Parse Reranking with WordNet Using a Hidden Variable Model

Terry Koo and Michael Collins

October 28, 2005

Contents

List of Figures

List of Tables

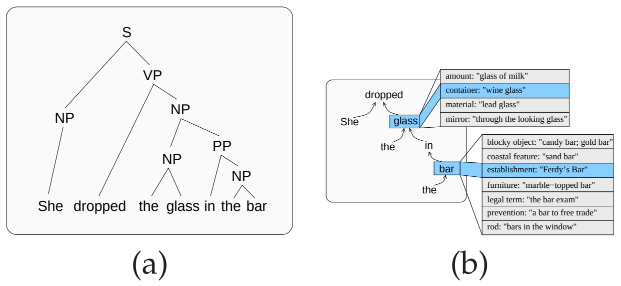
S
VP
NP
NP
PP
NP
NP
She dropped the glass in the bar
(a)
dropped
She
glass
the
in
bar
the
amount: "glass of milk"
container: "wine glass"
material: "lead glass"
mirror: "through the looking glass"
blocky object: "candy bar; gold bar"
coastal feature: "sand bar"
establishment: "Ferdy's Bar"
furniture: "marble−topped bar"
legal term: "the bar exam"
prevention: "a bar to free trade"
rod: "bars in the window"
(b)

Figure 1: Diagram (a) above shows a phrase structure tree, which gives rise to the dependency tree show in diagram (b). The nouns glass and bar have been expanded to show some of the possible word senses each can take on. The correct word senses are highlighted in blue.

1 High-Level Description

Our hidden variable model is a simple extension of the head-driven statistical models introduced in (Collins 1999) and (Charniak 1997). From each parse tree we produce a dependency tree, which captures the dependency relationships between headwords in the parse. Our main innovation is to treat this dependency tree as a pairwise Markov Random Field (MRF); each word in the dependency tree is given a hidden word sense, and these hidden senses interact along dependency arcs. Figure 2 illustrates the interaction of hidden word senses.

Our reranking algorithm is characterized by four elements. First, we make use of a feature vector representation, which reduces each MRF into a high-dimensional vector of feature counts. Second, we use a matched vector of parameters to define a probability distribution over the MRFs. Third, we define a loss function which approximates the training error of the algorithm. Fourth, we optimize the parameter vector through gradient descent on the loss function. The remainder of this section explains each of these four elements in greater detail.

1.1 Notation

We begin by defining notation and formally restating the problem. We are given a corpus of m training examples $x_1, \ldots, x_m$, where each example contains of a set of m_i candidate dependency trees $t_{i,1}, \ldots, t_{i,m_i}$. Each candidate tree is scored according to its adherence to the gold standard parses, and in each group of candidates, the highest-scored tree is specially labeled as $t_{i,1}$.

Recall that each word in the trees $t_{i,j}$ contains a hidden word sense. For convenience of notation, we define tree-wide *assignments* of word senses to all nodes. Note that the possible number of such assignments is exponential in the size of the tree; if there are n words and each word s_i possible

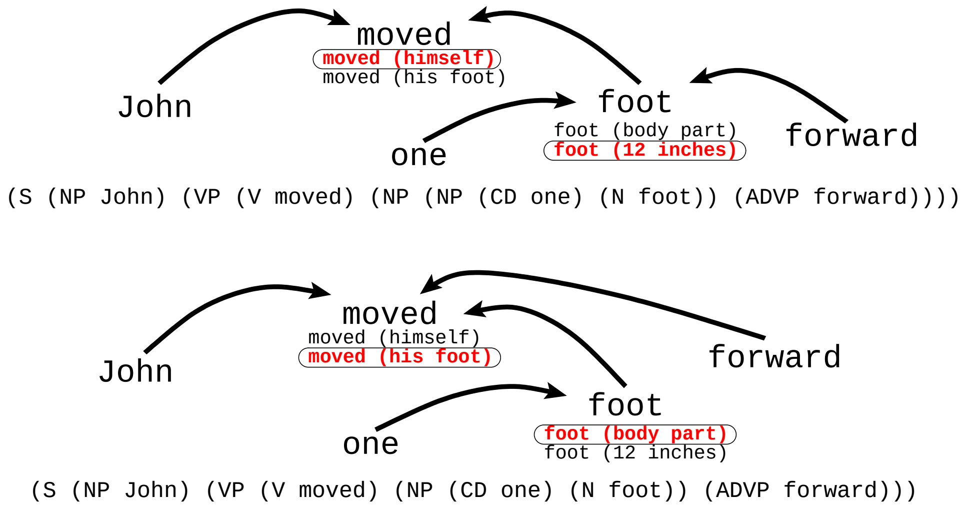

Figure 2: The sentence "John moved one foot forward" give rise to two the two alternate dependency trees above. The differing dependency structures, in turn, imply different plausible word sense assignments, colored red above.

senses, then there are $\prod_{i=1}^{n} s_i$ possible sense assignments.

Formally, let $m_{i,j}$ give the number of possible assignments to the hidden variables in candidate tree $t_{i,j}$; we label the assignments themselves as $a_{i,j,1}, \ldots, a_{i,j,m_{i,j}}$. Figure 3 arranges the various entities associated with a given data example x_i.

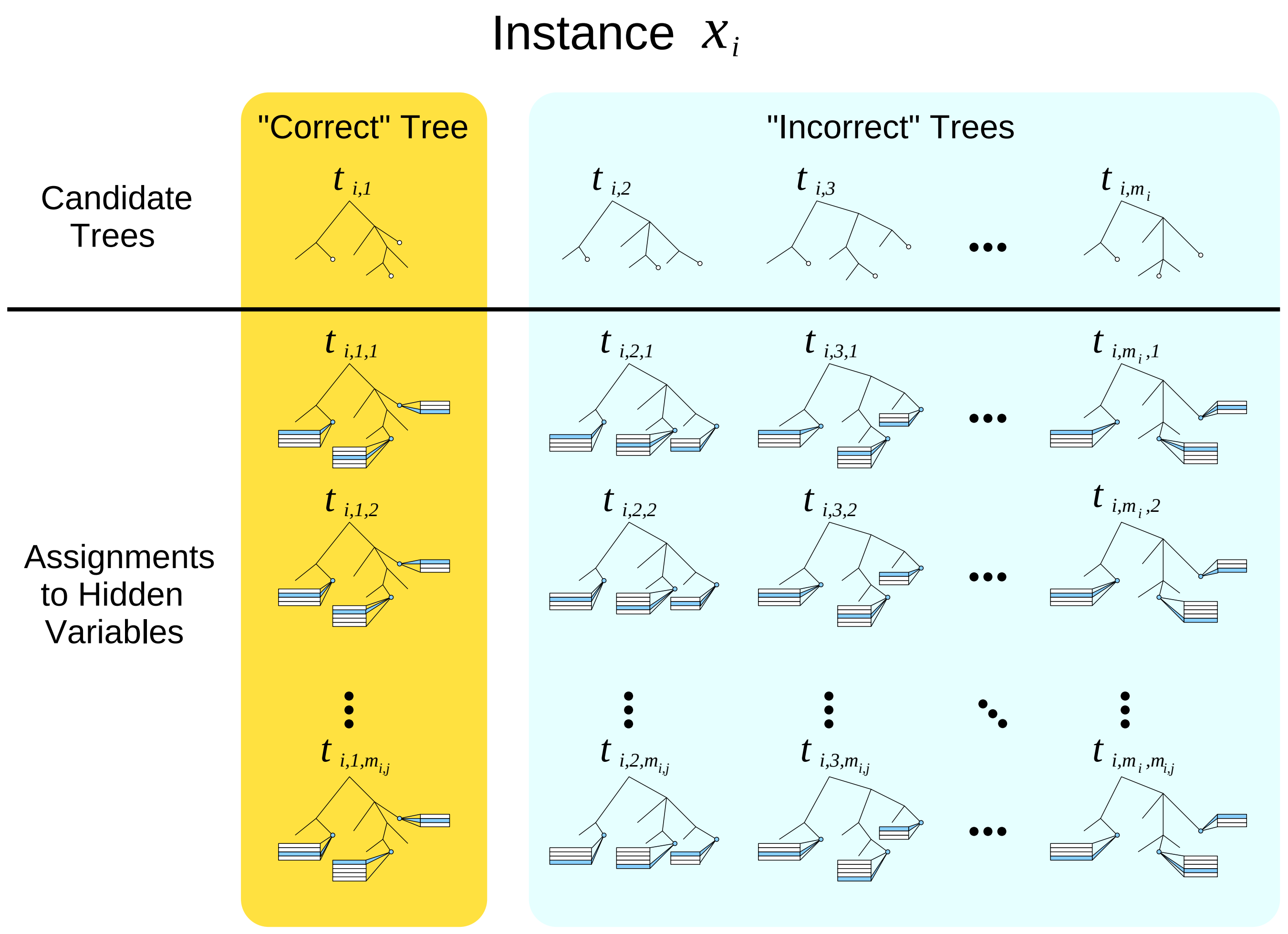

Figure 3: A table depicting the elements which compose a single reranking example x_i. The top row enumerates the candidate dependency trees $t_{i,j}$; note that the best-scored tree $t_{i,1}$ is labeled as "correct" tree, while the other trees are labeled "incorrect" trees. The assignments of hidden word senses are enumerated in columns below each candidate tree.

1.2 Feature-Based Probability Model

A dependency tree $t_{i,j}$ together with some sense assignment $a_{i,j,k}$ is reduced into a high-dimensional feature vector. We define the function $\Phi(i,j,k)$, which gives the feature vector representing tree $t_{i,j}$ with assignment $a_{i,j,k}$. Each dimension of the vector $\Phi(i,j,k)$ contains an occurrence count for some salient structure or relationship.

For example, $\Phi_1(i,j,k)$ might count the number of times the rule S $\Rightarrow$ NP VP appears in candidate parse $x_{i,j}$, while $\Phi_{100}(i,j,k)$ might count the number of times S $\Rightarrow$ NP VP appears with the WordNet synset company#1 (business institution) heading the NP.

However, we do not allow arbitrary features. Rather, we restrict ourselves to features which involve either a single word sense or a pair of word senses that are involved in a dependency relationship (i.e. they neighbor each other in the dependency tree). Although confining, this limitation allows us to bring powerful dynamic-programming methods to bear on the problem, as we will see in section 2.

To accompany the feature vectors, we define a dimensionally-matched parameter vector Θ. These parameters are used to induce a probability distribution

$$p(j,k \mid i,\Theta) = \frac{e^{\Phi(i,j,k)\cdot\Theta}}{\sum\limits_{j',k'} e^{\Phi(i,j',k')\cdot\Theta}}$$

which ranks the dependency trees and word sense assignments according to the likelihood that they yield the best-scoring structure and sense assignment. Note that according to the definition above, each dimension in Θ indicates the discriminative ability of the related feature. Continuing our example from above, if $\Theta_1 = 0.0000001$ and $\Theta_{100} = 0.5$, then we can conclude that the rule S $\Rightarrow$ NP VP by itself is not a good discriminator, but when seen with WordNet synset company#1, it is a strong indicator of a good parse.

We use the distribution $p(j,k \mid i,\Theta)$ above to define two additional probability distributions. First, by summing out the hidden word sense assignments, we obtain a distribution over candidate trees

$$p(j \mid i,\Theta) = \sum_k p(j,k \mid i,\Theta)$$

which gives the probability that dependency tree $t_{i,j}$ is the best-scoring tree. Second, by dividing the two previous distributions, we obtain a conditional probability distribution over the word sense assignments for a given dependency tree

$$p(k \mid i,j,\Theta) = \frac{p(j,k \mid i,\Theta)}{p(j \mid i,\Theta)}$$

1.3 Loss Function and Gradient

Ideally, the probability distribution p should satisfy the following property:

$$\forall i,\, \forall j > 1 \qquad p(1 \mid i, \Theta) > p(j \mid i, \Theta)$$

so that the best-scoring candidate tree $t_{i,1}$ is awarded the greatest share of the probability mass. Accordingly, the goal of the training phase of our algorithm is to generate parameters Θ that maximize the probability mass awarded to trees $t_{1,1}, t_{2,1}, \ldots, t_{m,1}$. This subsection captures this goal formally by defining a *loss function*[1]

$$\mathcal{L}(\Theta) = -\sum_i \log p(1 \mid i, \Theta)$$

which we will minimize by gradient descent. Substituting our definitions for the various probability distributions yields

$$\begin{aligned}
\mathcal{L}(\Theta) &= -\sum_i \log \sum_k p(1, k \mid i, \Theta) \\
&= -\sum_i \left[\log \sum_k \frac{e^{\Phi(i,1,k)\cdot\Theta}}{\sum\limits_{j',k'} e^{\Phi(i,j',k')\cdot\Theta}} \right] \\
&= \sum_i \left[-\log \left(\sum_k e^{\Phi(i,1,k)\cdot\Theta} \right) + \log \left(\sum_{j',k'} e^{\Phi(i,j',k')\cdot\Theta} \right) \right]
\end{aligned}$$

Our next step is to find an expression for $\partial\mathcal{L}/\partial\Theta$. For the sake of simplicity, we rewrite $\mathcal{L}$ in terms of functions F_i and G_i as follows:

$$\begin{aligned}
\mathcal{L}(\Theta) &= \sum_i \left(-F_i(\Theta) + G_i(\Theta) \right) \\
F_i(\Theta) &= \log \sum_k e^{\Phi(i,1,k)\cdot\Theta} \\
G_i(\Theta) &= \log \sum_{j,k} e^{\Phi(i,j,k)\cdot\Theta}
\end{aligned}$$

The gradient $\partial F_i/\partial\Theta$ is given by

[1] In practice, we wish to avoid overfitting the parameters to the training data, so we would actually use a loss function such as $\hat{\mathcal{L}}(\Theta) = -\sum_i \log p(1 \mid i, \Theta) + \frac{C}{2}\sum_m \Theta_m^2$, which adds a term penalizing the size of the parameters. We neglect this penalty term in our analysis as it adds nothing conceptually important to our description and can be dealt with trivially.

$$\begin{aligned}
\frac{\partial F_i}{\partial \Theta} &= \frac{\partial}{\partial \Theta} \log \sum_k e^{\Phi(i,1,k)\cdot\Theta} \\
&= \frac{\sum\limits_k \Phi(i,1,k) e^{\Phi(i,1,k)\cdot\Theta}}{\sum\limits_{k'} e^{\Phi(i,1,k')\cdot\Theta}} \\
&= \sum_k \Phi(i,1,k) \left(\frac{e^{\Phi(i,1,k)\cdot\Theta}}{\sum\limits_{k'} e^{\Phi(i,1,k')\cdot\Theta}} \right) \\
&= \sum_k \Phi(i,1,k) \left(\frac{e^{\Phi(i,1,k)\cdot\Theta} \Big/ \sum\limits_{q,r} e^{\Phi(i,q,r)\cdot\Theta}}{\sum\limits_{k'} e^{\Phi(i,1,k')\cdot\Theta} \Big/ \sum\limits_{q',r'} e^{\Phi(i,q',r')\cdot\Theta}} \right) \\
&= \sum_k \Phi(i,1,k) \left(\frac{p(1,k \,|\, i,\Theta)}{\sum\limits_{k'} p(1,k' \,|\, i,\Theta)} \right) \\
&= \sum_k \Phi(i,1,k) \left(\frac{p(1,k \,|\, i,\Theta)}{p(1 \,|\, i,\Theta)} \right) \\
&= \sum_k \Phi(i,1,k) p(k \,|\, i,1,\Theta) \\
&= \mathbf{E}_p[\Phi(i,1,k)]
\end{aligned}$$

where $\mathbf{E}_p[\Phi(i,j,k)]$ is the feature vector produced by candidate tree $x_{i,j}$ in expectation, under the probability distribution p. Graphically, $\mathbf{E}_p[\Phi(i,j,k)]$ is the weighted average of the feature vectors produced by the j^{th} column of Figure 3. We now turn our attention to $\partial G_i/\partial\Theta$:

$$\begin{aligned}
\frac{\partial G_i}{\partial \Theta} &= \frac{\partial}{\partial \Theta} \log \sum_{j',k'} e^{\Phi(i,j',k')\cdot\Theta} \\
&= \frac{\sum\limits_{j,k} \Phi(i,j,k) e^{\Phi(i,j,k)\cdot\Theta}}{\sum\limits_{j',k'} e^{\Phi(i,j',k')\cdot\Theta}} \\
&= \sum_{j,k} \Phi(i,j,k) \frac{e^{\Phi(i,j,k)\cdot\Theta}}{\sum\limits_{j',k'} e^{\Phi(i,j',k')\cdot\Theta}} \\
&= \sum_{j,k} \Phi(i,j,k) p(j,k \,|\, i,\Theta) \\
&= \sum_{j,k} \Phi(i,j,k) p(k \,|\, i,j,\Theta) p(j \,|\, i,\Theta) \\
&= \sum_j p(j \,|\, i,\Theta) \sum_k \Phi(i,j,k) p(k \,|\, i,j,\Theta)
\end{aligned}$$

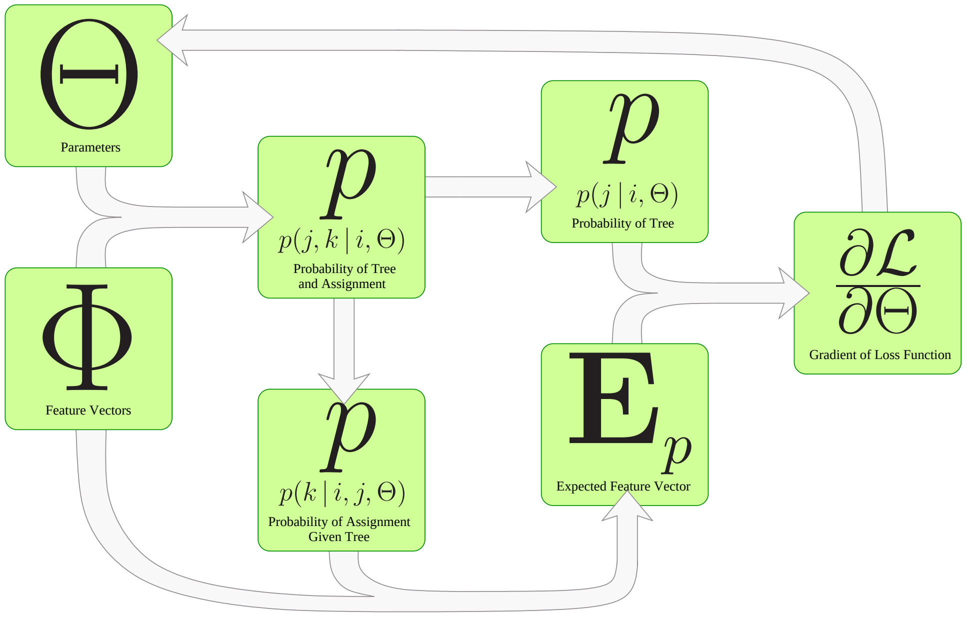
Θ
Parameters
Φ
Feature Vectors
p
$p(j, k \,|\, i, \Theta)$
Probability of Tree and Assignment
p
$p(k \,|\, i, j, \Theta)$
Probability of Assignment Given Tree
p
$p(j \,|\, i, \Theta)$
Probability of Tree
$\mathbf{E}_p$
Expected Feature Vector
$\frac{\partial \mathcal{L}}{\partial \Theta}$
Gradient of Loss Function

Figure 4: A flowchart depicting how the major elements of the reranking algorithm interact with each other.

$$= \sum_j p(j \,|\, i, \Theta)\mathbf{E}_p[\Phi(i, j, k)]$$

Therefore, $G_i(\Theta)$ is equal to the expected feature vector produced by the *entire* example x_i. Referring back to Figure 3 once more, we can think of $G_i(\Theta)$ graphically as the weighted average of the feature vectors produced by the entire table. Finally, we substitute our results above into the expression for $\partial\mathcal{L}/\partial\Theta$:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \Theta} &= \sum_i \left(-\frac{\partial F_i}{\partial \Theta} + \frac{\partial G_i}{\partial \Theta} \right) \\ &= \sum_i \left(-\mathbf{E}_p[\Phi(i, 1, k)] + \sum_j p(j \,|\, i, \Theta)\mathbf{E}_p[\Phi(i, j, k)] \right) \end{aligned}$$

Note that $\frac{\partial \mathcal{L}}{\partial \Theta}$ is completely expressed in terms of the two functions $p(j \,|\, i, \Theta)$ and $\mathbf{E}_p[\Phi(i, j, k)]$. This property figures importantly in Section 2, where we will show how p and $\mathbf{E}_p$, and hence the gradient, can be computed efficiently.

1.4 Summary

In summary, our reranking algorithm proceeds as follows. We first produce feature vectors from every dependency tree $t_{i,j}$ and sense assignment $a_{i,j,k}$. We initialize the parameters Θ, creating a probability distribution over trees and assignments. Next, we use the gradient of the loss function to slightly shift Θ in a beneficial direction; the process of calculating the gradient and shifting the parameters is repeated. The iteration is ended by a cutoff based either on the number of iterations or the amount of change resulting in the loss function. The flowchart in Figure 4 lays out the high-level operation of the algorithm.

2 Incorporating Belief Propagation

The reranking algorithm as described in the previous section is quite inefficient. The gradient must be recomputed on each iteration of gradient descent, which means that the algorithm must repeatedly evaluate

$$\mathbf{E}_p[\Phi(i,j,k)] \quad \text{and} \quad p(j \mid i, \Theta) \qquad \forall i, j$$

The formulae for $\mathbf{E}_p[\Phi(i,j,k)]$ and $p(j \mid i, \Theta)$ given in Section 1 require an enumeration over all possible assignments to hidden variables, a combinatorially complex task. In addition, feature vectors must be produced for every tree $t_{i,j}$ and sense assignment $a_{i,j,k}$; again, an combinatorially-sized task. However, recall from Section 1.2 that we imposed the following restriction on our features:

(i) A feature can involve at most two hidden word senses,

(ii) If a feature involves two senses, these two must be involved in a dependency relationship; that is, they must be linked by an edge in the dependency tree.

In this section, we show how these restrictions allow us to make use of belief propagation as a module which computes $\mathbf{E}_p[\Phi(i,j,k)]$ and $p(j \mid i, \Theta)$ in linear time (Yedidia et al. 2002). Additionally, the restrictions permit a decomposed feature vector representation which sidesteps the problem of enumerating a combinatorially-sized set of feature vectors.

The remainder of this section is divided into four subsections: the first subsection introduces new notation, the second subsection describes the belief propagation algorithm, and the third subsection explains how we use belief propagation to create p and $\mathbf{E}_p$, and the fourth subsection summarizes our modifications to the reranking algorithm.

2.1 Notation

We narrow the scope of our analysis to a single dependency tree $t_{i,j}$. Let the nodes in the tree be numbered $1, 2, \ldots, n$ and let $N(u)$ give the set of nodes neighboring u. Let the hidden word sense of node u be $s_u \in S_u$, where S_u is the set of possible senses of the word at node u. As before, we use the notation $a_{i,j,k}$ denote an assignment of word senses to all nodes.

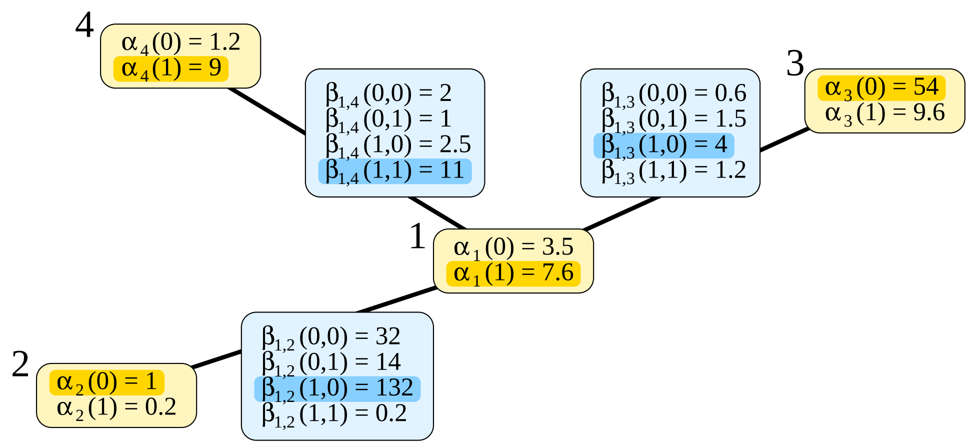

Figure 5: An example graph over binary-valued hidden variables in which the values of the weight functions α_u and $\beta_{u,v}$ have been filled in. As the highlighted values indicate, the probability distribution $p(a_{i,j,k})$ generated from this example would favor the assignment $a_{i,j,k} = (s_1 = 1, s_2 = 0, s_3 = 0, s_4 = 1)$.

The input to belief propagation is a set of node weights $\alpha_u \in S_u \mapsto \mathbb{R}$ and edge weights $\beta_{u,v} \in (S_u \times S_v) \mapsto \mathbb{R}$ such that $\alpha_u(s_u)$ gives a measure of the appropriateness of sense s_u being assigned to the word at node u, and $\beta_{u,v}(s_u, s_v)$ gives a measure of the appropriateness of values s_u and s_v appearing on edge (u, v) in conjunction[2]. These weight functions define the following probability distribution over assignments $a_{i,j,k}$:

$$p\left(a_{i,j,k}\right) = \frac{1}{Z_{i,j}} \prod_{w} \alpha_w(s_w) \prod_{u<v} \beta_{u,v}(s_u, s_v)$$

where $Z_{i,j}$ is a normalizing constant that ensures $\sum_k p(a_{i,j,k}) = 1$. Figure 5 depicts the functions α_u and $\beta_{u,v}$ for a small sample graph of four nodes.

The belief propagation algorithm produces three items as output. The first output is a set of node *beliefs* $b_u \in S_u \mapsto \mathbb{R}$ which satisfy the following properties:

1. $b_u(s_u)$ indicates the appropriateness of assigning sense s_u to the word at node u, and

2. $\sum\limits_{s_u \in S_u} b_u(s_u) = 1$

In fact, since we are performing belief propagation on a tree, the node beliefs b_u actually give the exact marginalized probability distribution at each node u; that is,

$$b_u(x_u) = \sum_{a_{i,j,k} \mid s_u = x_u} p(a_{i,j,k})$$

[2]Note that β must also satisfy $\beta_{u,v}(s_u, s_v) = \beta_{v,u}(s_v, s_u)$, so that the edge weights are undirected.

The second output is a set of pairwise beliefs $b_{u,v} \in (S_u \times S_v) \mapsto \mathbb{R}$ which, in the tree case, give the exact marginalized distributions over pairs of nodes; that is:

$$b_{u,v}(x_u, x_v) = \sum_{a_{i,j,k} \mid s_u = x_u, s_v = x_v} p(a_{i,j,k})$$

Proof that the node and edge beliefs are equal to the marginal probability distributions in the tree case can be found in Appendix A. Finally, the third output of belief propagation is the normalization constant $Z_{i,j}$ associated with $p(a_{i,j,k})$; this constant will play an important role in the efficient computation of $p(j \mid i, \Theta)$.

2.2 Mechanics of Belief Propagation

The core of belief propagation is the dynamic-programming technique known as the *message passing* algorithm, which allows linear-time[3] computation of all three outputs of belief propagation: node beliefs, pairwise beliefs, and normalization factor. In the interests of clarity, we will describe only the essentials here, leaving the details of a linear-time implementation and the necessary complexity analysis to Appendix B.

In the message passing algorithm, every node u transmits a message $m_{u \to v} \in S_v \mapsto \mathbb{R}$ to each of its neighbors v, where $m_{u \to v}(s_v)$ gives an indication of how strongly node u believes node v should have sense s_v. These messages are determined recursively by the following:

$$m_{u \to v}(s_v) = \sum_{s_u \in S_u} \alpha_u(s_u) \beta_{u,v}(s_u, s_v) \prod_{w \in N(u) \setminus v} m_{w \to u}(s_u)$$

so that the message from u to v is a combination of the messages received from u's other neighbors. In the case of a tree, the following scheme suffices to calculate the messages: pick an arbitrary root node, then compute messages from the leaves in towards the root and then from the root back out to the leaves. Figure 6 depicts this upstream-downstream process.
After all of the messages have been computed, the node beliefs b_u and pairwise beliefs $b_{u,v}$ are given by

$$\begin{aligned} b_u(s_u) &= \frac{1}{z_u} \alpha_u(s_u) \prod_{v \in N(u)} m_{v \to u}(s_u) \\ b_{u,v}(s_u, s_v) &= \frac{1}{z_{u,v}} \alpha_u(s_u) \alpha_v(s_v) \beta_{u,v}(s_u, s_v) \prod_{s \in N(u) \setminus v} m_{s \to u}(s_u) \prod_{t \in N(v) \setminus u} m_{t \to v}(s_v) \end{aligned}$$

where z_u and $z_{u,v}$ are normalizing factors which ensure that

[3]Time linear in the number of nodes in the tree.

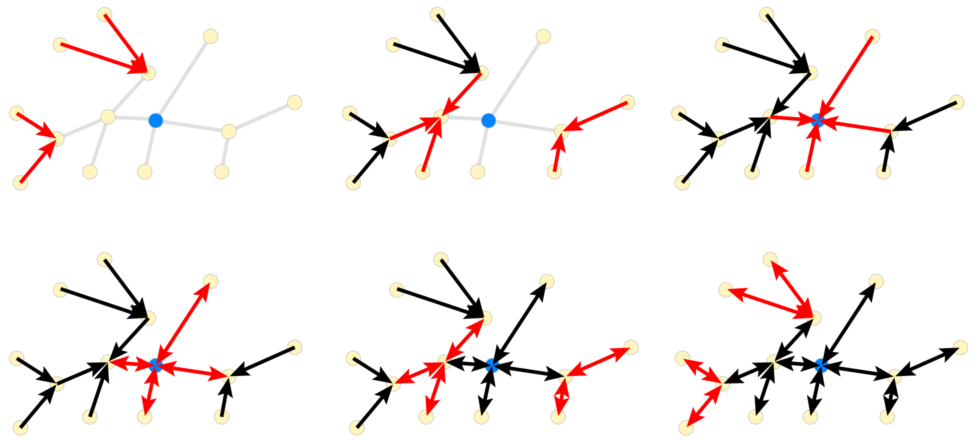

Figure 6: The steps of the message-passing algorithm on a small example graph. An arbitrarily chosen root has been colored blue, and the messages are shown as arrows.

$$\sum_{s_u \in S_u} b_u(s_u) = 1 \qquad \text{and} \qquad \sum_{s_u \in S_u,\, s_v \in S_v} b_{u,v}(s_u, s_v) = 1$$

We now turn our attention to the third output of belief propagation: the normalization factor $Z_{i,j}$. Fortunately, it turns out that

$$\forall u, v z_u = z_{u,v} = Z_{i,j}$$

so that any of the node or pairwise normalization constants can serve as the tree-wide normalization constant. This equivalence is a side effect of the proof that tree beliefs are equal to marginal probabilities; see the end of Appendix A, page 37.

2.3 Computing p and $\mathbf{E}_p$

We use belief propagation as a module within the larger reranking algorithm. The inputs to belief propagation are carefully prepared so that its outputs give rise to $p(j \mid i, \Theta)$ and $\mathbf{E}_p[\Phi(i, j, k)]$. We begin by observing that the conditional distribution $p(k \mid i, j, \Theta)$ from Section 1 and the distribution $p(a_{i,j,k})$ from Section 2.1 both compute the same probability distribution.

Now, recall that our features are restricted to either single word senses or pairs of senses joined by a dependency. Therefore, we can decompose the tree-wide feature vector $\Phi(i, j, k)$ into a set of node feature vectors $\phi_u(s_u)$ and pairwise feature vectors $\phi_{u,v}(s_u, s_v)$:

$$\Phi(i, j, k) = \left(\sum_u \phi_u(s_u)\right) + \left(\sum_{u<v} \phi_{u,v}(s_u, s_v)\right)$$

Suppose we use this decomposition to define the node and pairwise weight functions below:

$$
\begin{aligned}
\alpha_u(s_u) &= e^{\phi_u(s_u)\cdot\Theta} \\
\beta_{u,v}(s_u, s_v) &= e^{\phi_{u,v}(s_u,s_v)\cdot\Theta}
\end{aligned}
$$

Then, the probability distribution $p(a_{i,j,k})$ can be simplified as follows:

$$
\begin{aligned}
p(a_{i,j,k}) &= \frac{1}{Z_{i,j}} \prod_w e^{\phi_w(s_w)\cdot\Theta} \prod_{u<v} e^{\phi_{u,v}(s_u,s_v)\cdot\Theta} \\
&= \frac{1}{Z_{i,j}} e^{\left(\sum_w \phi_w(s_w)\right)\cdot\Theta} e^{\left(\sum_{u<v} \phi_{u,v}(s_u,s_v)\right)\cdot\Theta} \\
&= \frac{1}{Z_{i,j}} e^{\Phi(i,j,k)\cdot\Theta}
\end{aligned}
$$

and from the equality of $p(a_{i,j,k})$ and $p(k \,|\, i, j, \Theta)$ we can derive an alternate expression for the value of $Z_{i,j}$:

$$
\begin{aligned}
p(a_{i,j,k}) &= p(k \,|\, i, j, \Theta) \\
\frac{1}{Z_{i,j}} e^{\Phi(i,j,k)\cdot\Theta} &= \frac{e^{\Phi(i,j,k)\cdot\Theta}}{\sum_k e^{\Phi(i,j,k)\cdot\Theta}} \\
Z_{i,j} &= \sum_k e^{\Phi(i,j,k)\cdot\Theta}
\end{aligned}
$$

Define $Z_i = \sum_j Z_{i,j}$, and note that our new knowledge about $Z_{i,j}$ gives us a method for computing $p(j \,|\, i, \Theta)$ in $O(1)$ time per candidate tree

$$
p(j \,|\, i, \Theta) = \frac{\sum_k e^{\Phi(i,j,k)\cdot\Theta}}{\sum_{j',k'} e^{\Phi(i,j',k')\cdot\Theta}} = \frac{Z_{i,j}}{Z_i}
$$

Now, consider the quantity $\mathbf{E}_p[\Phi(i, j, k)]$; we reproduce its definition below:

$$
\mathbf{E}_p[\Phi(i, j, k)] = \sum_k p(k \,|\, i, j, \Theta)\Phi(i, j, k)
$$

We substitute in the equivalent probability distribution $p(a_{i,j,k})$ and decompose the feature vector $\Phi(i, j, k)$, simplifying as follows:

$$\begin{aligned}
\mathbf{E}_p[\Phi(i,j,k)] &= \sum_{a_{i,j,k}} p(a_{i,j,k}) \left(\left(\sum_w \phi_w(s_w) \right) + \left(\sum_{u<v} \phi_{u,v}(s_u, s_v) \right) \right) \\
&= \left(\sum_{u<v} \sum_{a_{i,j,k}} p(a_{i,j,k}) \phi_{u,v}(s_u, s_v) \right)
\end{aligned}$$

Recall that $\phi_w(s_w)$ is only sensitive to the word sense s_w. If we break the set of possible assignments $a_{i,j,k}$ into equivalence classes for which the sense s_w is the same, the value of $\phi_w(s_w)$ will remain constant within each equivalence class. This observation gives rise to the following simplification:

$$\begin{aligned}
\sum_w \sum_{a_{i,j,k}} p(a_{i,j,k}) \phi_w(s_w) &= \sum_w \sum_{x_w \in S_w} \phi_w(s_w) \sum_{a_{i,j,k} \,|\, s_w = x_w} p(a_{i,j,k}) \\
&= \sum_w \sum_{x_w \in S_w} \phi_w(s_w) b_w(s_w)
\end{aligned}$$

and similar reasoning applies to the pairwise feature vectors:

$$\begin{aligned}
\sum_{u<v} \sum_{a_{i,j,k}} p(a_{i,j,k}) \phi_{u,v}(s_w) &= \sum_{u<v} \sum_{x_u \in S_u,\, x_v \in S_v} \phi_{u,v}(s_u, s_v) \sum_{a_{i,j,k} \,|\, s_u = x_u,\, s_v = x_v} p(a_{i,j,k}) \\
&= \sum_{u<v} \sum_{x_u \in S_u,\, x_v \in S_v} \phi_{u,v}(s_u, s_v) b_{u,v}(s_u, s_v)
\end{aligned}$$

Therefore, if there are n nodes in the tree, the expected feature vector $\mathbf{E}_p$ can be computed in $O(n)$ time with the following expression:

$$\mathbf{E}_p[\Phi(i,j,k)] = \left(\sum_w \sum_{x_w \in S_w} \phi_w(s_w) b_w(s_w) \right) + \left(\sum_{u<v} \sum_{x_u \in S_u,\, x_v \in S_v} \phi_{u,v}(s_u, s_v) b_{u,v}(s_u, s_v) \right)$$

2.4 Summary

In summary, we have seen how restricting features to pairs of senses allows us to apply the belief propagation algorithm, a powerful dynamic-programming technique. Whereas a naive algorithm would require time exponential in the size of the dependency trees, belief propagation requires only linear time. The flowchart in Figure 7 updates Figure 4 to show how belief propagation replaces several items in the high-level algorithm.

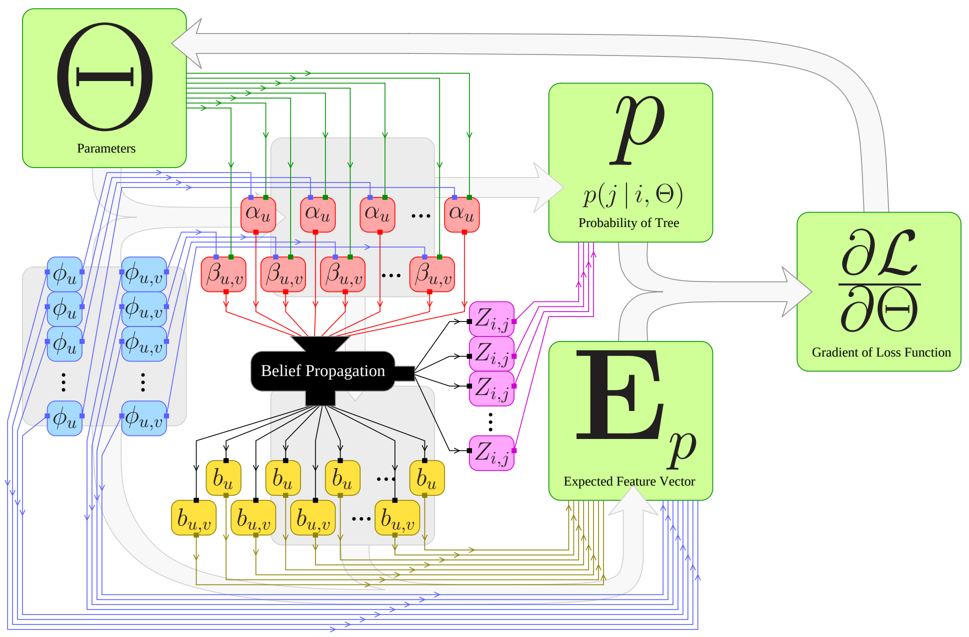

Figure 7: A revised version of the flowchart in Figure 4 that depicts use of belief propagation in the implementation of the reranking algorithm. Note that some high-level elements from the original flowchart have been faded out and overwritten with belief propagation items. Although these high-level elements have been replaced they still exist abstractly, implicitly defined by the belief propagation machinery which replaces them. For instance, the feature vector Φ has been decomposed into smaller feature vectors ϕ_u and $\phi_{u,v}$. The conditional distribution p is represented by the beliefs b_u and $b_{u,v}$. The distribution p is represented by the weight functions α_u and $\beta_{u,v}$.

3 Stochastic Gradient Descent

Another source of inefficiency in the high-level description of Section 1 is in the use of gradient descent. In every round of standard gradient descent, a line search is required to determine how far along the gradient the parameters Θ should be adjusted. Every point tested by this line search requires a pass over the training set to determine the value of $\mathcal{L}(\Theta)$ at that point, and completing just a single round of gradient descent would require several passes over the training set.

Since plain gradient descent is prohibitively expensive, we use a form of stochastic gradient descent. We iterate through the training examples, and for each example we calculate the gradient arising from that example alone. The parameters are then perturbed in the direction of the gradient, where the distance along the gradient is determined by a learning rate η. The learning rate is a decaying quantity given by

$$\eta \quad = \quad \frac{a_0}{1+ct}$$

where t is the number of examples which have been used so far and a_0 and c are parameters of stochastic gradient descent.

4 Data Sets

The data sets consist of parsed output produced by the Collins (1999) parser on the Penn Treebank, with an average of roughly 30 candidate parses per sentence. Section 23 is held out as a final test set, and the remainder of the sentences were divided into a training corpus of 35,540 sentences and a development-test corpus of 3,676 sentences.

These training, devtest, and final test corpuses are the same which were used to produce the Collins (2000) boosting reranker. In order to measure how much improvement our own reranker can provide beyond the boosting reranker, lists of the features that the boosting reranker used during training, development testing, and final testing were obtained. There were a total of 11,673 distinct boosting features with an average of roughly 40 features per tree. The next section discusses how we used these features in combination with our own reranker.

5 Incorporating Other Rankings

In the preceding descriptions of the reranking algorithm, we have always discussed the reranker as a stand-alone entity. However, we have found it useful to integrate our reranker with the rankings of other models. The following two subsections describe how we integrate the rankings of the base parser and the (Collins 2000) boosting reranker with our own reranker.

5.1 Integrating the Base Parser

The base parser produces an initial ranking over the candidate parses with its probability model. Although our reranker replaces that initial ranking, by no means is the ranking completely discarded; our aim in reranking is to supplement, rather than supplant, the original base model. We establish a special per-tree feature which $\phi_{\text{logp}}(i,j)$, which contains the log-probability assigned by the base parser to tree $t_{i,j}$. This feature is assigned a parameter θ_{logp}, and our probability model is adjusted to include this new feature as follows:

$$p(j \mid i, \Theta, \theta_{\text{logp}}) \quad = \quad \frac{e^{\phi_{\text{logp}}(i,j)\theta_{\text{logp}}} \sum_{k} e^{\Phi(i,j,k)\cdot\Theta}}{\sum_{j'} e^{\phi_{\text{logp}}(i,j')\theta_{\text{logp}}} \sum_{k'} e^{\Phi(i,j',k')\cdot\Theta}}$$

Unlike the normal features, however, we do not optimize the parameter θ_{logp} during stochastic gradient descent. Our reasoning is that the polarity of the log-probability feature will change frequently from example to example. In examples where the base parser's ranking is correct, the gradient $\frac{\partial \mathcal{L}_i}{\partial \theta_{\text{logp}}}$ will be strongly positive, but when the base parser makes a mistake, the gradient $\frac{\partial \mathcal{L}_i}{\partial \theta_{\text{logp}}}$ will be strongly negative. Since $\phi_{\text{logp}}(i, j)$ is such a powerful feature, our belief is that the back-and-forth oscillations in the value of θ_{logp} would strongly hinder the training process. Therefore, in our experiments we initialize θ_{logp} to a static value before training and keep it constant throughout. The value of θ_{logp} which we used in our final testing was chosen through validation on the development data set.

5.2 Integrating the Boosting Reranker

The original motivation behind making use of WordNet was to improve upon the performance obtained by the (Collins 2000) boosting reranker. In order to make the comparison as fair as possible, we incorporate the features chosen by the boosting reranker into our own reranker. Like the base parser's log-probability, the boosting reranker's features are per-tree features which are insensitive to word senses. Let $\Phi_{\text{boost}}(i, j)$ give the boosting feature vector for tree $t_{i,j}$, and let Θ_{boost} be the matched vector of parameters. We incorporate the boosting features into our probability model as follows:

$$p(j \mid i, \Theta, \Theta_{\text{boost}}, \theta_{\text{logp}}) = \frac{e^{\phi_{\text{logp}}(i,j)\theta_{\text{logp}}} e^{\Phi_{\text{boost}}(i,j)\cdot\Theta_{\text{boost}}} \sum_k e^{\Phi(i,j,k)\cdot\Theta}}{\sum_{j'} e^{\phi_{\text{logp}}(i,j')\theta_{\text{logp}}} e^{\Phi_{\text{boost}}(i,j')\cdot\Theta_{\text{boost}}} \sum_{k'} e^{\Phi(i,j',k')\cdot\Theta}}$$

and the gradient $\frac{\partial \mathcal{L}_i}{\partial \Theta_{\text{boost}}}$ is given by

$$\frac{\partial \mathcal{L}_i}{\partial \Theta_{\text{boost}}} = -\Phi_{\text{boost}}(i, 1) + \sum_j \Phi_{\text{boost}}(i, j) p(j \mid i, \Theta, \Theta_{\text{boost}}, \theta_{\text{logp}})$$

We attempted to integrate the boosting features in two ways. First, we altered our stochastic gradient descent algorithm to train both sets of parameters together; that is, for every example x_i, the algorithm performed:

$$\begin{aligned} \Theta &\leftarrow \Theta - \eta(tn + i)\frac{\partial \mathcal{L}_i}{\partial \Theta} \\ \Theta_{\text{boost}} &\leftarrow \Theta_{\text{boost}} - \eta_{\text{boost}}(tn + i)\frac{\partial \mathcal{L}_i}{\partial \Theta_{\text{boost}}} \end{aligned}$$

Note that we applied different learning rates to each parameter vector. We conducted experiments where set of parameters was trained in isolation, and it was clear that the boosting features performed best with a more aggressive learning rate than the newer features[4]. Therefore, when optimizing both sets of parameters simultaneously, we decided to keep the learning rates separate.

Unfortunately, selecting parameters for the two learning rates proved to be a difficult task. Simply reusing the learning rates that worked best for each set of features in isolation produced poor results; no doubt interactions between the two feature sets were invalidating the old learning rates. However, with two independent learning rates, exploring the possible space of learning rate parameters became prohibitively expensive.

Accordingly, we turned to a second, simpler integration method. We trained each reranker in isolation and then combined the two rankings with a weighted average. Therefore, when testing our probability model was effectively

$$p(j \mid i, \Theta, \Theta_{\text{boost}}, \theta_{\text{logp}}) \quad = \quad \frac{e^{\phi_{\text{logp}}(i,j)\theta_{\text{logp}}} e^{C_{\text{boost}}\Phi_{\text{boost}}(i,j)\cdot\Theta^*_{\text{boost}}} \sum_{k} e^{C\Phi(i,j,k)\cdot\Theta^*}}{\sum_{j'} e^{\phi_{\text{logp}}(i,j')\theta_{\text{logp}}} e^{C_{\text{boost}}\Phi_{\text{boost}}(i,j')\cdot\Theta^*_{\text{boost}}} \sum_{k'} e^{C\Phi(i,j',k')\cdot\Theta^*}}$$

where Θ^* and Θ^*_{boost} are the optimized parameters from the isolated training runs, and C and C_{boost} are the parameters of the weighted average. The particular Θ^*, Θ^*_{boost}, C, and C_{boost} that we used in our final testing were chosen by validation on the development data set.

6 Feature Sets

We trained and tested our reranker using several different feature sets. All of our feature sets, however, make use of WordNet (Miller et al. 1993) and have the same basic composition. The remainder of this section first describes the basic structure of our feature sets, then discusses some of the issues with this basic model, and finishes by explaining how our feature sets addressed these issues through various extensions upon the basic model.

6.1 Basic Feature Composition

Each feature set is divided into two kinds of features: single-sense features that are aimed at establishing a prior probability distribution over word sense assignments, and pairwise features that attempt to capture head-modifier relationships between word senses. Often, however, there may not be any word sense available; currently, we only retrieve WordNet synsets for nouns, and some nouns (especially proper names) do not

[4]In fact, even among the various types of new features we experimented with (these are described further in Section 6), each feature set required a different learning rate.

appear in WordNet. Therefore, when we fail to obtain a word sense, we simply substitute the bare word for the missing sense and treat the node as having a single word sense. Each node feature is a tuple consisting of four elements:

Word	The bare word at that node.
Sense	The word sense assigned to the word.
POS	The part-of-speech tag of the word.
Label	The nonterminal label that the word receives as it modifies its target; i.e. the label of the highest nonterminal to which this word propagates as a headword.

Including the word as well as the sense gives our reranker a handle on the prior probability distribution over each word's word senses, and the other information can provide additional clues. For instance, the part of speech tag specifies plurality of nouns, which can sometimes aid in word sense disambiguation; consider "sense" versus "senses": the plural is more likely to take on the sense of "the five senses" or "word senses" whereas the singular is more likely to take on the sense of "common sense" or "sense of security".

However, these 4-tuples can be quite specific, so we implement several levels of backoff. In particular, for every node u and word sense s_u, we produce the following node features:

$$\phi_u(s_u) \quad = \quad \left\{ \begin{array}{l} \big(\text{Word}_u,\ \ s_u,\ \ \text{POS}_u,\ \ \text{Label}_u\big) \\ \big(\text{Word}_u,\ \ s_u,\ \ \text{POS}_u,\qquad\quad\ \big) \\ \big(\text{Word}_u,\ \ s_u,\qquad\qquad \text{Label}_u\big) \\ \big(\text{Word}_u,\ \ s_u,\qquad\qquad\qquad\ \big) \\ \big(\qquad\quad\ \ s_u,\ \ \text{POS}_u,\ \ \text{Label}_u\big) \\ \big(\qquad\quad\ \ s_u,\ \ \text{POS}_u\qquad\quad\ \big) \\ \big(\qquad\quad\ \ s_u,\qquad\qquad \text{Label}_u\big) \end{array} \right.$$

Note that in the case where the word sense is nonexistent, we would only generate the last three of the node features listed above. Moving on, the pairwise features are tuples consisting of the following elements:

Modifier Sense	The word sense of the modifier in the dependency relationship.
Modifier POS	The part-of-speech tag of the modifier.
Head Sense	The word sense of the head in the dependency relationship.
Head POS	The part-of-speech tag of the head.
Production Label	The nonterminal label of the constituent produced by this head-modifier interaction.
Modifier Label	The nonterminal label of the head.
Head Label	The nonterminal label of the modifier.
Dominates Conjunction	True if the head dominates a word with POS tag `CC`.
Adjacency	True if the modifier's constituent neighbors the head's constituent.
Left/Right	Whether the modifier is on the left or right of the head.

The triple of nonterminal labels provides information about the kind of dependency. For instance, a subject-verb relationship would be reflected by the triple

$$\big(\text{PLbl},\ \text{MLbl},\ \text{HLbl}\big) \quad = \quad \big(\texttt{S},\ \texttt{NP},\ \texttt{VP}\big)$$

while the argument of a transitive verb might produce the triple (`VP`, `NP`, `VB`). However, the location information provided by the Adjacency and Left/Right features are necessary to further determine the type of dependency relationship. For instance, in the sentence "Yesterday the market fell two points," the triple (`S`, `NP`, `VP`) describes the temporal modification between "Yesterday" and "fell" as well as the subject-verb interaction between "market" and "fell", and in the sentence "The market fell two points yesterday," the triple (`VP`, `NP`, `VB`) ambiguously refers to either a verb-argument dependency or a temporal modification. To resolve these ambiguities, we can consult the adjacency element: the subject-verb and verb-argument dependencies produce +ADJ, while the temporal modifiers produce -ADJ.

The Dominates Conjunction feature can resolve other ambiguous triples. For example, the phrases “boys and girls” and “satellite communications” would both yield the triple (`NP`, `NP`, `NP`). However, in the former case, the dependency is a coordination between two like nouns, while in the latter case, the dependency is a restrictive modification of one noun by another. Using the Dominates Conjunction element, we can distinguish between the two: the coordination would produce +CC, while the modification would produce -CC.

Even more so than the node features, the pairwise features can be overly specific, so we use backed-off features to combat data sparseness. We first remove the two part of speech tags, and then we remove the three binary-valued elements. We also generate features where the either the head or modifier sense is removed, leaving only the part of speech tag. Thus, the full set of features generated by each pairwise word sense interaction is:

$$\phi_{u,v}(s_u, s_v) \;=\; \left\{ \begin{array}{llllllllll}
\big(s_u, & \text{POS}_u, & s_v, & \text{POS}_v, & \text{PLbl}, & \text{MLbl}, & \text{HLbl}, & \pm\text{CC}, & \pm\text{ADJ}, & \text{L/R}\big) \\
\big(s_u, & & s_v, & & \text{PLbl}, & \text{MLbl}, & \text{HLbl}, & \pm\text{CC}, & \pm\text{ADJ}, & \text{L/R}\big) \\
\big(s_u, & & s_v, & & \text{PLbl}, & \text{MLbl}, & \text{HLbl} & & & \big) \\
\big(s_u, & & & \text{POS}_v, & \text{PLbl}, & \text{MLbl}, & \text{HLbl}, & \pm\text{CC}, & \pm\text{ADJ}, & \text{L/R}\big) \\
\big(s_u, & & & \text{POS}_v, & \text{PLbl}, & \text{MLbl}, & \text{HLbl} & & & \big) \\
\big(& \text{POS}_u, & s_v, & & \text{PLbl}, & \text{MLbl}, & \text{HLbl}, & \pm\text{CC}, & \pm\text{ADJ}, & \text{L/R}\big) \\
\big(& \text{POS}_v, & s_v, & & \text{PLbl}, & \text{MLbl}, & \text{HLbl} & & & \big)
\end{array} \right.$$

6.2 Issues with the Basic Model

First of all, there is the issue of choosing which senses to use when producing features. WordNet provides an index which maps words to sets of senses; we call these immediately-available senses the “literal” word senses for a word. Unfortunately, these literal senses are much too fine-grained to use, even with the use of the backed-off features. Consider the word “chocolate”, to which WordNet assigns the following three senses:

(1) A beverage prepared from cocoa, milk, and sugar, as in “hot chocolate”,

(2) A solid substance made from roasted ground cacao beans, as in “chocolate bar” or “chocolate chips”, and

(3) A deep brown color.

The literal senses are clearly informative, but they provide far too much specificity, given the limited limited size of our training set. It is difficult to imagine features using literal senses being much more informative than plain lexicalized features.

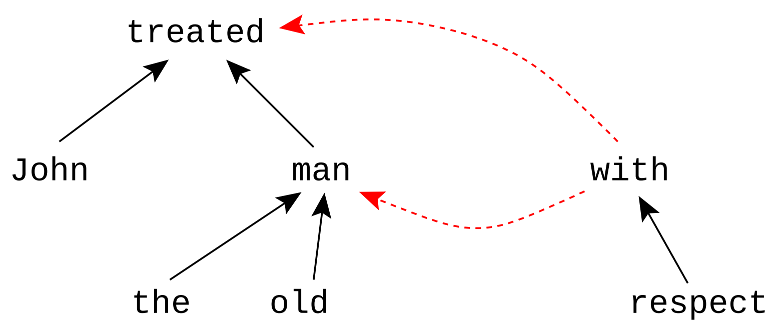

Figure 8: A dependency tree for the sentence "John treated the old man with respect."

Our intuition is that the important information lies somewhere above these lowest-level senses. While the knowledge that "chocolate" can refer to a chocolate-based beverage, chocolate-based food, and chocolate-colored color is almost useless, knowing that "chocolate" can refer to a beverage, solid food, or color in general is quite powerful. Two of the following subsections describe our attempts to recover this kind of knowledge through WordNet supersenses and hypernyms.

Another issue, and unfortunately one which we have only begun to address recently, is the interposition of function words at key points in the dependency tree. Most notably, the headword of a prepositional phrase is the preposition. Consider Figure 8, which displays a dependency tree containing a PP-attachment ambiguity.

Note that the preposition "with" interposes between its noun argument "respect" and the potential targets "treated" and "man". The function word "with" will only be assigned a single sense, and as we explained in Section 1 our features are restricted to pairs of neighboring word senses. Therefore, in our model the PP argument is unable to have any effect on the attachment preference of the prepositional phrase; the prepositional phrase "with respect" in Figure 8 would be treated no differently than the phrases "with arthritis" or "with aspirin". Many studies (Ratnaparkhi and Roukos 1994; Collins and Brooks 1995) have shown that the PP argument is essential in resolving PP-attachment ambiguities. Consider that the

We explored two possibilities for resolving this problem. First, we ran preliminary experiments in which each preposition was given two word senses. Variation of this binary word sense allowed a measure of information to pass through the preposition, so that the PP argument could affect the attachment preferences to some degree. Our experiments showed that this method offered some increase in performance; however, the gains were by no means large. Recently, we have tried a second approach that involves transforming the dependency tree to bring the PP argument into direct contact with its attachment site. We describe these efforts in the last subsection.

`noun.Tops`	`noun.act`	`noun.animal`
`noun.artifact`	`noun.attribute`	`noun.body`
`noun.cognition`	`noun.communication`	`noun.event`
`noun.feeling`	`noun.food`	`noun.group`
`noun.location`	`noun.motive`	`noun.object`
`noun.person`	`noun.phenomenon`	`noun.plant`
`noun.possession`	`noun.process`	`noun.quantity`
`noun.relation`	`noun.shape`	`noun.state`
`noun.substance`	`noun.time`	

Table 1: A listing of the 26 WordNet noun lexicographer filenames, which we use as "supersenses."

6.3 Supersenses

As we mentioned earlier, one of the issues with our basic feature model is the overspecified nature of literal word senses. One method by which we access higher-level information is through the use of WordNet "supersenses."

Every WordNet sense is processed from source material in a lexicographer file. WordNet noun senses originate from 26 such files, which are organized along general semantic boundaries; their filenames are given in Table 1. We use these lexicographer filenames as "supersenses", an idea we borrow from (Johnson and Ciaramita 2003).

For every word, we first retrieve its literal senses, and from each sense we derive its supersense, discarding the original senses. For example, the noun "crate" has two literal senses: a box-like object ("a wooden crate"), or the quantity held in a crate ("a crate of toys"), which give rise to the supersenses `noun.artifact` and `noun.quantity`. However, note that this process can reduce the number of senses the word is assigned. For instance, the noun "car" has five literal senses, but all five are members of `noun.artifact`, so we treat "car" as having only a single word sense.

The advantage of the supersense features is that the small number of supersenses yield a small number of features, making for more dependable training and generalization. However, the same coarseness which makes the supersenses train and generalize well also means that there is less information available in them.

6.4 Hypernyms

A second way in which we access higher-level information is through the use of WordNet *hypernymy*. Beyond the word to sense index and the supersenses, WordNet contains a great deal of information about the relationships between word senses. Noun hypernymy is a WordNet relation

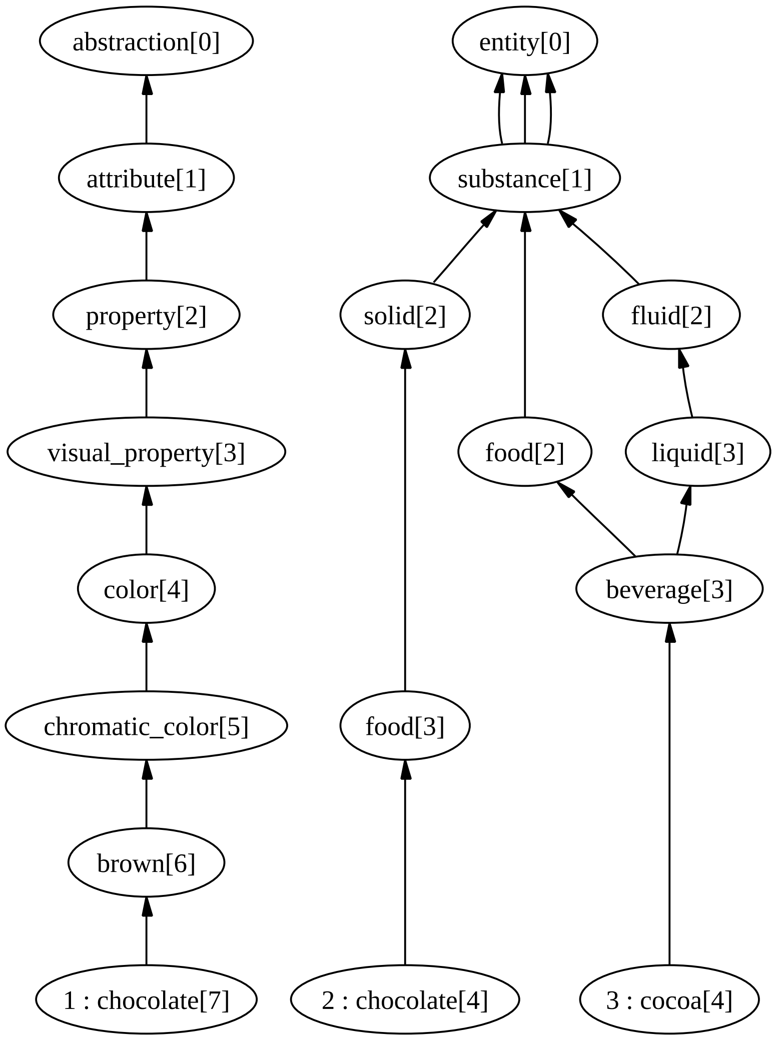

Figure 9: A depiction of the hypernym graph that arises from the literal senses of the word "chocolate."

which organizes noun senses according to the hierarchical[5] "is-a" relationship; for instance, "brown" is a hypernym of "chocolate" because chocolate is-a brown color.

By repeatedly following hypernym pointers, a series of gradually broader senses can be established, starting from the overspecified literal senses to a set of 9 top-level[6] hypernyms (see Figure 9 for an example). We believe that useful word sense information lies somewhere between the two

[5] Actually, WordNet hypernymy does not define a true hierarchy, as some senses may have more than one hypernym; for instance, the hypernyms of "wheeled vehicle" are "vehicle" and "container". For simplicity, however, we will continue to refer to hypernymy as a hierarchical relationship.

[6]Note that these top-level hypernyms are *not* the same as the supersenses described above; the supersenses derive from the 26 noun lexicographer filenames, and are not necessarily related to the hypernymy structure.

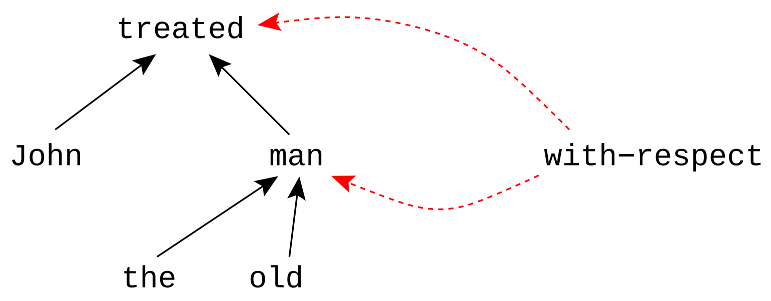

Figure 10: A PP-ambiguous dependency in which the preposition has been merged with its noun argument.

extremes of granularity; however, locating the exact depth of the useful region is tricky, as not all hypernym paths are the same length.

Rather than attempt to specify the useful depth, then, we simply produce features for all possibly hypernyms, and leave it to the learning method to sort out which hypernyms are useful. To give a specific example, consider the word "chocolate", which has three senses: "chocolate (beverage)", "chocolate (solid food)", and "chocolate (color)". For the solid food sense, we would produce features for "chocolate (solid food)", "food", "solid", "substance", and "entity"; and likewise for the other two senses. However, note that in our reranking model, the word "chocolate" would still have only three word senses, but each of these senses would carry features for all of its hypernyms.

The drawback of generating features for all hypernyms is of course an explosion in the number of features. When we generate pairwise features, we must not only process all pairs of word senses, but for each pair of senses, we must create features for all possible pairs of the *hypernyms* of the two senses involved, compounding the two quadratic costs.

6.5 Tree Transformations

As we have mentioned, one of the drawbacks of our current model is that function words often interpose in critical junctions of the dependency tree. Therefore, we have begun work on a method for transforming the dependency trees so that function words are merged with their arguments. The senses of the merged node would be conjunctions of the function word and the senses of its argument.

For example, Figure 10 shows how this transformation technique would alter the dependency tree from Figure 8. The supersenses of "with-respect" would be given as:

"with-`noun.cognition`"	"with-`noun.state`"
"with-`noun.act`"	"with-`noun.feeling`"
"with-`noun.attribute`"	

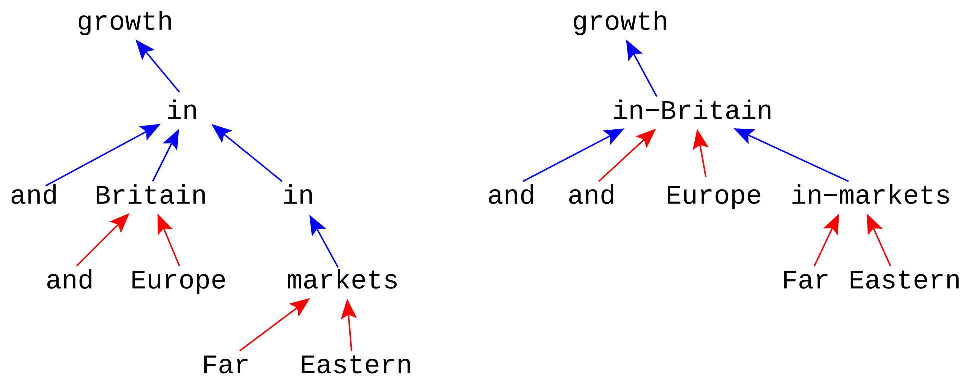

Figure 11: Dependency trees for the prepositional phrase "[growth] in Britain and Europe, and in Far Eastern markets" before and after transformations. The dependency arcs are colored according to whether the dependency originally modified a preposition or noun; note that after the transformation, some of the children of "in-Britain" originally modified "Britain" while others originally modified "in".

With this definition of the senses, the pairwise features that arise from the dependency between "with-respect" and "treated" or the dependency between "with-respect" and "man" would capture interactions between the senses of "respect" and the senses of "treated" or "man", as well as the preposition "with". Therefore, the reranking model would be able to learn to disambiguate this PP-attachment ambiguity.

The general idea behind these tree transformations is simple, but in a full implementation, there many tricky details that need to be addressed, and there is often no clear technique for resolving them. For instance, although the noun and preposition share the same node, we should still document the dependency between the preposition and its argument. Therefore, we define the node feature set of a merged node as containing all of the features that would normally arise from the two nodes when separate. Returning to the example of Figure 10, the node features of "with-respect" would include all the normal node features of "with" and "respect", as well as all of the normal *pairwise* features arising from the dependency between "with" and "respect".

By the same token, however, we should also preserve the features that would normally arise between the preposition and its other neighbors, as well as the features that would normally arise between the noun argument and its other neighbors. For example, consider the dependency trees shown in Figure 11, which depict a complex prepositional phrase before and after transformations are applied. From the dependency between "growth" and "in-Britain", we should produce the pairwise features that arise from the senses of "growth" and "in" as well as those from "growth" and "in-Britain". Similarly, from the dependency between "in-Britain" and "Europe", we should produce the pairwise features that arise from the senses of "Britain" and "Europe", as well as those from "in-Britain" and "Europe". However, note that we should *not* produce features derived

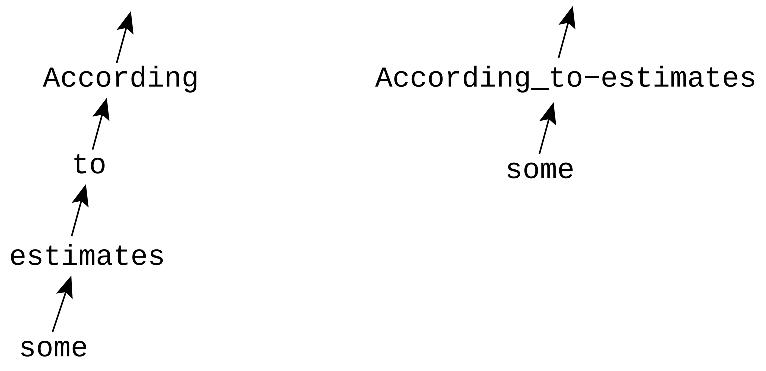

Figure 12: Dependency trees for the prepositional phrase "According to some estimates" both before and after transformations.

from "growth" and "Britain" or "in" and "Europe", as these dependencies do not exist in the untransformed tree. The coloration of the dependency arcs in Figure 11 reflects whether the noun or preposition should be used in the production of these preserved features.

A last issue is how we deal with cascaded prepositions, such as the example in Figure 12. Our current approach is to merge all chains of preposition nodes, concatenating the text of the prepositions and treating them as a single preposition. Note that this approach would discard the preposition-to-preposition dependencies in the original tree and alter the preposition-to-noun features. In the example in Figure 12, we would discard the pairwise features arising from "According" and "to", and instead of producing pairwise features for "to" and "estimates", we would produce features for "According-to" and "estimates".

Finally, although we have consistently used prepositional phrases as examples, these tree transformations can easily be applied to other interposing function words. Figure 13 shows some other transformations that could prove useful. Our work with tree transformations is still in its early stages, and we have no experimental results to report.

7 Results and Discussion

This section describes our experimental results and discusses their implications. We report development test results for the hypernym and supersense feature sets; as the supersense features outperformed the hypernym features, we only evaluated the supersense features in our final tests. Our experiments fall into two categories: reranking tests, which measure the additive improvement of the new reranker over the base parser, and combined reranking tests, which measure the additive improvement of the new reranker over the Collins (2000) boosting reranker. The remainder of this section presents our experimental results for both kinds of tests, and concludes with a discussion the deficiencies of our reranking model that

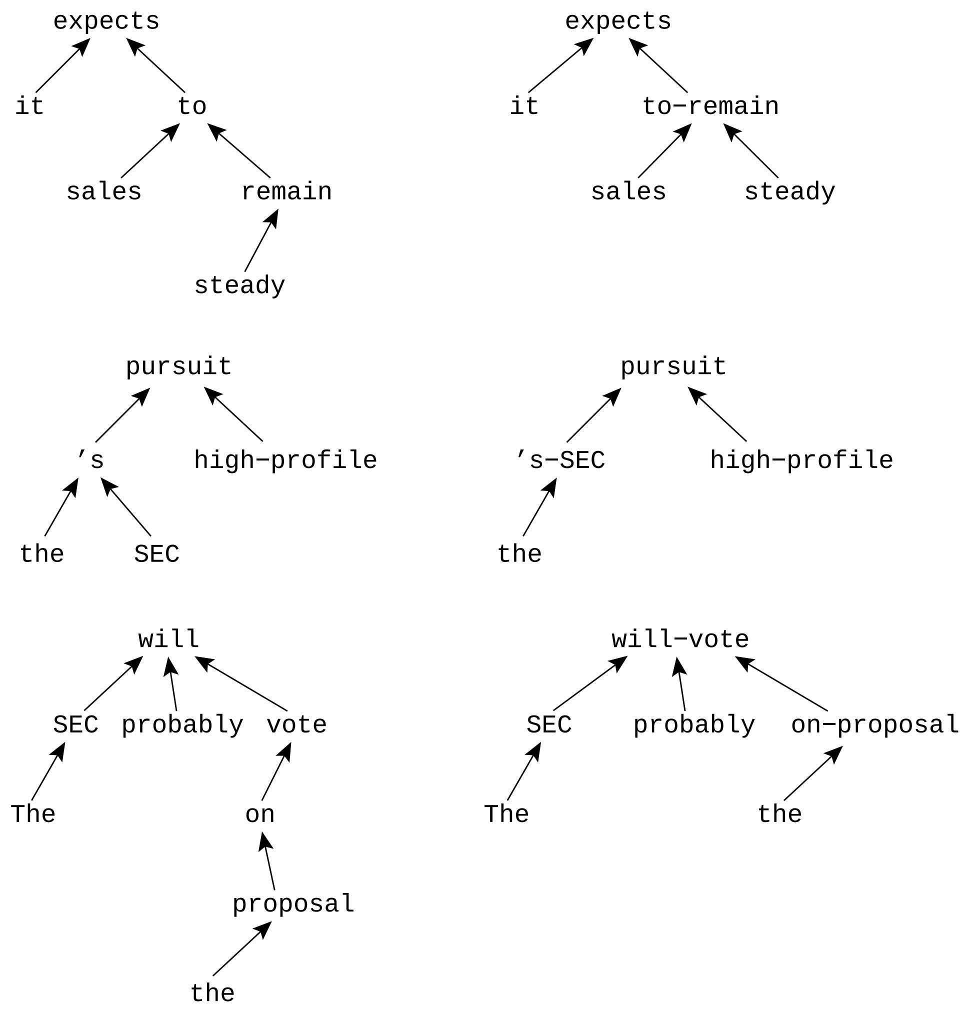

Figure 13: Dependency trees depicting transformations which operate on possessive markers and verbal auxiliaries such as the infinitival "to" and modal verbs. Each transformation removes an intervening functional word, allowing the meaningful words to interact directly. The text of the phrases are "it expects sales to remain steady", "the SEC's high-profile pursuit", and "The SEC will probably vote on the proposal".

could be alleviated to yield better performance.

7.1 Reranking Tests

The point of comparison for the reranking tests was the `logp` baseline, which is simply the score of the base parser by itself. Accordingly, for a fair comparison, we integrated the base parser's ranking with our reranker as described in Section 5.1.

In tests on the development set, the hypernym feature set achieved an improvement of $\approx 0.685\%$ over the `logp` baseline. The supersense features, on the other hand, achieved an improvement of $\approx 0.972\%$ past baseline, a significant gain. Figure 14 shows the scores achieved by both hypernym and supersense features as training progressed.

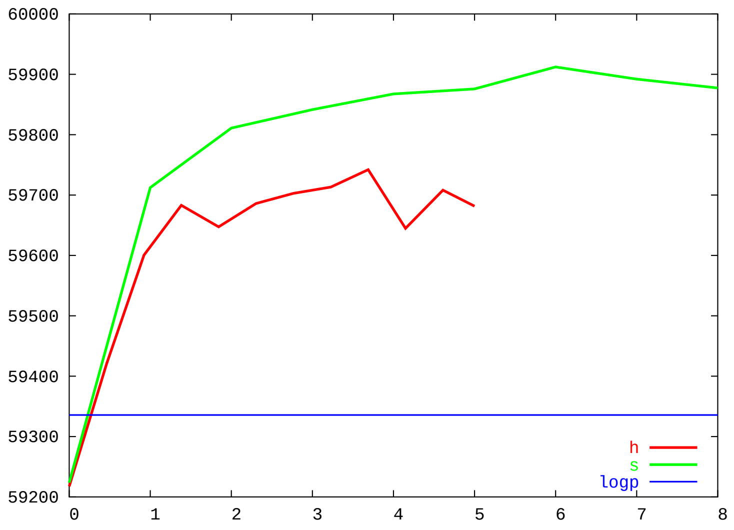

Figure 14: The development-set scores of the hypernym (h) and supersense (s) features are graphed versus the number of passes over the training set.

Given the generality of the supersense categories, we expected the supersense features to generalize well and were supported by our tests on the development set. The inclusion of backed-off features turned out to be crucial; without backoff, the supersense features achieve only a $\approx 0.77\%$ improvement over baseline, dropping by a factor of about $1/5$. Figure 15 graphs the performance of the backed-off and non-backed-off features.

Nevertheless, the supersense feature set achieved an improvement of only $\approx 0.5\%$ over baseline on the section 23 test set. We attribute at least some of this drop to overspecialization of the model toward the development set. We hope to avoid such problems in the future by using a multi-way averaging scheme; we would train and optimize several different rerankers on different development sets, and combine their output using a weighted average.

A somewhat unexpected result was the relative absence of overtraining in the hypernym features. The entire hypernym feature space contains over 30 million features, while the training set spans only about a million trees. Nevertheless, the hypernym features managed to achieve and maintain nontrivial gains. One possible explanation could be that the reranking algorithm has a tendency to assign more weight to frequent features. Since the high-level hypernyms appear most frequently, they take control of the hypernym feature set, and the hypernym feature set effectively migrates toward a supersense-like feature set.

7.2 Combined Reranking Tests

In our combined reranking tests, we compared our scores to the `boost` baseline, which is the score of the base parser augmented by the (Collins

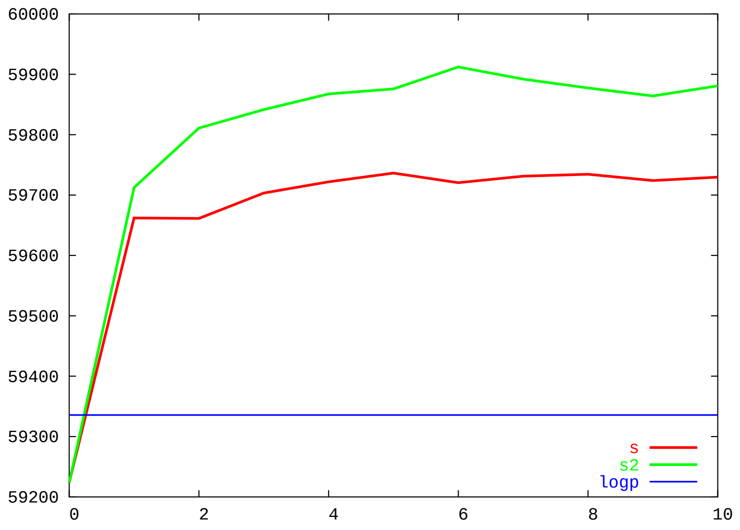

Figure 15: The development-set scores of the the backed-off supersense features (`s2`) and non-backed-off features (`s`) are graphed for every pass over the training set.

2000) boosting reranker. In order to measure the true additive improvement, we integrated the boosting reranker and base parser's rankings with our reranker, as described in Sections 5.1 and 5.2.

As was remarked earlier, we discovered that training both sets of parameters in conjunction failed to produce good results. However, by using a weighted combination of new and old rerankers, we were able to obtain small improvements. For the supersense features, the optimal weighting yielded a development set improvement of $\approx 0.153\%$ past the `boost` baseline. Unfortunately, this improvement did not carry over to the final test results.

7.3 Deficiencies of the Reranking Model

One of the biggest deficiencies of our current reranking model is that our feature sets currently only assign noun word senses. This restriction was originally imposed in order to keep the implementation simple. However, it has become apparent that word senses for other parts of speech should also be used.

The power of our hidden variable model is in its ability to model sense-to-sense interactions, yet when the model is restricted to noun senses only, there are few sense-to-sense interactions. Noun-noun dependencies occur in only a handful of situations: noun-noun restrictive modification (as in "satellite communications"), appositions, and conjunctions. These noun-noun interactions are typically quite short-range, near the leaves of the parse tree, and therefore they have only a small effect on the correctness of a parse.

If we included verb senses, we would greatly increase the amount of sense-to-sense interaction. Moreover, verb-noun interactions typically span a larger region of the parse tree, being closer to the core structure of a parse; therefore, we can expect to make stronger gains by learning verb-noun interactions.

In addition, if we apply the tree transformations described in Section 6.5, we could also derive noun-noun dependencies from instances of prepositional phrase modification. These interactions are longer range and usually quite ambiguous, so we could also stand to gain much from using tree transformations.

Another deficiency of our reranker arises from the predominance of proper nouns in the Wall Street Journal corpus. Naturally, WordNet cannot be expected to provide coverage of these proper nouns. In fact, the use of WordNet on proper nouns can sometimes cause misleading sense interactions. For example, consider "Apple Computer, Inc.", in which the fruit sense of "Apple" would be assigned. Another kind of confusion is exemplified by the name "John", which WordNet gives the senses of "slang for toilet", "king of England", "apostle", and "part of the bible" (i.e. the gospel according to John). We might address this difficulty by making use of a named-entity tagger, or even by abstaining from assigning word senses to proper nouns at all.

The example of "John" also points out another drawback of using WordNet: the frequent occurrence of rare or irrelevant "outlier" senses. For example, one of the senses of "investment" is "a covering of an organ or organism", and the senses of "man" include "Isle of Man" and "board game piece" (e.g. chess man). Although our reranking model should be able to learn to avoid these outlier senses, confusions are still possible, and there is no reason to heap so much responsibility on the reranker.

One way to overcome this issue might be to establish a prior probability distribution on the word sense assignments with an independent word sense disambiguation system. Priors could also be inferred from the WordNet sense ordering[7], although this approach could be more noisy. An alternative approach would be to use an unsupervised word-clustering method on a large corpus of parsed output from the same domain; words would be clustered based on the distribution of neighboring words in the dependency tree. The clusters formed with this technique would only reflect those word senses which appear in the domain, thereby eliminating the troublesome outlier senses.

8 Conclusion

*** more conclusion to come later ***

[7]WordNet orders its word sense index according to how frequently each sense is assigned to a word in various semantically-tagged corpora. However, these counts are not always available, so not all word senses will be ordered.

- mention how the lack of noun-noun interactions is making most head-modifier interactions devoid of word-sense “action”

- pp-attachment amb problem

In the future, we may experiment with hypernym features that use depth thresholding, producing features only for hypernyms which are within a certain distance from the top level. A proper threshold could keep the number of features at a manageable level, while retaining the benefits of hypernym features. Our hope is that the reduction of the feature space would lead to more effective training, allowing better performance than the unrestricted hypernym features.

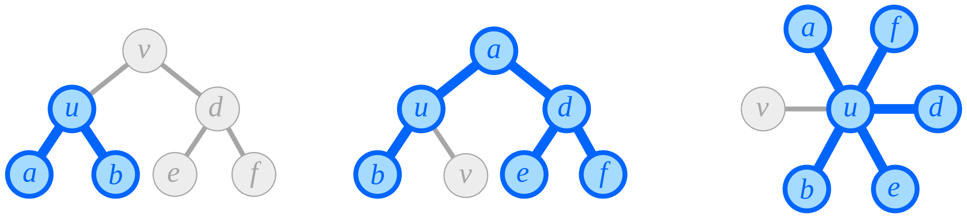

Figure 16: Depictions of the shape of $\mathcal{T}_{u/v}$ in some sample graphs.

Appendix

A Equivalence of Tree Beliefs and Marginal Probabilities

For the purposes of our proof, we define the following notation. Let $\mathcal{T}$ denote the entire tree, and for every edge (u, v), let $\mathcal{T}_{u/v}$ denote the subtree of $\mathcal{T}$ that is rooted at u and that lies "behind" v (i.e. if we were to divide $\mathcal{T}$ into two connected components by cutting edge (u, v), then $\mathcal{T}_{u/v}$ is the component which contains u; see Figure 16 for some visual examples). Note that $\forall v \forall s, t \in N(v)$, $\mathcal{T}_{s/v} \cap \mathcal{T}_{t/v} = \{\}$, or else we could show a cycle in $\mathcal{T}$.

We define the function **a** such that for any set of nodes S, $\mathbf{a}(S)$ enumerates through all word sense assignments to the nodes in S. We define **a** over trees as well; $\mathbf{a}(\mathcal{T})$ and $\mathbf{a}(\mathcal{T}_{u/v})$ enumerate through sense assignments to the nodes of these subtrees. Under this new notation, the marginal probabilies for each node and pair of nodes can now be given by:

$$\begin{aligned} p_u(x_u) &= \frac{1}{Z_{i,j}} \sum_{\mathbf{a}(\mathcal{T})\,|\,s_u=x_u} \prod_{w\in\mathcal{T}} \alpha_w(s_w) \prod_{(u',v')\in\mathcal{T}} \beta_{u',v'}(s_{u'}, s_{v'}) \\ p_{u,v}(x_u, x_v) &= \frac{1}{Z_{i,j}} \sum_{\mathbf{a}(\mathcal{T})\,|\,s_u=x_u, s_v=x_v} \prod_{w\in\mathcal{T}} \alpha_w(s_w) \prod_{(u',v')\in\mathcal{T}} \beta_{u',v'}(s_{u'}, s_{v'}) \end{aligned}$$

We also define partial normalization constants:

$$\begin{aligned} \mathbf{Z}(\mathcal{T}_{u/v}) &= \sum_{\mathbf{a}(\mathcal{T}_{u/v})} \prod_{w\in\mathcal{T}_{u/v}} \alpha_w(s_w) \prod_{(u',v')\in\mathcal{T}_{u/v}} \beta_{u',v'}(s_{u'}, s_{v'}) \\ \mathbf{Z}(\mathcal{T}_{u/v}\,|\,s_u = x_u) &= \sum_{\mathbf{a}(\mathcal{T}_{u/v})\,|\,s_u=x_u} \prod_{w\in\mathcal{T}_{u/v}} \alpha_w(s_w) \prod_{(u',v')\in\mathcal{T}_{u/v}} \beta_{u',v'}(s_{u'}, s_{v'}) \end{aligned}$$

where $\mathbf{Z}(\mathcal{T}_{u/v})$ gives the normalization constant for the subtree $\mathcal{T}_{u/v}$, and $\mathbf{Z}(\mathcal{T}_{u/v}\,|\,s_u = x_u)$ gives the normalization constant for subtree $\mathcal{T}_{u/v}$ when sense s_u is fixed to the particular value x_u.

It is worthwhile to explore some of the properties of these partial normalization constants before we continue. First, note that the normalization constant $\mathbf{Z}(\mathcal{T}_{u/v})$ can be built up out of partially-fixed normalization constants $\mathbf{Z}(\mathcal{T}_{u/v} \mid s_u = x_u)$ as follows:

$$\mathbf{Z}(\mathcal{T}_{u/v}) = \sum_{x_u \in S_u} \mathbf{Z}(\mathcal{T}_{u/v} \mid s_u = x_u)$$

The result above also holds when more than one word sense is being fixed

$$\mathbf{Z}(\mathcal{T}_{u/v}) = \sum_{x_u \in S_u,\, x_w \in S_w} \mathbf{Z}(\mathcal{T}_{u/v} \mid s_u = x_u, s_w = x_w)$$

and in general, when a subset S of the nodes has their word senses fixed, then

$$\mathbf{Z}(\mathcal{T}_{u/v}) = \sum_{\mathbf{a}(S)} \mathbf{Z}(\mathcal{T}_{u/v} \mid \mathbf{a}(S))$$

In addition, the product between the normalization constants for any two disjoint subtrees $\mathcal{T}_{u/v}$ and $\mathcal{T}_{s/t}$ yields a normalization constant for the union of the two subtrees:

$$\begin{aligned}
\mathbf{Z}(\mathcal{T}_{u/v})\mathbf{Z}(\mathcal{T}_{s/t}) &= \left(\sum_{\mathbf{a}(\mathcal{T}_{u/v})} \prod_{w \in \mathcal{T}_{u/v}} \alpha_w(s_w) \prod_{(u',v') \in \mathcal{T}_{u/v}} \beta_{u',v'}(s_{u'}, s_{v'}) \right) \\
&\quad \left(\sum_{\mathbf{a}(\mathcal{T}_{s/t})} \prod_{r \in \mathcal{T}_{s/t}} \alpha_r(s_r) \prod_{(s',t') \in \mathcal{T}_{s/t}} \beta_{s',t'}(s_{s'}, s_{t'}) \right) \\
&= \sum_{\mathbf{a}(\mathcal{T}_{u/v})} \sum_{\mathbf{a}(\mathcal{T}_{s/t})} \left(\begin{array}{c} \prod_{w \in \mathcal{T}_{u/v}} \alpha_w(s_w) \prod_{(u',v') \in \mathcal{T}_{u/v}} \beta_{u',v'}(s_{u'}, s_{v'}) \\ \prod_{r \in \mathcal{T}_{s/t}} \alpha_r(s_r) \prod_{(s',t') \in \mathcal{T}_{s/t}} \beta_{s',t'}(s_{s'}, s_{t'}) \end{array} \right) \\
&= \sum_{\mathbf{a}(\mathcal{T}_{u/v} \cup \mathcal{T}_{s/t})} \prod_{w \in (\mathcal{T}_{u/v} \cup \mathcal{T}_{s/t})} \alpha_w(s_w) \prod_{(u',v') \in (\mathcal{T}_{u/v} \cup \mathcal{T}_{s/t})} \beta_{u',v'}(s_{u'}, s_{v'}) \\
&= \mathbf{Z}\left(\mathcal{T}_{u/v} \cup \mathcal{T}_{s/t} \right)
\end{aligned}$$

We can draw an interesting conclusion from the above. Recall that for any node v, $\forall s, t \in N(v)$, $\mathcal{T}_{s/v} \cap \mathcal{T}_{t/v} = \{\}$; that is, all neighboring subtrees are disjoint. Therefore, we can construct the tree normalization constant $\mathbf{Z}(\mathcal{T}_{u/v})$ by piecing together smaller normalization constants with node and edge weights.

$$\mathbf{Z}(\mathcal{T}_{u/v}) = \sum_{x_u \in S_u} \alpha_u(x_u) \prod_{w \in N(u) \setminus v} \sum_{x_w \in S_w} \beta_{u,w}(x_u, x_w) \mathbf{Z}(\mathcal{T}_{w/u} \mid s_w = x_w)$$

and a natural extension is that for any node u, we can compute the normalization constant for the entire tree by combining the tree normalization constants for all of the subtrees neighboring u:

$$\forall u, \qquad \mathbf{Z}(\mathcal{T}) = \sum_{x_u \in S_u} \alpha_u(x_u) \prod_{w \in N(u)} \sum_{x_w \in S_w} \beta_{u,w}(x_u, x_w)\mathbf{Z}(\mathcal{T}_{w/u} \,|\, s_w = x_w)$$

We now continue with our proof that tree beliefs equal marginal probabilities. For convenience, we reproduce the recursive formula defining the messages:

$$m_{u\to v}(s_v) = \sum_{s_u \in S_u} \alpha_u(s_u)\beta_{u,v}(s_u, s_v) \prod_{w \in N(u)\setminus v} m_{w\to u}(s_u)$$

The key intuition in our proof is to think of the messages $m_{u\to v}$ as dynamic-programming subproblems, where each message $m_{u\to v}$ is related to the normalization factor $\mathbf{Z}(\mathcal{T}_{u/v})$. The exact relation is given by predicate P below:

$$P(m_{u\to v}) \;\equiv\; \left\{ m_{u\to v}(s_v) = \sum_{x_u \in S_u} \beta_{u,v}(x_u, s_v)\mathbf{Z}(\mathcal{T}_{u/v} \,|\, s_u = x_u) \right\}$$

and the following proof shows that P holds for messages produced by the message-passing algorithm.

Proof by Induction

Base Case The messages emanating from the leaves of $\mathcal{T}$ are the base case of the induction. For a leaf ℓ and its parent p, $P(m_{\ell\to p})$ holds trivially. Since the only neighbor of a leaf node is its parent, $N(\ell) \setminus p = \{\}$ and $\mathbf{Z}(\mathcal{T}_{\ell/p} \,|\, s_\ell = x_\ell) = \alpha_\ell(x_\ell)$. Therefore,

$$\begin{aligned} m_{\ell\to p}(s_p) &= \sum_{s_\ell \in S_\ell} \alpha_\ell(s_\ell)\beta_{\ell,p}(s_\ell, s_p) \prod_{v \in N(\ell)\setminus p} m_{v\to \ell}(s_\ell) \\ &= \sum_{s_\ell \in S_\ell} \alpha_\ell(s_\ell)\beta_{\ell,p}(s_\ell, s_p) \\ &= \sum_{x_\ell \in S_\ell} \beta_{\ell,p}(x_\ell, s_p)\mathbf{Z}(\mathcal{T}_{\ell/p} \,|\, s_\ell = x_\ell) \end{aligned}$$

so that $P(m_{\ell\to p})$ holds.

Inductive Case We prove the property $P(m_{u\to v})$, and our inductive assumption is that $\forall w \neq v,\ P(m_{w\to u})$ holds. We argue that this is a fair inductive assumption, since the messages $m_{w\to u},\ w \neq v$ are exactly those messages which would be required to compute $m_{u\to v}$ in the message passing algorithm (as laid out in Section 2.2). We begin by writing out the formula for $m_{u\to v}(s_v)$, substituting in our inductive assumptions:

$$m_{u\to v}(s_v) = \sum_{x_u \in S_u} \alpha_u(x_u)\beta_{u,v}(x_u, s_v) \prod_{w\in N(u)\backslash v} \sum_{x_w \in S_w} \beta_{w,u}(x_w, x_u)\mathbf{Z}(\mathcal{T}_{w/u} \mid s_w = x_w)$$

We define $F(w, x_w) = \beta_{w,u}(x_w, x_u)\mathbf{Z}(\mathcal{T}_{w/u} \mid s_w = x_w)$ and rewrite the above as follows:

$$m_{u\to v}(s_v) = \sum_{x_u \in S_u} \alpha_u(x_u)\beta_{u,v}(x_u, s_v) \prod_{w\in N(u)\backslash v} \sum_{x_w \in S_w} F(w, x_w)$$

Next, we rearrange the product over sums $\prod_{w\in N(u)\backslash v} \sum_{x_w \in S_w} F(w, x_w)$ into a sum over products:

$$\begin{aligned}
&\prod_{w\in N(u)\backslash v} \sum_{x_w \in S_w} F(w, x_w) \\
= &\begin{pmatrix} \left(F(w_1, x^1_{w_1}) + F(w_1, x^2_{w_1}) + \ldots + F(w_1, x^{|S_{w_1}|}_{w_1})\right) \\ \left(F(w_2, x^1_{w_2}) + F(w_2, x^2_{w_2}) + \ldots + F(w_2, x^{|S_{w_2}|}_{w_2})\right) \\ \vdots \\ \left(F(w_M, x^1_{w_1}) + F(w_M, x^2_{w_M}) + \ldots + F(w_M, x^{|S_{w_M}|}_{w_M})\right) \end{pmatrix} \\
= &\begin{pmatrix} \left(F(w_1, x^1_{w_1}) \quad F(w_2, x^1_{w_2}) \quad \ldots \quad F(w_M, x^1_{w_M})\right) + \\ \left(F(w_1, x^1_{w_1}) \quad F(w_2, x^1_{w_2}) \quad \ldots \quad F(w_M, x^2_{w_M})\right) + \\ \vdots \\ \left(F(w_1, x^2_{w_1}) \quad F(w_2, x^1_{w_2}) \quad \ldots \quad F(w_M, x^1_{w_M})\right) + \\ \vdots \\ \left(F(w_1, x^{|S_{w_1}|}_{w_1}) \quad F(w_2, x^{|S_{w_2}|}_{w_2}) \quad \ldots \quad F(w_M, x^{|S_{w_M}|}_{w_M})\right) + \end{pmatrix} \\
= &\sum_{\mathbf{a}(N(u)\backslash v)} \prod_{w\in N(u)\backslash v} F(w, s_w)
\end{aligned}$$

Applying this rearrangement to our original expression allows the following simplifications:

$$\begin{aligned}
m_{u\to v}(s_v) &= \sum_{x_u \in S_u} \alpha_u(x_u)\beta_{u,v}(x_u, s_v) \sum_{\mathbf{a}(N(u)\backslash v)} \prod_{w\in N(u)\backslash v} \beta_{w,u}(s_w, x_u)\mathbf{Z}(\mathcal{T}_{w/u} \mid s_w) \\
&= \sum_{x_u \in S_u} \beta_{u,v}(x_u, s_v) \sum_{\mathbf{a}(N(u)\backslash v)} \mathbf{Z}\left(\bigcup_{w'\in N(u)\backslash v} \mathcal{T}_{w'/u} \,\middle|\, \mathbf{a}(N(u) \setminus v)\right) \\
&\qquad \alpha_u(x_u) \prod_{w\in N(u)\backslash v} \beta_{w,u}(s_w, x_u)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{x_u \in S_u} \beta_{u,v}(x_u, s_v) \sum_{\mathbf{a}(N(u)\setminus v)} \mathbf{Z}(\mathcal{T}_{u/v} \,|\, \mathbf{a}(N(u) \setminus v), s_u = x_u) \\
&= \sum_{x_u \in S_u} \beta_{u,v}(x_u, s_v) \mathbf{Z}(\mathcal{T}_{u/v} \,|\, s_u = x_u)
\end{aligned}
$$

Therefore, $P(m_{u \to v})$ holds. ■

Now, using the definition in proposition P, we can prove that the node beliefs are equal to the marginal probabilities:

$$
\begin{aligned}
b_u(x_u) &= \frac{1}{z_u} \alpha_u(x_u) \prod_{v \in N(u)} m_{v \to u}(s_u) \\
&= \frac{1}{z_u} \alpha_u(x_u) \prod_{v \in N(u)} \sum_{x_v \in S_v} \beta_{v,u}(x_v, x_u) \mathbf{Z}(\mathcal{T}_{v/u} \,|\, s_v = x_v) \\
&= \frac{1}{z_u} \alpha_u(x_u) \sum_{\mathbf{a}(N(u))} \prod_{v \in N(u)} \beta_{v,u}(x_v, x_u) \mathbf{Z}(\mathcal{T}_{v/u} \,|\, s_v = x_v) \\
&= \frac{1}{z_u} \sum_{\mathbf{a}(N(u))} \mathbf{Z}\left(\bigcup_{w \in N(u)} \mathcal{T}_{w/u} \,\middle|\, \mathbf{a}(N(u)), s_u = x_u \right) \alpha_u(x_u) \prod_{v \in N(u)} \beta_{v,u}(x_v, x_u) \\
&= \frac{1}{z_u} \sum_{\mathbf{a}(N(u))} \mathbf{Z}\left(\mathcal{T} \,|\, \mathbf{a}(N(u)), s_u = x_u\right) \\
&= \frac{1}{z_u} \mathbf{Z}\left(\mathcal{T} \,|\, s_u = x_u\right) \qquad\qquad (*) \\
&= \frac{1}{z_u} \sum_{\mathbf{a}(\mathcal{T}) \,|\, s_u = x_u} \prod_{w \in \mathcal{T}} \alpha_w(s_w) \prod_{(u',v') \in \mathcal{T}} \beta_{u',v'}(s_{u'}, s_{v'}) = p_u(x_u)
\end{aligned}
$$

and similar reasoning can be applied to the pairwise beliefs:

$$
\begin{aligned}
b_{u,v}(x_u, x_v) &= \frac{1}{z_{u,v}} \alpha_u(x_u) \alpha_v(x_v) \beta_{u,v}(x_u, x_v) \prod_{s \in N(u) \setminus v} m_{s \to u}(s_u) \prod_{t \in N(v) \setminus u} m_{t \to v}(s_v) \\
&= \frac{1}{z_{u,v}} \beta_{u,v}(x_u, x_v) \left(\alpha_u(x_u) \prod_{s \in N(u) \setminus v} \sum_{x_s \in S_s} \beta_{s,u}(x_s, x_u) \mathbf{Z}(\mathcal{T}_{s/u} \,|\, s_s = x_s) \right) \\
&\qquad \left(\alpha_v(x_v) \prod_{t \in N(v) \setminus u} \sum_{x_t \in S_t} \beta_{t,v}(x_t, x_v) \mathbf{Z}(\mathcal{T}_{t/v} \,|\, s_t = x_t) \right) \\
&= \frac{1}{z_{u,v}} \beta_{u,v}(x_u, x_v) \left(\alpha_u(x_u) \sum_{\mathbf{a}(N(u) \setminus v)} \prod_{s \in N(u) \setminus v} \beta_{s,u}(x_s, x_u) \mathbf{Z}(\mathcal{T}_{s/u} \,|\, s_s = x_s) \right) \\
&\qquad \left(\alpha_v(x_v) \sum_{\mathbf{a}(N(v) \setminus u)} \prod_{t \in N(v) \setminus u} \beta_{t,v}(x_t, x_v) \mathbf{Z}(\mathcal{T}_{t/v} \,|\, s_t = x_t) \right)
\end{aligned}
$$

$$\begin{aligned}
&= \frac{1}{z_{u,v}} \beta_{u,v}(x_u, x_v) \mathbf{Z}(\mathcal{T}_{u/v} \mid s_u = x_u) \mathbf{Z}(\mathcal{T}_{v/u} \mid s_v = x_v) \\
&= \frac{1}{z_{u,v}} \mathbf{Z}(\mathcal{T} \mid s_u = x_u, s_v = x_v) \qquad (*) \\
&= \frac{1}{z_{u,v}} \sum_{\mathbf{a}(\mathcal{T}) \mid s_u = x_u, s_v = x_v} \prod_{w \in \mathcal{T}} \alpha_w(s_w) \prod_{(u',v') \in \mathcal{T}} \beta_{u',v'}(s_{u'}, s_{v'}) = p_{u,v}(x_u, x_v)
\end{aligned}$$

Incidentally, the above also proves that the node and pairwise normalization constants are equal to the tree-wide normalization constant $Z_{i,j}$. Consider the lines marked $(*)$ above, and note that

$$\sum_{x_u \in S_u} b_u(x_u) = 1 \quad \Rightarrow \quad z_u = \sum_{x_u \in S_u} \mathbf{Z}(\mathcal{T} \mid s_u = x_u) = \mathbf{Z}(\mathcal{T}) = Z_{i,j}$$

$$\sum_{x_u \in S_u, x_v \in S_v} b_{u,v}(x_u, x_v) = 1 \quad \Rightarrow \quad z_{u,v} = \sum_{x_u \in S_u, x_v \in S_v} \mathbf{Z}(\mathcal{T} \mid s_u = x_u, s_v = x_v) = \mathbf{Z}(\mathcal{T}) = Z_{i,j}$$

B Computing Messages and Beliefs in Linear Time

We show that given node and pairwise potentials α_u and $\beta_{u,v}$, belief propagation can create messages and beliefs in time linear in the size of the tree. Let n be the number of nodes in the tree, and let the constant $S = \max |S_u|$, bound the size of the word-sense domains. We reproduce below the recursive formula defining the messages:

$$m_{u \to v}(s_v) = \sum_{s_u \in S_u} \alpha_u(s_u) \beta_{u,v}(s_u, s_v) \prod_{w \in N(u) \setminus v} m_{w \to u}(s_u)$$

Recall that we create the messages in each tree by first computing messages from the leaves inward to the root, and then from the root outward to the leaves (see Figure 6 on page 11). We call the first set of messages (leaves to root) *upstream* messages and the second set (root to leaves) *downstream* messages.

Now, suppose we wish to compute the upstream message from node u to its parent p. This message $m_{u \to p}(s_p)$ will have $|S_p| \leq S$ dimensions, and for each dimension we must take a summation over $|S_u| \leq S$ products of the $|N(u)| - 1$ messages which arrive from u's children. After multiplying in the weights α_u and $\beta_{u,p}$ and completing the summations, the entire message will require $O(|N(u)| S^2)$ time.

As we compute this upstream message, however, we save each product of child messages in a "message product" array $mp_u(s_u)$ that is attached to the node. That is, we set

$$mp_u(s_u) = \prod_{c \in N(u) \setminus p} m_{c \to u}(s_u)$$

We will save a lot of time by reusing these computations later on. Of course, these message product arrays use up an additional $O(nS)$ space. However, note that the messages, node beliefs, and pairwise beliefs already require $O(nS)$, $O(nS)$, and $O(nS^2)$ space respectively, so incurring an additional $O(nS)$ space cost is acceptable.

Now, we compute the downstream messages. We begin at the root node r and compute the downstream message to a child node u. Again, there are $|S_u| \leq S$ dimensions and a summation over $|S_r| \leq S$ terms. However, instead of directly computing the product of messages, we reuse the stored values in the message product array, dividing them by the single held-out message:

$$\prod_{c \in N(r) \setminus u} m_{c \to r}(s_r) = \frac{mp_r(s_r)}{m_{u \to r}(s_r)}$$

When we compute the downstream messages from a non-root node u to its child c, we augment u's stored message products with the downstream message received from u's parent p, and then we reuse the message products as before; that is:

$$\begin{aligned} mp_u(s_u) &\leftarrow mp_u(s_u) m_{p \to u}(s_u) \\ \prod_{v \in N(u) \setminus c} m_{v \to u}(s_u) &= \frac{mp_u(s_u)}{m_{c \to u}(s_u)} \end{aligned}$$

Therefore, we can compute each required message product in $O(1)$ time, so that an entire downstream message can be computed in $O(S^2)$ time.

However, each node u has $O(|N(u)|)$ downstream messages to create, and thus taking care of node u's entire quota of downstream messages requires $O(|N(u)|S^2)$ time.

Thus, every node u (except the root) must compute a single upstream message at a cost of $O(|N(u)|S^2)$, and every node u (except the leaves) must compute $O(|N(u)|)$ downstream messages at a cost of $O(S^2)$ each. The total cost of computing all messages, therefore, is bounded by

$$2 \sum_u O(|N(u)|S^2) = O\left(S^2 \sum_u |N(u)|\right)$$

The summation $\sum_u |N(u)|$ counts each edge twice, once for each endpoint; therefore, $\sum_u |N(u)| = O(n)$, and the total time required to compute all messages is $O(nS^2)$.

Note that at the finish of the message passing algorithm, for every node u, the message product array mp_u contains the product of all incoming messages: when we compute the upstream messages, we initialize mp_u as the product of all child messages, and when we compute the downstream messages, we mp_u to include the parent message. We will exploit this property below.

We compute the normalizing constants and beliefs by defining intermediate terms

$$\begin{aligned} B_u(s_u) &= \alpha_u(s_u) mp_u(s_u) \\ B_{u,v}(s_u, s_v) &= \alpha_u(s_u)\alpha_v(s_v)\beta_{u,v}(s_u, s_v)\left(\frac{mp_u(s_u)}{m_{v\to u}(s_u)}\right)\left(\frac{mp_v(s_v)}{m_{u\to v}(s_v)}\right) \end{aligned}$$

and using them to create the normalization constants and beliefs:

$$\begin{aligned} z_u &= \sum_{s_u \in S_u} B_u(s_u) \\ z_{u,v} &= \sum_{s_u \in S_u,\, s_v \in S_v} B_{u,v}(s_u, s_v) \\ b_u(s_u) &= \frac{1}{z_u} B_u(s_u) \\ b_{u,v}(s_u, s_v) &= \frac{1}{z_{u,v}} B_{u,v}(s_u, s_v) \end{aligned}$$

Computing each B_u requires $O(S)$ time and computing each $B_{u,v}$ requires $O(S^2)$ time; preparing all of the intermediate terms requires $O(nS + nS^2) = O(nS^2)$ time. Next, each z_u requires $O(S)$ time and each $z_{u,v}$ requires $O(S^2)$ time; all of them together require another $O(S^2)$ time. Finally, the node and pairwise beliefs require only $O(1)$ additional computation per belief, for a total of $O(n)$. Therefore, after the creation of the messages, normalization constants and beliefs can be computed with an additional $O(nS^2)$ time.

In conclusion, given node and pairwise weights α_u and $\beta_{u,v}$, the belief propagation algorithm can create node and pairwise beliefs b_u and $b_{u,v}$, together with the associated normalization constants z_u and $z_{u,v}$, in $O(nS^2)$ time.

References

Eugene Charniak. Statistical Parsing with a Context-Free Grammar and Word Statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 1997.

Michael Collins. Head-Driven Statistical Models for Natural Language Parsing. Doctoral Dissertation, Dept. of Computer Science, Brown University, Providence, RI, 1999.

Michael Collins. Discriminative Reranking for Natural Language Parsing. In *Proceedings of 17th International Conference on Machine Learning*, 2000.

Michael Collins and James Brooks. Prepositional Phrase Attachment through a Backed-off Model. In *Proceedings of the Third Workshop on Very Large Corpora*, 1995.

Mark Johnson and Massimiliano Ciaramita. Supersense Tagging of Unknown Nouns in WordNet. In *EMNLP 2003*, 2003.

George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. Five Papers on WordNet. Technical report, Stanford University, 1993.

A. Ratnaparkhi and S. Roukos. A Maximum Entropy Model for Prepositional Phrase Attachment. In *Proceedings of the ARPA Workshop on Human Language Technology*, 1994.

Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Understanding Belief Propagation and its Generalizations. Technical Report TR2001-22, Mitsubishi Electric Research Labs, 2002.