Learning with Probabilistic Features for Improved Pipeline Models

Razvan C. Bunescu
School of EECS
Ohio University
Athens, OH 45701
bunescu@ohio.edu

Abstract

We present a novel learning framework for pipeline models aimed at improving the communication between consecutive stages in a pipeline. Our method exploits the confidence scores associated with outputs at any given stage in a pipeline in order to compute probabilistic features used at other stages downstream. We describe a simple method of integrating probabilistic features into the linear scoring functions used by state of the art machine learning algorithms. Experimental evaluation on dependency parsing and named entity recognition demonstrate the superiority of our approach over the baseline pipeline models, especially when upstream stages in the pipeline exhibit low accuracy.

1 Introduction

Machine learning algorithms are used extensively in natural language processing. Applications range from fundamental language tasks such as part of speech (POS) tagging or syntactic parsing, to higher level applications such as information extraction (IE), semantic role labeling (SRL), or question answering (QA). Learning a model for a particular language processing problem often requires the output from other natural language tasks. Syntactic parsing and dependency parsing usually start with a textual input that is tokenized, split in sentences and POS tagged. In information extraction, named entity recognition (NER), coreference resolution, and relation extraction (RE) have been shown to benefit from features that use POS tags and syntactic dependencies. Similarly, most SRL approaches assume a parse tree representation of the input sentences. The common practice in modeling such dependencies is to use a pipeline organization, in which the output of one task is fed as input to the next task in the sequence. One advantage of this model is that it is very simple to implement; it also allows for a modular approach to natural language processing. The key disadvantage is that errors propagate between stages in the pipeline, significantly affecting the quality of the final results. One solution is to solve the tasks jointly, using the principled framework of probabilistic graphical models. Sutton et al. (2004) use factorial Conditional Random Fields (CRFs) (Lafferty et al., 2001) to jointly predict POS tags and segment noun phrases, improving on the cascaded models that perform the two tasks in sequence. Wellner et al. (2004) describe a CRF model that integrates the tasks of citation segmentation and citation matching. Their empirical results show the superiority of the integrated model over the pipeline approach. While more accurate than their pipeline analogues, probabilistic graphical models that jointly solve multiple natural language tasks are generally more demanding in terms of finding the right representations, the associated inference algorithms and their computational complexity. Recent negative results on the integration of syntactic parsing with SRL (Sutton and McCallum, 2005) provide additional evidence for the difficulty of this general approach. When dependencies between the tasks can be formulated in terms of constraints between their outputs, a simpler approach is to solve the tasks separately and integrate the constraints in a linear programming formulation, as proposed by Roth and

Yih (2004) for the simultaneous learning of named entities and relations between them. More recently, Finkel et al. (2006) model the linguistic pipelines as Bayesian networks on which they perform Monte Carlo inference in order to find the most likely output for the final stage in the pipeline.

In this paper, we present a new learning method for pipeline models that mitigates the problem of error propagation between the tasks. Our method exploits the probabilities output by any given stage in the pipeline as weights for the features used at other stages downstream. We show a simple method of integrating probabilistic features into linear scoring functions, which makes our approach applicable to state of the art machine learning algorithms such as CRFs and Support Vector Machines (Vapnik, 1998; Schölkopf and Smola, 2002). Experimental results on dependency parsing and named entity recognition show useful improvements over the baseline pipeline models, especially when the basic pipeline components exhibit low accuracy.

2 Learning with Probabilistic Features

We consider that the task is to learn a mapping from inputs $x \in \mathcal{X}$ to outputs $y \in \mathcal{Y}(x)$. Each input x is also associated with a different set of outputs $z \in \mathcal{Z}(x)$ for which we are given a probabilistic confidence measure $p(z|x)$. In a pipeline model, z would correspond to the annotations performed on the input x by all stages in the pipeline other than the stage that produces y. For example, in the case of dependency parsing, x is a sequence of words, y is a set of word-word dependencies, z is a sequence of POS tags, and $p(z|x)$ is a measure of the confidence that the POS tagger has in the output z. Let ϕ be a representation function that maps an example (x, y, z) to a feature vector $\phi(x, y, z) \in \mathbb{R}^d$, and $w \in \mathbb{R}^d$ a parameter vector. Equations (1) and (2) below show the traditional method for computing the optimal output $\hat{y}$ in a pipeline model, assuming a linear scoring function defined by w and ϕ.

$$\hat{y}(x) = \operatorname*{argmax}_{y \in \mathcal{Y}(x)} w \cdot \phi(x, y, \hat{z}(x)) \quad (1)$$

$$\hat{z}(x) = \operatorname*{argmax}_{z \in \mathcal{Z}(x)} p(z|x) \quad (2)$$

The weight vector w is learned by optimizing a predefined objective function on a training dataset.

In the model above, only the best annotation $\hat{z}$ produced by upstream stages is used for determining the optimal output $\hat{y}$. However, $\hat{z}$ may be an incorrect annotation, while the correct annotation may be ignored because it was assigned a lower confidence value. We propose exploiting all possible annotations and their probabilities as illustrated in the new model below:

$$\hat{y}(x) = \operatorname*{argmax}_{y \in \mathcal{Y}(x)} w \cdot \psi(x, y) \quad (3)$$

$$\psi(x, y) = \sum_{z \in \mathcal{Z}(x)} p(z|x) \cdot \phi(x, y, z) \quad (4)$$

In most cases, directly computing $\psi(x, y)$ is unfeasible, due to a large number of annotations in $\mathcal{Z}(x)$. In our dependency parsing example, $\mathcal{Z}(x)$ contains all possible POS taggings of sentence x; consequently its cardinality is exponential in the length of the sentence. A more efficient way of computing $\psi(x, y)$ can be designed based on the observation that most components ϕ_i of the original feature vector ϕ utilize only a limited amount of evidence from the example (x, y, z). We define $(\tilde{x}, \tilde{y}, \tilde{z}) \in \mathcal{F}_i(x, y, z)$ to capture the actual evidence from (x, y, z) that is used by one instance of feature function ϕ_i. We call $(\tilde{x}, \tilde{y}, \tilde{z})$ a *feature instance* of ϕ_i in the example (x, y, z). Correspondingly, $\mathcal{F}_i(x, y, z)$ is the set of all feature instances of ϕ_i in example (x, y, z). Usually, $\phi_i(x, y, z)$ is set to be equal with the number of instances of ϕ_i in example (x, y, z), i.e. $\phi_i(x, y, z) = |\mathcal{F}_i(x, y, z)|$. Table 1 illustrates three feature instances $(\tilde{x}, \tilde{y}, \tilde{z})$ generated by three typical dependency parsing features in the example from Figure 1. Because the same feature may be instantiated multiple times in the same example, the components of each feature instance are annotated with their positions relative to the example. Given these definitions, the feature vector $\psi(x, y)$ from (4) can be

	ϕ_1 : **DT → NN**	ϕ_2 : **NNS → thought**	ϕ_3 : **be ← in**
$\tilde{y}$	$10 \rightarrow 11$	$2 \rightarrow 4$	$7 \leftarrow 9$
$\tilde{z}$	DT_{10} NN_{11}	NNS_2	
$\tilde{x}$		thought_4	be_7 in_9
$\lvert\mathcal{F}_i\rvert$	$O(\lvert x\rvert^2)$	$O(\lvert x\rvert)$	$O(1)$

Table 1: Feature instances.

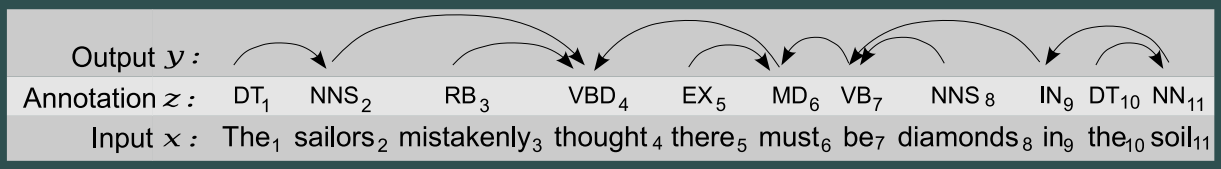

Figure 1: Dependency Parsing Example.

rewritten in a component-wise manner as follows:

$$\begin{aligned}
\psi(x,y) &= [\psi_1(x,y) \ldots \psi_d(x,y)] \quad (5)\\
\psi_i(x,y) &= \sum_{z\in\mathcal{Z}(x)} p(z|x)\cdot\phi_i(x,y,z)\\
&= \sum_{z\in\mathcal{Z}(x)} p(z|x)\cdot|\mathcal{F}_i(x,y,z)|\\
&= \sum_{z\in\mathcal{Z}(x)} p(z|x) \sum_{(\tilde{x},\tilde{y},\tilde{z})\in\mathcal{F}_i(x,y,z)} 1\\
&= \sum_{z\in\mathcal{Z}(x)} \sum_{(\tilde{x},\tilde{y},\tilde{z})\in\mathcal{F}_i(x,y,z)} p(z|x)\\
&= \sum_{(\tilde{x},\tilde{y},\tilde{z})\in\mathcal{F}_i(x,y,\mathcal{Z}(x))} \sum_{z\in\mathcal{Z}(x),z\supseteq\tilde{z}} p(z|x)
\end{aligned}$$

where $\mathcal{F}_i(x,y,\mathcal{Z}(x))$ stands for:

$$\mathcal{F}_i(x,y,\mathcal{Z}(x)) = \bigcup_{z\in\mathcal{Z}(x)} \mathcal{F}_i(x,y,z)$$

We introduce $p(\tilde{z}|x)$ to denote the expectation:

$$p(\tilde{z}|x) = \sum_{z\in\mathcal{Z}(x),z\supseteq\tilde{z}} p(z|x)$$

Then $\psi_i(x,y)$ can be written compactly as:

$$\psi_i(x,y) = \sum_{(\tilde{x},\tilde{y},\tilde{z})\in\mathcal{F}_i(x,y,\mathcal{Z}(x))} p(\tilde{z}|x) \quad (6)$$

The total number of terms in (6) is equal with the number of instantiations of feature ϕ_i in the example (x,y) across all possible annotations $z\in\mathcal{Z}(x)$, i.e. $|\mathcal{F}_i(x,y,\mathcal{Z}(x))|$. Usually this is significantly smaller than the exponential number of terms in (4). The actual number of terms depends on the particular feature used to generate them, as illustrated in the last row of Table 1 for the three features used in dependency parsing. The overall time complexity for calculating $\psi(x,y)$ also depends on the time complexity needed to compute the expectations $p(\tilde{z}|x)$. When z is a sequence, $p(\tilde{z}|x)$ can be computed efficiently using a constrained version of the forward-backward algorithm (to be described in Section 3). When z is a tree, $p(\tilde{z}|x)$ will be computed using a constrained version of the CYK algorithm (to be described in Section 4).

The time complexity can be further reduced if instead of $\psi(x,y)$ we use its subcomponent $\hat{\psi}(x,y)$ that is calculated based only on instances that appear in the optimal annotation $\hat{z}$:

$$\begin{aligned}
\hat{\psi}(x,y) &= [\hat{\psi}_1(x,y) \ldots \hat{\psi}_d(x,y)] \quad (7)\\
\hat{\psi}_i(x,y) &= \sum_{(\tilde{x},\tilde{y},\tilde{z})\in\mathcal{F}_i(x,y,\hat{z})} p(\tilde{z}|x) \quad (8)
\end{aligned}$$

The three models are summarized in Table 2 below. In the next two sections we illustrate their application to two common tasks in language processing: dependency parsing and named entity recognition.

M_1	$\hat{y}(x)$	$=$	$\operatorname{argmax}_{y\in\mathcal{Y}(x)} w\cdot\phi(x,y)$
	$\phi(x,y)$	$=$	$\phi(x,y,\hat{z}(x))$
	$\hat{z}(x)$	$=$	$\operatorname{argmax}_{z\in\mathcal{Z}(x)} p(z\|x)$
M_2	$\hat{y}(x)$	$=$	$\operatorname{argmax}_{y\in\mathcal{Y}(x)} w\cdot\psi(x,y)$
	$\psi(x,y)$	$=$	$[\psi_1(x,y) \ldots \psi_d(x,y)]$
	$\psi_i(x,y)$	$=$	$\sum_{(\tilde{x},\tilde{y},\tilde{z})\in\mathcal{F}_i(x,y,\mathcal{Z}(x))} p(\tilde{z}\|x)$
M_3	$\hat{y}(x)$	$=$	$\operatorname{argmax}_{y\in\mathcal{Y}(x)} w\cdot\hat{\psi}(x,y)$
	$\hat{\psi}(x,y)$	$=$	$[\hat{\psi}_1(x,y) \ldots \hat{\psi}_d(x,y)]$
	$\hat{\psi}_i(x,y)$	$=$	$\sum_{(\tilde{x},\tilde{y},\tilde{z})\in\mathcal{F}_i(x,y,\hat{z})} p(\tilde{z}\|x)$

Table 2: Three Pipeline Models.

3 Dependency Parsing Pipeline

In a traditional dependency parsing pipeline (model M_1 in Table 2), an input sentence x is first aug-

mented with a POS tagging $\hat{z}(x)$, and then processed by a dependency parser in order to obtain a dependency structure $\hat{y}(x)$. To evaluate the new pipeline models we use MSTPARSER[1], a linearly scored dependency parser developed by McDonald et al. (2005). Following the edge based factorization method of Eisner (1996), the score of a dependency tree in the first order version is defined as the sum of the scores of all edges in the tree. Equivalently, the feature vector of a dependency tree is defined as the sum of the feature vectors of all edges in the tree:

$$M_1\text{: } \phi(x,y) = \sum_{u \to v \in y} \phi(x, u \to v, \hat{z}(x))$$

$$M_2\text{: } \psi(x,y) = \sum_{u \to v \in y} \psi(x, u \to v)$$

$$M_3\text{: } \hat{\psi}(x,y) = \sum_{u \to v \in y} \hat{\psi}(x, u \to v)$$

For each edge $u \to v \in y$, MSTPARSER generates features based on a set of feature templates that take into account the words and POS tags at positions u, v, and their left and right neighbors $u \pm 1$, $v \pm 1$. For example, a particular feature template $\mathcal{T}$ used inside MSTPARSER generates the following POS bigram features:

$$\phi_i(x, u \to v, z) = \begin{cases} 1, & \text{if } \langle z_u, z_v \rangle = \langle t_1, t_2 \rangle \\ 0, & \text{otherwise} \end{cases}$$

where $t_1, t_2 \in \mathcal{P}$ are the two POS tags associated with feature index i. By replacing y with $u \to v$ in the feature expressions from Table 2, we obtain the following formulations:

$$M_1\text{: } \phi_i(x, u \to v) = \begin{cases} 1, & \text{if } \langle \hat{z}_u, \hat{z}_v \rangle = \langle t_1, t_2 \rangle \\ 0, & \text{otherwise} \end{cases}$$

$$M_2\text{: } \psi_i(x, u \to v) = p(\tilde{z} = \langle t_1, t_2 \rangle | x)$$

$$M_3\text{: } \hat{\psi}_i(x, u \to v) = \begin{cases} p(\tilde{z} = \langle t_1, t_2 \rangle | x), & \text{if } \langle \hat{z}_u, \hat{z}_v \rangle = \langle t_1, t_2 \rangle \\ 0, & \text{otherwise} \end{cases}$$

where, following the notation from Section 2, $\tilde{z} = \langle z_u, z_v \rangle$ is the actual evidence from z that is used by feature i, and $\hat{z}$ is the top scoring annotation produced by the POS tagger. The implementation in MSTPARSER corresponds to the traditional pipeline model M_1. Given a method for computing feature probabilities $p(\tilde{z} = \langle t_1, t_2 \rangle | x)$, it is straightforward to modify MSTPARSER to implement models M_2 and M_3 – we simply replace the feature vectors ϕ with ψ and $\hat{\psi}$ respectively. As mentioned in Section 2, the time complexity of computing the feature vectors ψ in model M_2 depends on the complexity of the actual evidence $\tilde{z}$ used by the features. For example, the feature template $\mathcal{T}$ used above is based on the POS tags at both ends of a dependency edge, consequently it would generate $|\mathcal{P}|^2$ features in model M_2 for any given edge $u \to v$. There are however feature templates used in MSTPARSER that are based on the POS tags of up to 4 tokens in the input sentence, which means that for each edge they would generate $|\mathcal{P}|^4 \approx 4.5M$ features. Whether using all these probabilistic features is computationally feasible or not also depends on the time complexity of computing the confidence measure $p(\tilde{z}|x)$ associated with each feature.

3.1 Probabilistic POS features

The new pipeline models M_2 and M_3 require an annotation model that, at a minimum, facilitates the computation of probabilistic confidence values for each output. We chose to use linear chain CRFs (Lafferty et al., 2001) since CRFs can be easily modified to compute expectations of the type $p(\tilde{z}|x)$, as needed by M_2 and M_3.

The CRF tagger was implemented in MALLET (McCallum, 2002) using the original feature templates from (Ratnaparkhi, 1996). The model was trained on sections 2–21 from the English Penn Treebank (Marcus et al., 1993). When tested on section 23, the CRF tagger obtains 96.25% accuracy, which is competitive with more finely tuned systems such as Ratnaparkhi's MaxEnt tagger.

We have also implemented in MALLET a constrained version of the forward-backward procedure that allows computing feature probabilities $p(\tilde{z}|x)$. If $\tilde{z} = \langle t_{i_1} t_{i_2} ... t_{i_k} \rangle$ specifies the tags at k positions in the sentence, then the procedure recomputes the α parameters for all positions between i_1 and i_k by constraining the state transitions to pass through the specified tags at the k positions. A similar approach was used by Culotta et al. in (2004) in order to associate confidence values with sequences of contiguous tokens identified by a CRF model as fields in an information extraction task. The constrained proce-

[1] URL: http://sourceforge.net/projects/mstparser

dure requires $(i_k - i_1)|\mathcal{P}|^2 = O(N|\mathcal{P}|^2)$ multiplications in an order 1 Markov model, where N is the length of the sentence. Because MSTPARSER uses an edge based factorization of the scoring function, the constrained forward procedure will need to be run for each feature template, for each pair of tokens in the input sentence x. If the evidence $\tilde{z}$ required by the feature template $\mathcal{T}$ constrains the tags at k positions, then the total time complexity for computing the probabilistic features $p(\tilde{z}|x)$ generated by $\mathcal{T}$ is:

$$O(N^3|\mathcal{P}|^{k+2}) = O(N|\mathcal{P}|^2) \cdot O(N^2) \cdot O(|\mathcal{P}|^k) \quad (9)$$

As mentioned earlier, some feature templates used in the dependency parser constrain the POS tags at 4 positions, leading to a $O(N^3|\mathcal{P}|^6)$ time complexity for a length N sentence. Experimental runs on the same machine that was used for CRF training show that such a time complexity is not yet feasible, especially because of the large size of $\mathcal{P}$ (46 POS tags). In order to speed up the computation of probabilistic features, we made the following two approximations:

1. Instead of using the constrained forward-backward procedure, we enforce an independence assumption between tags at different positions and rewrite $p(\tilde{z} = \langle t_{i_1} t_{i_2} ... t_{i_k} \rangle | x)$ as:

$$p(t_{i_1} t_{i_2} ... t_{i_k} | x) \approx \prod_{j=1}^{k} p(t_{i_j} | x)$$

   The marginal probabilities $p(t_{i_j}|x)$ are easily computed using the original forward and backward parameters as:

$$p(t_{i_j}|x) = \frac{\alpha_{i_j}(t_{i_j}|x)\beta_{i_j}(t_{i_j}|x)}{Z(x)}$$

   This approximation eliminates the factor $O(N|\mathcal{P}|^2)$ from the time complexity in (9).

2. If any of the marginal probabilities $p(t_{i_j}|x)$ is less than a predefined threshold $(\tau|\mathcal{P}|)^{-1}$, we set $p(\tilde{z}|x)$ to 0. When $\tau \geq 1$, the method is guaranteed to consider at least the most probable state when computing the probabilistic features. Looking back at Equation (4), this is equivalent with summing feature vectors only over the most probable annotations $z \in \mathcal{Z}(x)$.

   The approximation effectively replaces the factor $O(|\mathcal{P}|^k)$ in (9) with a quasi-constant factor.

The two approximations lead to an overall time complexity of $O(N^2)$ for computing the probabilistic features associated with any feature template $\mathcal{T}$, plus $O(N|\mathcal{P}|^2)$ for the unconstrained forward-backward procedure. We will use M_2' to refer to the model M_2 that incorporates the two approximations. The independence assumption from the first approximation can be relaxed without increasing the asymptotic time complexity by considering as independent only chunks of contiguous POS tags that are at least a certain number of tokens apart. Consequently, the probability of the tag sequence will be approximated with the product of the probabilities of the tag chunks, where the exact probability of each chunk is computed in constant time with the constrained forward-backward procedure. We will use M_2'' to refer to the resulting model.

3.2 Experimental Results

MSTPARSER was trained on sections 2–21 from the WSJ Penn Treebank, using the gold standard POS tagging. The parser was then evaluated on section 23, using the POS tagging output by the CRF tagger. For model M_1 we need only the best output from the POS tagger. For models M_2' and M_2'' we compute the probability associated with each feature using the corresponding approximations, as described in the previous section. In model M_2'' we consider as independent only chunks of POS tags that are 4 tokens or more apart. If the distance between the chunks is less than 4 tokens, the probability for the entire tag sequence in the feature is computed exactly using the constrained forward-backward procedure. Table 3 shows the accuracy obtained by models M_1, $M_2'(\tau)$ and $M_2''(\tau)$ for various values of the threshold parameter τ. The accuracy is computed over unlabeled dependencies i.e. the percentage of words for which the parser has correctly identified the parent in the dependency tree. The pipeline

M_1	$M_2'(1)$	$M_2'(2)$	$M_2'(4)$	$M_2''(4)$
88.51	88.66	88.67	88.67	**88.70**

Table 3: Dependency parsing results.

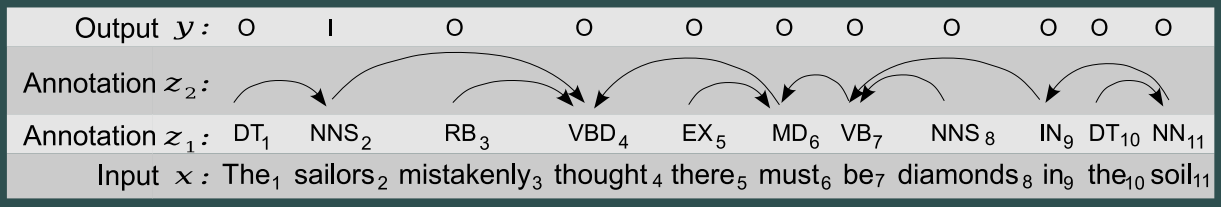

Figure 2: Named Entity Recognition Example.

model M_2' that uses probabilistic features outperforms the traditional pipeline model M_1. As expected, M_2'' performs slightly better than M_2', due to a more exact computation of feature probabilities. Overall, only by using the probabilities associated with the POS features, we achieve an absolute error reduction of 0.19%, in a context where the POS stage in the pipeline already has a very high accuracy of 96.25%. We expect probabilistic features to yield a more substantial improvement in cases where the pipeline model contains less accurate upstream stages. Such a case is that of NER based on a combination of POS and dependency parsing features.

4 Named Entity Recognition Pipeline

In Named Entity Recognition (NER), the task is to identify textual mentions of predefined types of entities. Traditionally, NER is modeled as a sequence classification problem: each token in the input sentence is tagged as being either inside (I) or outside (O) of an entity mention. Most sequence tagging approaches use the words and the POS tags in a limited neighborhood of the current sentence position in order to compute the corresponding features. We augment these *flat features* with a set of *tree features* that are computed based on the words and POS tags found in the proximity of the current token in the dependency tree of the sentence. We argue that such dependency tree features are better at capturing predicate-argument relationships, especially when they span long stretches of text. Figure 2 shows a sentence x together with its POS tagging z_1, dependency links z_2, and an output tagging y. Assuming the task is to recognize mentions of people, the word *sailors* needs to be tagged as inside. If we extracted only flat features using a symmetric window of size 3, the relationship between *sailors* and *thought* would be missed. This relationship is useful, since an agent of the predicate *thought* is likely to be a person entity. On the other hand, the nodes *sailors* and *thought* are adjacent in the dependency tree of the sentence. Therefore, their relationship can be easily captured as a dependency tree feature using the same window size.

For every token position, we generate flat features by considering all unigrams, bigrams and trigrams that start with the current token and extend either to the left or to the right. Similarly, we generate tree features by considering all unigrams, bigrams and trigrams that start with the current token and extend in any direction in the undirected version of the dependency tree. The tree features are also augmented with the actual direction of the dependency arcs between the tokens. If we use only words to create n-gram features, the token *sailors* will be associated with the following features:

- **Flat:** *sailors*, *the sailors*, ⟨*S*⟩ *the sailors*, *sailors mistakenly*, *sailors mistakenly thought*.
- **Tree:** *sailors*, *sailors* ← *the*, *sailors* → *thought*, *sailors* → *thought* ← *must*, *sailors* → *thought* ← *mistakenly*.

We also allow n-grams to use word classes such as POS tags and any of the following five categories: ⟨1C⟩ for tokens consisting of one capital letter, ⟨AC⟩ for tokens containing only capital letters, ⟨FC⟩ for tokens that start with a capital letter, followed by small letters, ⟨CD⟩ for tokens containing at least one digit, and ⟨CRT⟩ for the current token.

The set of features can then be defined as a Cartesian product over word classes, as illustrated in Figure 3 for the original tree feature *sailors* → *thought* ← *mistakenly*. In this case, instead of one completely lexicalized feature, the model will consider 12 different features such as *sailors* → *VBD* ← *RB*, *NNS* → *thought* ← *RB*, or *NNS* → *VBD* ← *RB*.

$$\begin{bmatrix} \langle \text{CRT} \rangle \\ \text{NNS} \\ \text{sailors} \end{bmatrix} \times [\rightarrow] \times \begin{bmatrix} \text{VBD} \\ \text{thought} \end{bmatrix} \times [\leftarrow] \times \begin{bmatrix} \text{RB} \\ \text{mistakenly} \end{bmatrix}$$

Figure 3: Dependency tree features.

The pipeline model M_2 uses features that appear in all possible annotations $z = \langle z_1, z_2 \rangle$, where z_1 and z_2 are the POS tagging and the dependency parse respectively. If the corresponding evidence is $\tilde{z} = \langle \tilde{z}_1, \tilde{z}_2 \rangle$, then:

$$p(\tilde{z}|x) = p(\tilde{z}_2|\tilde{z}_1, x)p(\tilde{z}_1|x)$$

For example, $NNS_2 \rightarrow thought_4 \leftarrow RB_3$ is a feature instance for the token *sailors* in the annotations from Figure 2. This can be construed as having been generated by a feature template $\mathcal{T}$ that outputs the POS tag t_i at the current position, the word x_j that is the parent of x_i in the dependency tree, and the POS tag t_k of another dependent of x_j (i.e. $t_i \rightarrow x_j \leftarrow t_k$). The probability $p(\tilde{z}|x)$ for this type of features can then be written as:

$$p(\tilde{z}|x) = p(i \rightarrow j \leftarrow k|t_i, t_k, x) \cdot p(t_i, t_k|x)$$

The two probability factors can be computed exactly as follows:

1. The M_2 model for dependency parsing from Section 3 is used to compute the probabilistic features $\psi(x, u \rightarrow v|t_i, t_k)$ by constraining the POS annotations to pass through tags t_i and t_k at positions i and k. The total time complexity for this step is $O(N^3|\mathcal{P}|^{k+2})$.

2. Having access to $\psi(x, u \rightarrow v|t_i, t_k)$, the factor $p(i \rightarrow j \leftarrow k|t_i, t_k, x)$ can be computed in $O(N^3)$ time using a constrained version of Eisner's algorithm, as will be explained in Section 4.1.

3. As described in Section 3.1, computing the expectation $p(t_i, t_k|x)$ takes $O(N|\mathcal{P}^2|)$ time using the constrained forward-backward algorithm.

The current token position i can have a total of N values, while j and k can be any positions other than i. Also, t_i and t_k can be any POS tag from $\mathcal{P}$. Consequently, the feature template $\mathcal{T}$ induces $O(N^3|\mathcal{P}|^2)$ feature instances. Overall, the time complexity for computing the feature instances generated by $\mathcal{T}$ is $O(N^6|\mathcal{P}|^{k+4})$, as results from:

$$O(N^3|\mathcal{P}|^2) \cdot (O(N^3|\mathcal{P}|^{k+2}) + O(N^3) + O(N|\mathcal{P}|^2))$$

While still polynomial, this time complexity is feasible only for small values of N. In general, the time complexity for computing probabilistic features in the full model M_2 increases with both the number of stages in the pipeline and the complexity of the features.

Motivated by efficiency, we decided to use the pipeline model M_3 in which probabilities are computed only over features that appear in the top scoring annotation $\hat{z} = \langle \hat{z}_1, \hat{z}_2 \rangle$, where $\hat{z}_1$ and $\hat{z}_2$ represent the best POS tagging, and the best dependency parse respectively. In order to further speed up the computation of probabilistic features, we made the following approximations:

1. We consider the POS tagging and the dependency parse independent and rewrite $p(\tilde{z}|x)$ as:

$$p(\tilde{z}|x) = p(\tilde{z}_1, \tilde{z}_2|x) \approx p(\tilde{z}_1|x)p(\tilde{z}_2|x)$$

2. We enforce an independence assumption between POS tags. Thus, if $\tilde{z}_1 = \langle t_{i_1} t_{i_2} ... t_{i_k} \rangle$ specifies the tags at k positions in the sentence, then $p(\tilde{z}_1|x)$ is rewritten as:

$$p(t_{i_1} t_{i_2} ... t_{i_k}|x) \approx \prod_{j=1}^{k} p(t_{i_j}|x)$$

3. We also enforce a similar independence assumption between dependency links. Thus, if $\tilde{z}_2 = \langle u_1 \rightarrow v_1 ... u_k \rightarrow v_k \rangle$ specifies k dependency links, then $p(\tilde{z}_2|x)$ is rewritten as:

$$p(u_1 \rightarrow v_1 ... u_k \rightarrow v_k|x) \approx \prod_{l=1}^{k} p(u_l \rightarrow v_l|x)$$

For example, the probability $p(\tilde{z}|x)$ of the feature instance $NNS_2 \rightarrow thought_4 \leftarrow RB_3$ is approximated as:

$$\begin{aligned} p(\tilde{z}|x) &\approx p(\tilde{z}_1|x) \cdot p(\tilde{z}_2|x) \\ p(\tilde{z}_1|x) &\approx p(t_2 = NNS|x) \cdot p(t_3 = RB|x) \\ p(\tilde{z}_2|x) &\approx p(2 \rightarrow 4|x) \cdot p(3 \rightarrow 4|x) \end{aligned}$$

We will use M_3' to refer to the resulting model.

4.1 Probabilistic Dependency Features

The probabilistic POS features $p(t_i|x)$ are computed using the forward-backward procedure in CRFs, as described in Section 3.1. To completely specify the pipeline model for NER, we also need an efficient method for computing the probabilistic dependency features $p(u \rightarrow v|x)$, where $u \rightarrow v$ is a dependency edge between positions u and v in the sentence x. MSTPARSER is a large-margin method that computes an unbounded score $s(x, y)$ for any given sentence x and dependency structure $y \in \mathcal{Y}(x)$ using the following edge-based factorization:

$$s(x, y) = \sum_{u \rightarrow v \in y} s(x, u \rightarrow v) = w \sum_{u \rightarrow v \in y} \phi(x, u \rightarrow v)$$

The following three steps describe a general method for associating probabilities with output substructures. The method can be applied whenever a structured output is associated a score value that is unbounded in $\mathbb{R}$, assuming that the score of the entire output structure can be computed efficiently based on a factorization into smaller substructures.

S1. Map the unbounded score $s(x, y)$ from $\mathbb{R}$ into $[0, 1]$ using the *softmax* function (Bishop, 1995):

$$n(x, y) = \frac{e^{s(x,y)}}{\sum_{y \in \mathcal{Y}(x)} e^{s(x,y)}}$$

The normalized score $n(x, y)$ preserves the ranking given by the original score $s(x, y)$. The normalization constant at the denominator can be computed in $O(N^3)$ time by replacing the *max* operator with the *sum* operator inside Eisner's chart parsing algorithm.

S2. Compute a normalized score for the substructure by summing up the normalized scores of all the complete structures that contain it. In our model, dependency edges are substructures, while dependency trees are complete structures. The normalized score will then be computed as:

$$n(x, u \rightarrow v) = \sum_{y \in \mathcal{Y}(x), u \rightarrow v \in y} n(x, y)$$

The sum can be computed in $O(N^3)$ time using a constrained version of the algorithm that computes the normalization constant in step S1. This constrained version of Eisner's algorithm works in a similar manner with the constrained forward backward algorithm by restricting the dependency structures to contain a predefined edge or set of edges.

S3. Use the isotonic regression method of Zadrozny and Elkan (2002) to map the normalized scores $n(x, u \rightarrow v)$ into probabilities $p(u \rightarrow v|x)$. A potential problem with the softmax function is that, depending on the distribution of scores, the exponential transform could dramatically overinflate the higher scores. Isotonic regression, by redistributing the normalized scores inside $[0, 1]$, can alleviate this problem.

4.2 Experimental Results

We test the pipeline model M_3' versus the traditional model M_1 on the task of detecting mentions of person entities in the ACE dataset[2]. We use the standard training – testing split of the ACE 2002 dataset in which the training dataset is also augmented with the documents from the ACE 2003 dataset. The combined dataset contains 674 documents for training and 97 for testing. We implemented the CRF model in MALLET using three different sets of features: **Tree**, **Flat**, and **Full** corresponding to the union of all flat and tree features. The POS tagger and the dependency parser were trained on sections 2-21 of the Penn Treebank, followed by an isotonic regression step on section 23 for the dependency parser. We compute precision recall (PR) graphs by varying a threshold on the token level confidence output by the CRF tagger, and summarize the tagger performance using the area under the curve. Table 4 shows the results obtained by the two models under the three feature settings. The model based on probabilistic features consistently outperforms the traditional model, especially when only tree features are used. Dependency parsing is significantly less accurate than POS tagging. Consequently, the improvement for the tree based model is more substantial than for the flat

Model	Tree	Flat	Full
M_3'	**76.78**	**77.02**	**77.96**
M_1	74.38	76.53	77.02

Table 4: Mention detection results.

[2]URL: http://www.nist.gov/speech/tests/ace

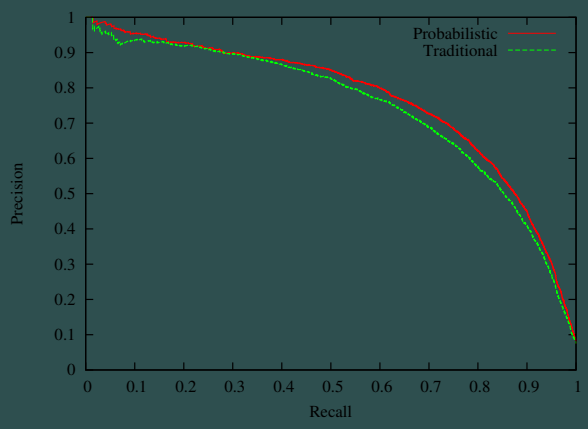

Figure 4: PR graphs for tree features.

model, confirming our expectation that probabilistic features are more useful when upstream stages in the pipeline are less accurate. Figure 4 shows the PR curves obtained for the tree-based models, on which we see a significant 5% improvement in precision over a wide range of recall values.

5 Related Work

In terms of the target task – improving the performance of linguistic pipelines – our research is most related to the work of Finkel et al. (2006). In their approach, output samples are drawn at each stage in the pipeline conditioned on the samples drawn at previous stages, and the final output is determined by a majority vote over the samples from the final stage. The method needs very few samples for tasks such as textual entailment, where the final outcome is binary, in agreement with a theoretical result on the rate of convergence of the voting Gibbs classifier due to Ng and Jordan (2001). While their sampling method is inherently approximate, our full pipeline model M_2 is exact in the sense that feature expectations are computed exactly in polynomial time whenever the inference step at each stage can be done in polynomial time, irrespective of the cardinality of the final output space. Also, the pipeline models M_2 and M_3 and their more efficient alternatives propagate uncertainty during both training and testing through the vector of probabilistic features, whereas the sampling method takes advantage of the probabilistic nature of the outputs only during testing. Overall, the two approaches can be seen as complementary. In order to be applicable with minimal engineering effort, the sampling method needs NLP researchers to write packages that can generate samples from the posterior. Similarly, the new pipeline models could be easily applied in a diverse range of applications, assuming researchers develop packages that can efficiently compute marginals over output substructures.

6 Conclusions and Future Work

We have presented a new, general method for improving the communication between consecutive stages in pipeline models. The method relies on the computation of probabilities for count features, which translates in adding a polynomial factor to the overall time complexity of the pipeline whenever the inference step at each stage is done in polynomial time, which is the case for the vast majority of inference algorithms used in practical NLP applications. We have also shown that additional independence assumptions can make the approach more practical by significantly reducing the time complexity. Existing learning based models can implement the new method by replacing the original feature vector with a more dense vector of probabilistic features[3]. It is essential that every stage in the pipeline produces probabilistic features, and to this end we have described an effective method for associating probabilities with output substructures.

We have shown for NER that simply using the probabilities associated with features that appear only in the top annotation can lead to useful improvements in performance, with minimal engineering effort. In future work we plan to empirically evaluate NER with an approximate version of the full model M_2 which, while more demanding in terms of time complexity, could lead to even more significant gains in accuracy. We also intend to comprehensively evaluate the proposed scheme for computing probabilities by experimenting with alternative normalization functions.

Acknowledgements

We would like to thank Rada Mihalcea and the anonymous reviewers for their insightful comments and suggestions.

[3]The Java source code will be released on my web page.

References

Christopher M. Bishop. 1995. *Neural Networks for Pattern Recogntion*. Oxford University Press.

Aron Culotta and Andrew McCallum. 2004. Confidence estimation for information extraction. In *Proceedings of Human Language Technology Conference and North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, Boston, MA.

Jason M. Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th Conference on Computational linguistics*, pages 340–345, Copenhagen, Denmark.

Jenny R. Finkel, Christopher D. Manning, and Andrew Y. Ng. 2006. Solving the problem of cascading errors: Approximate Bayesian inference for linguistic annotation pipelines. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 618–626, Sydney, Australia.

John Lafferty, Andrew McCallum, and Fernando Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of 18th International Conference on Machine Learning (ICML-2001)*, pages 282–289, Williamstown, MA.

M. Marcus, B. Santorini, and M. A. Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics*, 19(2):313–330.

Andrew Kachites McCallum. 2002. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics (ACL-05)*, pages 91–98, Ann Arbor, Michigan.

Andrew Y. Ng and Michael I. Jordan. 2001. Convergence rates of the Voting Gibbs classifier, with application to bayesian feature selection. In *Proceedings of 18th International Conference on Machine Learning (ICML-2001)*, pages 377–384, Williamstown, MA.

Adwait Ratnaparkhi. 1996. A maximum entropy model for part of speech tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP-96)*, pages 133–141, Philadelphia, PA.

D. Roth and W. Yih. 2004. A linear programming formulation for global inference in natural language tasks. In *Proceedings of the Eighth Conference on Computational Natural Language Learning (CoNLL-2004)*, pages 1–8, Boston, MA.

Bernhard Schölkopf and Alexander J. Smola. 2002. *Learning with kernels - support vector machines, regularization, optimization and beyond*. MIT Press, Cambridge, MA.

Charles Sutton and Andrew McCallum. 2005. Joint parsing and semantic role labeling. In *CoNLL-05 Shared Task*.

Charles Sutton, Khashayar Rohanimanesh, and Andrew McCallum. 2004. Dynamic conditional random fields: Factorized probabilistic models for labeling and segmenting sequence data. In *Proceedings of 21st International Conference on Machine Learning (ICML-2004)*, pages 783–790, Banff, Canada, July.

Vladimir N. Vapnik. 1998. *Statistical Learning Theory*. John Wiley & Sons.

Ben Wellner, Andrew McCallum, Fuchun Peng, and Michael Hay. 2004. An integrated, conditional model of information extraction and coreference with application to citation matching. In *Proceedings of 20th Conference on Uncertainty in Artificial Intelligence (UAI-2004)*, Banff, Canada, July.

Bianca Zadrozny and Charles Elkan. 2002. Transforming classifier scores into accurate multiclass probability estimates. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, Edmonton, Alberta.