexity of tag inference with our model is only $O(|T|n)$, with a rich feature set and many possible tags, the simple summation of feature weights and comparison of sums implied by Equation 1 can still be slow. With our full model, a given token occurrence can have up to 53 features present, and on the WSJ development set, we measured the average number of features present with a non-zero weight for at least one tag to be 38.0. Given 45 possible tags in the Penn Treebank tag set and our full model, the average number of relevant non-zero feature weights per token on the WSJ development set is 1215.0. We reduce computational costs in two ways. First, we introduce a method of combining feature weights that effectively reduces the number of features per token by a factor of 8. Then we introduce a refined version of a tag dictionary that reduces the number of tags considered per token by a factor of more than 12 without noticeably affecting tagging accuracy. The combination of these techniques reduces the number of non-zero feature weights used per token by a factor of 75, which, in our Perl implementation, speeds up tagging by a factor of 45.

3.1 Combining Feature Weights

The base lexical feature types in our model form a natural hierarchy as follows:

- 1. Original case tokens
  - 1.1. Unsupervised distributional word clusters
  - 1.2. Lower-cased tokens
    - 1.2.1. Lower-cased 4-character prefixes
      - 1.2.1.1. Lower-cased 3-character prefixes
        - 1.2.1.1.1. Lower-cased 2-character prefixes
          - 1.2.1.1.1.1. Lower-cased 1-character prefixes
    - 1.2.2. Lower-cased 4-character suffixes
      - 1.2.2.1. Lower-cased 3-character suffixes
        - 1.2.2.1.1. Lower-cased 2-character suffixes
          - 1.2.2.1.1.1. Lower-cased 1-character suffixes
  - 1.3. Shape 1 features
    - 1.3.1. Shape 2 features
      - 1.3.1.1. Contains upper case token
      - 1.3.1.2. Contains digit
      - 1.3.1.3. Contains hyphen

The significance of the hierarchy is that the occurrence of a base feature of any of these types fully determines which features of the types below it in the hierarchy also occur. For example, given a whole token with its original casing, the corresponding features of all the other feature types in the hierarchy are completely determined. Given just the lower-cased version of the token, the lower-cased prefixes and suffixes are determined, but the distributional word cluster and the shape features are not completely determined, because they depend on capitalization.[2]

We use this hierarchy to perform a simple transformation on the trained tagging model. For every base lexical feature f found in the training data, we add to the value of each feature weight associated with that base feature, the value of all corresponding feature weights for base features below f in the hierarchy. For instance, to the feature weight for the 3-character suffix *ion*, the tag *NN*, and the position -1 (i.e, the word preceding the word being tagged), we add the value of feature weight for the 2-character suffix *on*, the tag *NN*, and the position -1, plus the value of the feature weight for the 1-character suffix *n*, the tag *NN*, and the position -1.

To use this transformed model, we make a corresponding modification to feature extraction in the tagger. We carry out feature extraction top-down with respect to the base feature hierarchy, and whenever we find a base feature f for which there are any corresponding feature weights in the model, we skip the extraction of all the base features below f in the hierarchy. We can do that because the model has been transformed to incorporate the weights for all the skipped features into the corresponding feature weights associated with f. The weights for the skipped features are still kept in the model, so that they can be used when we encounter an unknown feature of the same type as f, such as an unknown whole word, or an unknown 4-character suffix, when we have seen the corresponding 3-character suffix.

The word-class-sequence features are arranged into a similar hierarchy, which is used in a similar way.

1. $\langle c(w_{i-2}), c(w_{i-1}), c(w_{i+1}), c(w_{i+2}) \rangle$
   - 1.1. $\langle c(w_{i-2}), c(w_{i-1}) \rangle$
   - 1.2. $\langle c(w_{i+1}), c(w_{i+2}) \rangle$
   - 1.3. $\langle c(w_{i-1}), c(w_{i+1}) \rangle$
     - 1.3.1. $c(w_{i-1})$
     - 1.3.2. $c(w_{i+1})$

Note that in this hierarchy, we have introduced a new feature type that does not actually exist in the trained model, the combination of the word-class bigrams preceding and following the word being tagged. The weights for the features of this type are constructed from the sums of the weights of other features lower in the hierarchy. To keep the size of the transformed model from exploding, we limit the instances of this feature type to those seen at least twice in the training data. We found this covered about 80% of the tagging decisions for the WSJ development set. We also included in the transformed model all possible instances (including those not observed in the training data) of the feature type 1.3 for word-class bigrams surrounding the word being tagged, which allows us to drop the feature weights for the lowest two feature types 1.3.1 and 1.3.2 after their feature weights have been added to the weights for the word-class-bigram features.

Altogether, these transformations increase the size of our full model from 151,174 features with 861,111 non-zero feature weights to 392,318 features with 17,047,515 non-zero feature weights. While this may be a substantial relative increase in size, the resulting model is still not particularly large in absolute terms.

Feature Weight Combination	Features per Token	Weights per Token	Tokens per Second	All Tag Error %	OOV Tag Error %
No	38.0	1215.0	1100	2.69	9.40
Yes	4.7	194.0	6400	2.69	9.40

Table 2: WSJ development set speeds and error rates without and with feature weight combination

In Table 2, we show the effect of these transformations on the speed of tagging the WSJ development set while considering all possible labels for each token. As expected, feature weight combination has no effect on tagging error, since it results in the same tagging decisions as the original model and feature extraction method. The "Features per Token" column shows the average number of features used for

[2]Note that we could have placed the "contains digit" or "contains hyphen" features under "lower-cased tokens" instead of "Shape 2"; our choice here was arbitrary.

each tagging decision, without and with the model and feature extraction feature-weight-combination transformations. The transformations reduce this number by a factor of 8.13. The "Weights per Token" column is the corresponding number of non-zero feature weights used for each tagging decision. Feature weight combination reduces this number by a factor of 6.26.

The "Tokens per Second" measurements are rounded to two significant digits due to the limited precision of our observations. Time was measured to the nearest second, and for each tagger, the data set was replicated enough times for the total tagging time to fall in the range of 100 to 200 seconds. The times reported include only reading the sentence-split tokenized text, extracting features, and predicting tags; time to read in model parameters and initialize the corresponding data structures is not inlcuded. Times are for a single-threaded implementation in Perl on a Linux workstation equipped with Intel Xeon X5550 2.67 GHz processors. In this implementation, feature weight combination increases tagging throughput by a factor of 5.82.

3.2 Pruning Possible Tags

It has long been standard practice to prune the set of possible tags considered for each word, in order to speed up tagging. Ratnaparkhi (1996) may have been the first to use the common heuristic of defining a tag dictionary allowing any tag for unknown words, but restricting each known word to the tags it was observed with in the training data. In addition, the tag dictionary for known words is sometimes further pruned (e.g., Banko and Moore, 2004; Giménez and Màrquez, 2004, 2012) according to the relative frequency of tags for each word. Tags observed in the training data with less than some fixed proportion of the occurrences of a particular word are not considered as possible tags for that word in test data.

In our experiments, we find these heuristics produce fast tagging, but lead to a noticable loss of accuracy, because known words are never allowed to be labeled with tags they were not observed with in the training data. This is similar to the problem of unseen n-grams in statistical language modeling, so we apply methods developed in that field to the problem of dictionary pruning for POS tagging. We construct our tag dictionary based on a "bigram" model of the probability $p(t|w)$ of a tag t given a word w, estimated from the annotated training data. The probabilities for tags that have never been seen with a given word, as well as all the tag probabilities for unknown words, are estimated by interpolation with a "unigram" distribution over the tags.

To estimate the probabilities of tags given words, we use the same interpolated-Kneser-Ney-smoothed (Chen and Goodman, 1999) model that we used in Section 2.2.1 in our supervised word-clustering procedure. In this model, we estimate the probabilty $p(t|w)$ by interpolating a discounted relative frequency estimate with a lower-order estimate of $p(t)$. The lower-order estimates are based on "diversity counts", taking the count of a tag t to be the number of distinct words ever observed with that tag. This has the desirable property for POS tagging that closed-class tags receive a very low estimated probabilty of being assigned to a rare or unknown word, even though they occur very frequently with a small number of frequent words. We use a single value for the discount parameter in the Kneser-Ney formula, chosen to maximize the estimated probability of the reference tagging of the development set. These probabilities are estimated ignoring distinctions among digit characters, just as in the features of our tagging model.

We construct our tag dictionary by setting a threshold on the value of $p(t|w)$. Whenever $p(t|w)$ is less than or equal to the threshold, the tag t is considered not to be a possible POS tag for the word w. Our preferred threshold ($p(t|w) > 0.0005$) is set to prune as agressively as possible while maintaining tagging accuracy on the WSJ development set. This threshold is applied to both known and unknown words, which produces 24 possible tags for unknown words by applying the threshold to the lower-order probability estimate $p(t)$. Note that the probabilities we use for pruning can be viewed as posteriors of a very simple POS tagging model, which makes inferring a tag dictionary an instance of coarse-to-fine inference with posterior pruning (Charniak et al., 2006; Weiss and Taskar, 2010).

The standard tag dictionary pruning heuristics can be viewed as a application of the same approach, but with the $p(t|w)$ probabilities being unsmoothed relative-frequency estimates for known words and a uniform distribution for unknown words. The original Ratnaparkhi heuristic amounts to thresholding these probabilities at 0, with a higher threshold being applied when using additional pruning.

Pruning Method	Tags per Token	Weights per Token	Tokens per Second	All Tag Error %	OOV Tag Error %	Seen Tag Error %	Unseen Tag Error %	OOV Mean Tags	Seen Mean Tags	Unseen Mean Tags
None	45.0	194.0	6400	2.69	9.40	2.19	52.0	45.0	45.0	45.0
Ratnaparkhi	3.7	19.0	47000	2.81	9.40	2.07	100.0	45.0	2.3	1.3
Ratnaparkhi+	2.9	14.3	56000	2.81	9.40	2.07	100.0	45.0	1.4	1.2
Kneser-Ney	3.5	16.1	49000	2.69	9.45	2.18	55.5	21.3	2.8	10.2
Kneser-Ney+	1.8	6.1	67000	2.81	9.74	2.14	83.8	10.6	1.4	2.8

Table 3: WSJ development set speeds and error rates for different tag dictionary pruning methods

In Table 3 we compare these methods of tag dictionary pruning on the WSJ development set, when combined with our feature-weight-combination technique. The "Tags per Token" column shows the average number of tags considered for each tagging decision, depending on the tag pruning method used. "Weights per Token", "Tokens per Second", "All Tag Error %", and "OOV Tag Error %" are as in Table 2. The first line repeats the experiment with no tag dictionary pruning from Table 2. The next line gives results for Ratnaparkhi's dictionary pruning method, and the next line, "Ratnaparkhi+", gives results for the maximum additional pruning by thresholding based on unsmoothed relative frequencies that does not increase overall tagging error ($p(t|w) > 0.005$). We see that these taggers are much faster than the unpruned tagger, but noticeably less accurate.

The final two lines of Table 3 are for our tag dictionary pruning method, with different pruning thresholds. The "Kneser-Ney" line represents our preferred threshold, set to prune as agressively as possible without noticeably degrading the overall tagging error on the WSJ development set. This produces a lower error rate than either Ratnaparkhi or Ratnaparkhi+ pruning, but Ratnaparkhi+ pruning results in faster tagging. However, if we increase the pruning threshold until we match the Ratnaparkhi+ error rate, as shown in the final "Kneser-Ney+" line, our method is faster than Ratnaparkhi+.

The remaining columns of Table 3 provide some insight as to why Kneser-Ney-smoothed pruning with our preferred threshold results in lower error than Ratnaparkhi and Ratnaparkhi+ pruning. The column labeled "Seen Tag Error %" is the error rate for examples with word/tag pairs seen in training. The column labeled "Unseen Tag Error %" is the error rate for examples with word/tag pairs not seen in training, but with a word that was seen in training. There are 660 of the latter examples in the WSJ development set, which amounts to 0.5% of that data set. By construction, the error rate of the Ratnaparkhi and Ratnaparkhi+ pruning methods on this subset of the data is 100%, but both the unpruned tagger and the tagger with Kneser-Ney-smoothed pruning correctly tag nearly half of these examples.

The Ratnaparkhi and Ratnaparkhi+ pruning methods are somewhat more accurate than the Kneser-Ney-smoothed pruning method on the seen word/tag pairs and the unknown words, but not enough to overcome the losses on the unseen word/tag pairs with known words. In absolute numbers on the WSJ development set, both the Ratnaparkhi and Ratnaparkhi+ pruning methods make 131 fewer errors on the seen word/tag pairs and 2 fewer errors on the unknown words, but 294 more errors on the unseen word/tag pairs with known words, compared to Kneser-Ney-smoothed pruning method with our preferred threshold. The final three columns of Table 3 show the mean number of tags allowed by each dictionary for these three categories of examples. Compared to Ratnaparkhi and Ratnaparkhi+ pruning, our preferred threshold for Kneser-Ney-smoothed pruning slightly increases the number of tags considered for seen word/tag pairs, substantially reduces the number of tags considered for unknown words, and substantially increases the number of tags considered for unseen word/tag pairs with known words.

4 Comparison to Other Taggers

We compared our tagger to several publicly available taggers, on the standard WSJ POS tagging test set. As far as we know, six taggers have been reported to have an error rate of less than 2.7% (accuracy greater than 97.3%) on this test set. Three of these are publicly available: the Stanford tagger (Toutanova et al., 2003; Manning, 2011), the Prague COMPOST tagger (Spoustová, et al., 2009), and the UPenn bidirectional tagger (Shen et al., 2007).[3] We tested two versions of the Stanford tagger, one based on their most accurate model "wsj-0-18-bidirectional-distsim", and one based on the much faster, but less accurate model "english-left3words-distsim" recommended for practical use on the Stanford tagger website. The UPenn tagger is run with a beam width of 3, which is the setting that gave their best reported results.

These taggers were all tested on on the same Linux workstation as our Perl tagger. To obtain comparable speed measurements omitting time for initialization, we performed two runs with each tagger. one on the first 1000 sentences of the test set, and another with those 1000 sentences followed by the entire test set replicated enough times to produce a difference in total time of at least 100 seconds. The tagging speed was inferred from the difference in these two times. The Stanford tagger reports tagging times directly, and these agreed with our measurements to two significant digits, which is the precision limit of our measurements.

We also report on the SVMTool tagger of Giménez and Màrquez (2004). Giménez recently provided us with benchmarks, which he obtained with a somewhat faster processor than ours, the Intel Xeon X5660 2.80 GHz. We give results for two versions of this tagger, one in Perl and one in C++, both with a combination of left-to-right and right-to-left tagging, which gives higher accuracy with this tagger than either direction by itself.

Tagger	Implementation Language	WSJ Tokens per Second	WSJ All Tag Error %	WSJ OOV Tag Error %	Brown Tokens per Second	Brown All Tag Error %	Brown OOV Tag Error %
This work	Perl	51000	2.66	9.02	40000	3.46	10.64
Stanford fast	Java	80000	3.13	10.31	50000	4.47	12.62
Stanford accurate	Java	5900	2.67	7.90	1600	3.86	11.21
COMPOST	C	2600	2.57	10.03	2700	3.36	12.16
UPenn	Java	270	2.67	10.39	290	3.90	12.96
SVMTool	Perl	1340	2.86	11.37			
SVMTool	C++	7700	2.86	11.37			

Table 4: WSJ test set and Brown corpus speeds and error rates compared to publicly available taggers

Results on the WSJ test set are shown in Table 4. We include a column giving the implementation language of each tagger to help interpret the results. Generally, we would expect an algorithm implemented in Perl to be slower than the same algorithm implemented in Java, which in turn would probably be slower than the same algorithm implemented in C/C++; although depending on the libraries used and the degree of optimization in the compilers, Java can sometimes be competitive with C/C++ (See, for example, `http://blog.famzah.net/2010/07/01/cpp-vs-python-vs-perl-vs-php-performance-benchmark/`).

"This work" refers to our tagger with feature weight combination and Kneser-Ney-smoothed dictionary pruning, with the pruning threshold set to maximize pruning without decreasing overall tagging accuracy on the WSJ development set. The fast Stanford tagger is the fastest overall by a wide margin, but it is also the least accurate. Our tagger is both the second fastest and the second most accurate, having an error rate relatively 3.9% higher (absolutely 0.09% higher) than the COMPOST tagger. But our tagger is almost 20 times faster than COMPOST, and more than 8 times faster than the accurate Stanford tagger, the second fastest tagger of equivalent or better accuracy. This is despite the fact that our tagger is written in Perl, while the other high-accuracy taggers are written either in Java or C.

As a final, out-of-domain evaluation, we ran the five taggers that we had direct access to on the Brown Corpus subset (3279 sentences, 83769 tokens) from the Penn Treebank. As might be expected, tagging was in general both slower and less accurate than on in-domain data. Our tagger maintained its relative position with respect to both speed and accuracy compared to all the other taggers. The only qualitative change in position of any tagger is that on the Brown Corpus data, the accurate Stanford tagger is slower than COMPOST, which actually runs faster than it does on the WSJ test set.

[3] A fourth tagger, the semi-supervised condensed nearest neighbor tagger of Søgaard (2011), has some released source code, but not a complete tagger nor detailed instructions on how to build the tagger Søgaard evaluates.

5 Conclusions

We have shown that a feature-rich model for POS tagging by independent classifiers can reach tagging accuracies comparable to several state-of-the art taggers, and we have introduced implementation strategies that result in much faster tagging than any other high-accuracy tagger we are aware of, despite these other taggers being implemented in faster programming languages.

A number of the techniques introduced here may have applications to other tasks. The sort of word-class-sequence models derived by supervised clustering described in Section 2.2 may be useful for other sequence labeling tasks, such as named-entity recognition. Our method of pruning the tag dictionary with smoothed probability distributions could also be used for label pruning for other problems with large label sets. Finally, the feature-weight-combination technique of Section 3.1 can be applied to any rich feature space in which the features have the kind of hierarchical structure we see in POS tagging. Such feature spaces are common in NLP, since we are almost always dealing with lexical items and their sublexical features.

Acknowledgements

Thanks to Chris Manning, John Bauer, and Jenny Liu for help with the Stanford tagger; to Johanka Spoustová and Jan Hajič, for help with the COMPOST tagger; to Jesús Giménez, for benchmarking the SVMTool tagger; and to Kuzman Ganchev and Dan Bikel, for valuable comments on earlier versions of this paper.

References

Michele Banko and Robert C. Moore. 2004. Part of speech tagging in context. In *Proceedings of the 20th International Conference on Computational Linguistics*, August 23–27, Geneva, Switzerland, 556–561.

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel coarse-to-ne PCFG parsing. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the ACL*, June 4–9, New York, New York, USA, 168–175.

Stanley F. Chen and Joshua T. Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13(4):359–393.

Michael Collins. 2002a. Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, July 6–7, Philadelphia, Pennsylvania, USA, 1–8.

Michael Collins. 2002b. Ranking algorithms for named-entity extraction: boosting and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association of Computational Linguistics*, July 7–12, Philadelphia, Pennsylvania, USA, 489–486.

Koby Crammer and Yoram Singer. 2001. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2:265–292.

Radu Florian, Abe Ittycheriah, Honyan Jing, and Tong Zhang. 2003. Named entity recognition through classifier combination. In *Proceedings of CoNLL-2003*, May 31–June 1, Edmonton, Alberta, Canada.

Inderjit S. Dhillon, Subramanyam Mallela, and Rahul Kumar. 2003. A divisive information-theoretic feature clustering algorithm for text classification. *Journal of Machine Learning Research*, 3:1265–1287.

Jesús Giménez and Lluís Màrquez. 2004. SVMTool: a general POS tagger generator based on support vector machines. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*, May 26–28, Lisbon, Portugal, 43–46.

Jesús Giménez and Lluís Màrquez. 2012. SVMTool Technical Manual v1.4. `http://www.lsi.upc.edu/~nlp/SVMTool/SVMTool.v1.4.pdf`

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technology*, June 3–8, Montreal, Quebec, Canada, 142–151.

Percy Liang, Hal Daumé III, and Dan Klein. 2008. Structure compilation: trading structure for features. In *Proceedings of the 25th International Conference on Machine Learning*, July 5–9, Helsinki, Finland, 592–599.

Christopher D. Manning. 2011. Part-of-speech tagging from 97% to 100%: Is it time for some linguistics? In Alexander Gelbukh (ed.), *Computational Linguistics and Intelligent Text Processing, 12th International Conference, CICLing 2011, Proceedings, Part I. Lecture Notes in Computer Science 6608*, Springer, 171–189.

Mitchell P. Marcus, Beatrice Santorini, and Mary A. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313-330.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association of Computational Linguistics*, June 25–30, Ann Arbor, Michigan, USA, 91–98.

Adwait Ratnaparkhi. 1996. A maximum entropy model for part-of-speech tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, May 17–18, Philadelphia, Pennsylvania, USA, 133–142.

Libin Shen, Giorgio Satta, and Aravind K. Joshi. 2007. Guided learning for bidirectional sequence classification. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, June 23–30, Prague, Czech Republic, 760–767.

Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab, and Miroslav Spousta. 2009. Semi-supervised training for the averaged perceptron POS tagger. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, March 30–April 3, Athens, Greece, 763–771.

Anders Søgaard. 2011. Semisupervised condensed nearest neighbor for part-of-speech tagging. In *Proceedings of the 49th Annual Meeting of the Association of Computational Linguistics*, June 19–24, Portland, Oregon, USA, 48–52.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, May 27–June 1, Edmonton, Alberta, Canada, 173–180.

David Weiss and Ben Taskar. 2010. Structured prediction cascades. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, May 13–15, Chia Laguna Resort, Sardinia, Italy, 916-923.

Tong Zhang. 2004. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the 21st International Conference on Machine Learning*, July 4–8, Banff, Alberta, Canada, 919–926.