A Pipeline Model for Bottom-Up Dependency Parsing

Ming-Wei Chang **Quang Do** **Dan Roth**
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{mchang21, quangdo2, danr}@uiuc.edu

Abstract

We present a new machine learning framework for multi-lingual dependency parsing. The framework uses a linear, pipeline based, bottom-up parsing algorithm, with a look ahead local search that serves to make the local predictions more robust. As shown, the performance of the first generation of this algorithm is promising.

1 System Description

1.1 Parsing as a Pipeline

Pipeline computation is a common computational strategy in natural language processing, where a task is decomposed into several stages that are solved sequentially. For example, a semantic role labeling program may start by using a part-of-speech tagger, than apply a shallow parser to chunk the sentence into phrases, and continue by identifying predicates and arguments and then classifying them.

(Yamada and Matsumoto, 2003) proposed a bottom-up dependency parsing algorithm, where the local actions, chosen from among *Shift, Left, Right*, are used to generate a dependency tree using a shift-reduce parsing approach. Moreover, they used SVMs to learn the parsing decisions between pairs of consecutive words in the sentences [1]. This is a true pipeline approach in that the classifiers are trained on individual decisions rather than on the overall quality of the parser, and chained to yield the global structure. It suffers from the limitations of pipeline processing, such as accumulation of errors, but nevertheless, yields very competitive parsing results.

We devise two natural principles for enhancing pipeline models. First, inference procedures should be incorporated to make robust prediction for each stage. Second, the number of predictions should be minimized to prevent error accumulation. According to these two principles, we propose an improved pipeline framework for multi-lingual dependency parsing that aims at addressing the limitations of the pipeline processing. Specifically, (1) we use local search, a look ahead policy, to improve the accuracy of the predicted actions, and (2) we argue that the parsing algorithm we used minimizes the number of actions (Chang et al., 2006).

We use the set of actions: *Shift, Left, Right, WaitLeft, WaitRight* for the parsing algorithm. The pure *Wait* action was suggested in (Yamada and Matsumoto, 2003). However, here we come up with these five actions by separating actions *Left* into (real) *Left* and *WaitLeft*, and *Right* into (real) *Right* and *WaitRight*. Predicting these turns out to be easier due to finer granularity. We then use local search over consecutive actions and better exploit the dependencies among them.

The parsing algorithm is a modified shift-reduce parser (Aho et al., 1986) that makes use of the actions described above and applies them in a left to right manner on consecutive word pairs (a, b) $(a < b)$ in the word list T. T is initialized as the full sentence. Latter, the actions will change the contents of T. The actions are used as follows:

[1] A pair of words may become consecutive after the words between them become the children of these two words

Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL-X),
pages 186–190, New York City, June 2006.

S*hift*: there is no relation between a and b.

R*ight*: b is the parent of a,

L*eft*: a is the parent of b

W*ait***L***eft*: a is the parent of b, but it's possible that b is a parent of other nodes. Action is deferred.

The actions control the procedure of building trees. When **L***eft* or **R***ight* is performed, the algorithm has found a parent and a child. Then, the function *deleteWord* will be called to eliminate the child word, and the procedure will be repeated until the tree is built. In projective languages, we discovered that action **W***ait***R***ight* is not needed. Therefore, for projective languages, we just need 4 actions.

In order to complete the description of the algorithm we need to describe which pair of consecutive words to consider once an action is taken. We describe it via the notion of the *focus point*, which represents the index of the current word in T. In fact, determining the focus point does not affect the correctness of the algorithm. It is easy to show that any pair of consecutive words in the sentence can be considered next. If the correct action is chosen for the corresponding pair, this will eventually yield the correct tree (but may necessitate multiple cycles through the sentence).

In practice, however, the actions chosen will be noisy, and a wasteful focus point policy will result in a large number of actions, and thus in error accumulation. To minimize the number of actions taken, we want to find a good focus point placement policy.

There are many natural placement policies that we can consider (Chang et al., 2006). In this paper, according to the policy we used, after **S** and **WL**, the focus point moves one word to the right. After **L** or **R**, we adopt the policy *Step Back*: the focus moves back one word to the left. Although the focus placement policy here is similar to (Yamada and Matsumoto, 2003), they did not explain why they made this choice. In (Chang et al., 2006), we show that the policy movement used here minimized the number of actions during the parsing procedure. We can also show that the algorithm can parse a sentence with projective relationships in only one round.

Once the parsing algorithm, along with the focus point policy, is determined, we can train the action classifiers. Given an annotated corpus, the parsing algorithm is used to determine the action taken for each consecutive pair; this is used to train a classifier to predict one of the four actions. The details of the classifier and the features are given in Section 3.

Algorithm 1 Pseudo Code of the dependency parsing algorithm. *getFeatures* extracts the features describing the currently considered pair of words; *getAction* determines the appropriate action for the pair; *assignParent* assigns the parent for the child word based on the action; and *deleteWord* deletes the word which become child once the action is taken.

Let t represents for a word and its part of speech
For sentence $T = \{t_1, t_2, \ldots, t_n\}$
focus$= 1$
while *focus*$< |T|$ **do**
  $\vec{v} = getFeatures(t_{focus}, t_{focus+1})$
  $\alpha = getAction(t_{focus}, t_{focus+1}, \vec{v})$
  if $\alpha =$ **L** or $\alpha =$ **R then**
    $assignParent(t_{focus}, t_{focus+1}, \alpha)$
    $deleteWord(T, focus, \alpha)$
    // performing Step Back here
    $focus = focus - 1$
  else
    $focus = focus + 1$
  end if
end while

When we apply the trained model on new data, the sentence is processed from left to right to produce the predicted dependency tree. The evaluation process is somewhat more involved, since the action classifier is not used as it is, but rather via a local search inference step. This is described in Section 2. Algorithm 1 depicts the pseudo code of our parsing algorithm.

Our algorithm is designed for projective languages. For non-projective relationships in some languages, we convert them into near projective ones. Then, we directly apply the algorithm on modified data in training stage. Because the sentences in some language, such as Czech, etc. , may have multi roots, in our experiment, we ran multiple rounds of Algorithm 1 to build the tree.

1.2 Labeling the Type of Dependencies

In our work, labeling the type of dependencies is a post-task after the phase of predicting the head for the tokens in the sentences. This is a multi-class classification task. The number of the de-

pendency types for each language can be found in the organizer's introduction paper of the shared task of CoNLL-X. In the phase of learning dependency types, the parent of the tokens, which was labeled in the first phase, will be used as features. The predicted actions can help us to make accurate predictions for dependency types.

1.3 Dealing with Crossing Edges

The algorithm described in previous section is primarily designed for projective languages. To deal with non-projective languages, we use a similar approach of (Nivre and Nilsson, 2005) to map non-projective trees to projective trees. Any single rooted projective dependency tree can be mapped into a projective tree by the *Lift* operation. The definition of *Lift* is as follows: *Lift*$(w_j \rightarrow w_k) =$ **parent**$(w_j) \rightarrow w_k$, where $a \rightarrow b$ means that a is the parent of b, and **parent** is a function which returns the parent word of the given word. The procedure is as follows. First, the mapping algorithm examines if there is a crossing edge in the current tree. If there is a crossing edge, it will perform *Lift* and replace the edge until the tree becomes projective.

2 Local Search

The advantage of a pipeline model is that it can use more information that is taken from the outcomes of previous prediction. However, this may result in accumulating error. Therefore, it is essential for our algorithm to use a reliable action predictor. This motivates the following approach for making the local prediction in a pipeline model more reliable. Informally, we devise a local search algorithm and use it as a look ahead policy, when determining the predicted action.

In order to improve the accuracy, we might want to examine all the combinations of actions proposed and choose the one that maximizes the score. It is clearly intractable to find the global optimal prediction sequence in a pipeline model of the depth we consider. The size of the possible action sequence increases exponentially so that we can not examine every possibility. Therefore, a local search framework which uses additional information, however, is suitable and tractable.

The local search algorithm is presented in Algorithm 2. The algorithm accepts two parameters, *model* and *depth*. We assume a classifier that can give a confidence in its prediction. This is represented here by *model*. *depth* is the parameter determining the depth of the local search. *State* encodes the configuration of the environment (in the context of the dependency parsing this includes the sentence, the focus point and the current parent and children for each node). Note that the features extracted for the action classifier depends on *State*, and *State* changes by the *update* function when a prediction is made. In this paper, the *update* function cares about the child word elimination, relationship addition and *focus point* movement.

The search algorithm will perform a search of length *depth*. Additive scoring is used to score the sequence, and the first action in this sequence is performed. Then, the *State* is updated, determining the

Algorithm 2 Pseudo code for the local search algorithm. In the algorithm, $\mathbf{y}$ represents the a action sequence. The function *search* considers all possible action sequences with $|depth|$ actions and returns the sequence with highest score.

```
Algo predictAction(model, depth, State)
x = getNextFeature(State)
y = search(x, depth, model, State)
lab = y[1]
State = update(State, lab)
return lab

Algo search(x, depth, model, State)
maxScore = −∞
F = {y | ||y|| = depth}
for y in F do
  s = 0, TmpState = State
  for i = 1 ... depth do
    x = getNextFeature(TmpState)
    s = s + log(score(y[i], x))
    TmpState = update(TmpState, y[i])
  end for
  if s > maxScore then
    ŷ = y
    maxScore = s
  end if
end for
return ŷ
```

next features for the action classifiers and *search* is called again.

One interesting property of this framework is that we use future information in addition to past information. The pipeline model naturally allows access to all the past information. But, since our algorithm uses the search as a look ahead policy, it can produce more robust results.

3 Experiments and Results

In this work we used as our learning algorithm a regularized variation of the perceptron update rule as incorporated in SNoW (Roth, 1998; Carlson et al., 1999), a multi-class classifier that is specifically tailored for large scale learning tasks. SNoW uses softmax over the raw activation values as its confidence measure, which can be shown to be a reliable approximation of the labels' probabilities. This is used both for labeling the actions and types of dependencies. There is no special language enhancement required for each language. The resources provided for 12 languages are described in: (Hajič et al., 2004; Chen et al., 2003; Böhmová et al., 2003; Kromann, 2003; van der Beek et al., 2002; Brants et al., 2002; Kawata and Bartels, 2000; Afonso et al., 2002; Džeroski et al., 2006; Civit Torruella and Martí Antonín, 2002; Nilsson et al., 2005; Oflazer et al., 2003; Atalay et al., 2003).

3.1 Experimental Setting

The feature set plays an important role in the quality of the classifier. Basically, we used the same feature set for the action selection classifiers and for the label classifiers. In our work, each example has average fifty active features. For each word pair (w_1, w_2), we used their LEMMA, the POSTAG and also the POSTAG of the children of w_1 and w_2. We also included the LEMMA and POSTAG of surrounding words in a window of size $(2, 4)$. We considered 2 words before w_1 and 4 words after w_2 (we agree with the window size in (Yamada and Matsumoto, 2003)). The major difference of our feature set compared with the one in (Yamada and Matsumoto, 2003) is that we included the previous predicted action. We also added some conjunctions of the above features to ensure expressiveness of the model. (Yamada and Matsumoto, 2003) made use of the polynomial kernel of degree 2 so they in fact use more conjunctive features. Beside these features, we incorporated the information of FEATS for the languages when it is available. The columns in the data files we used for our work are the LEMMA, POSTAG, and the FEATS, which is treated as atomic. Due to time limitation, we did not apply the local search algorithm for the languages having the FEATS features.

3.2 Results

Table 1 shows our results on Unlabeled Attachment Scores (UAS), Labeled Attachment Scores (LAS), and Label Accuracy score (LAC) for 12 languages. Our results are compared with the average scores (AV) and the standard deviations (SD), of all the systems participating in the shared task of CoNLL-X.

Our average UAS for 12 languages is 83.54% with the standard deviation 6.01; and 76.80% with the standard deviation 9.43 for average LAS.

4 Analysis and Discussion

We observed that our UAS for Arabic is generally lower than for other languages. The reason for the low accuracy of Arabic is that the sentence is very long. In the training data for Arabic, there are 25% sentences which have more than 50 words. Since we use a pipeline model in our algorithm, it required more predictions to complete a long sentence. More predictions in pipeline models may result in more mistakes. We think that this explains our relatively low Arabic result. Moreover, in our current system, we use the same window size (2,4) for feature extraction in all languages. Changing the windows size seems to be a reasonable step when the sentences are longer.

For Czech, one reason for our relatively low result is that we did not use the whole training corpus due to time limitation [2] . Actually, in our experiment on the development set, when we increase the size of training data in the training phase we got significantly higher result than the system trained on the smaller data. The other problem for Czech is that Czech is one of the languages with many types of part of speech and dependency types, and also the

[2]Training our system for most languages takes 30 minutes or 1 hour for both phases of labeling HEAD and DEPREL. It takes 6-7 hours for Czech with 50% training data.

Language	UAS			LAS			LAC		
	Ours	AV	SD	Ours	AV	SD	Ours	AV	SD
Arabic	76.09	73.48	4.94	60.92	59.94	6.53	75.69	75.12	5.49
Chinese	89.60	84.85	5.99	85.05	78.32	8.82	87.28	81.66	7.92
Czech	81.78	77.01	6.70	72.88	67.17	8.93	80.42	76.59	7.69
Danish	86.85	84.52	8.97	80.60	78.31	11.34	86.51	84.50	4.35
Dutch	76.25	75.07	5.78	72.91	70.73	6.66	80.15	77.57	5.92
German	86.90	82.60	6.73	84.17	78.58	7.51	91.03	86.26	6.01
Japanese	90.77	89.05	5.20	89.07	85.86	7.09	92.18	89.90	5.36
Portuguese	88.60	86.46	4.17	83.99	80.63	5.83	88.84	85.35	5.45
Slovene	80.32	76.53	4.67	69.52	65.16	6.78	79.26	76.31	6.40
Spanish	83.09	77.76	7.81	79.72	73.52	8.41	89.26	85.71	4.56
Swedish	89.05	84.21	5.45	82.31	76.44	6.46	84.82	80.00	6.24
Turkish	73.15	69.35	5.51	60.51	55.95	7.71	73.75	69.59	7.94

Table 1: Our results are compared with the average scores. UAS=Unlabeled Attachment Score, LAS=Labeled Attachment Score, LAC=Label Accuracy, AV=Average score, and SD=standard deviation.

length of the sentences in Czech is relatively long. These facts make recognizing the HEAD and the types of dependencies more difficult.

Another interesting aspect is that we have not used the information about the syntactic and/or morphological features (FEATS) properly. For the languages for which FEATS is available, we have a larger gap, compared with the top system.

5 Further Work and Conclusion

In the shared task of CoNLL-X, we have shown that our dependency parsing system can do well on multiple languages without requiring special knowledge for each of the languages.

From a technical perspective, we have addressed the problem of using learned classifiers in a pipeline fashion, where a task is decomposed into several stages and classifiers are used sequentially to solve each stage. This is a common computational strategy in natural language processing and is known to suffer from error accumulation and an inability to correct mistakes in previous stages. We abstracted two natural principles, one which calls for making the local classifiers used in the computation more reliable and a second, which suggests to devise the pipeline algorithm in such a way that it minimizes the number of actions taken.

However, since we tried to build a single approach for all languages, we have not fully utilized the capabilities of our algorithms. In future work we will try to specify both features and local search parameters to the target language.

Acknowledgement This research is supported by NSF ITR IIS-0428472, a DOI grant under the Reflex program and ARDA's Advanced Question Answering for Intelligence (AQUAINT) program.

References

A. V. Aho, R. Sethi, and J. D. Ullman. 1986. Compilers: Principles, techniques, and tools. In *Addison-Wesley Publishing Company, Reading, MA*.

A. Carlson, C. Cumby, J. Rosen, and D. Roth. 1999. The SNoW learning architecture. Technical Report UIUCDCS-R-99-2101, UIUC Computer Science Department, May.

M. Chang, Q. Do, and D. Roth. 2006. Local search for bottom-up dependency parsing. Technical report, UIUC Computer Science Department.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*.

D. Roth. 1998. Learning to resolve natural language ambiguities: A unified approach. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 806–813.

H. Yamada and Y. Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *IWPT2003*.