Bidirectional Dependency Parser for Indian Languages

Aswarth Abhilash and Prashanth Mannem
Language Technologies Research Center
International Institute of Information Technology
Hyderabad, AP 500032, India
{abhilash.d, prashanth}@research.iiit.ac.in

Abstract
In this paper, we apply bidirectional dependency parsing algorithm for parsing Indian languages such as Hindi, Bangla and Telugu as part of NLP Tools Contest, ICON 2010. The parser builds the dependency tree incrementally with the two operations namely proj and non-proj. The complete dependency tree given by the unlabeled parser is used by SVM (Support Vector Machines) classifier for labeling. The system achieved Labeled Attachment Score (LAS) of 84.79%, 69.09%, 68.95% for Hindi, Bangla and Telugu. While using fine-grained dependency labels, it achieved LAS of 83.12%, 65.97%, 67.45% respectively.

1 Introduction
The task of NLP tools contest in ICON 2009 (Husain, 2009) is to do dependency parsing for Indian languages Hindi, Bangla and Hindi given the training, development and testing datasets. Various teams had participated in it. Ambati et al (2009), Nivre (2009a) used the transition-based Malt Parser (2007b) for their experiments. They explored various features with various configuration settings in it. Zeman (2009) combined the output of the various well-known dependency parsers into a superparser by using a simple voting method to perform the task. Mannem (2009b) used the bi-directional dependency parsing algorithm proposed by Shen and Joshi (2008). It does a best first search over the sentence and picks the most confident dependency relation between a pair of words every time. It builds the tree incrementally without following a specific direction (either from left-to-right or right-to-left). The search can start at any position and expands the partial analyses in any direction. A slight variant of this approach has been proposed by Goldberg and Elhadad (2010). It can be viewed as a special case of Shen and Joshi (2008) with beam search one. This year also, the same task has been put up by the organizers with the larger annotated data than in 2009.

In this work, we apply this bidirectional parsing algorithm by using two actions to build the dependency tree for a sentence. The two actions distinguish nodes that have an outgoing non-projective arc and those that don’t. Once the complete dependency tree is generated by the unlabeled parser, we used LIBSVM tools (Wu et al., 2004) for labeling. Our system achieved Labeled Attachment Score of 83.12%, 65.97% and 67.45% for Hindi, Telugu and Bangla respectively with fine-grained dependency labels.

2 Our approach
The relatively free word order property in Indian languages indicate the presence of non-projective dependencies in them. A relation is non-projective if the yield/projection of the parent in a relation

Figure 1: Example of dependency tree built using the two actions proj and non-proj
is not contiguous. 18.75% of Hindi sentences in the released treebank are non-projective. Some of the sentences have multiple non-projective arcs within a sentence. For Bangla, it is 6.02%. Telugu treebank has very few non-projective structures (0.82%) due to the kind of corpus chosen for treebank annotation. The average sentence lengths in the Hindi, Bangla and Telugu treebanks are 21.92 words, 6.42 words and 3.94 words respectively. The smaller sentence length in Telugu explains the low occurrence of non-projectivity in Telugu treebank.

Non-projectivity has been handled in incremental parsers earlier. Nivre and Nilsson (2005) used a pseudo-projective parsing technique to retrieve non-projective dependencies as a post processing step in a transition based parser. Nivre (2009b) proposed a transition system which can parse arbitrary non-projective trees by swapping the order of words in the input while parsing.

In our approach to dependency parsing, we use two different operations to connect nodes that have outgoing non-projective arcs and those that don’t. A dependency tree is built by connecting nodes with two operations proj and non-proj. The operation non-proj is used to connect the head in a non-projective relation to its parent. proj is used to connect all the other nodes in a dependency tree to their parents. In the example dependency tree given in Figure 1, there is a non-projective arc from w1 to w5 connected through non-proj operation. Nodes connected through non-proj indicate scope for an outgoing non-projective arc from them. During parsing, the set of candidate parents for a node will include all the nodes which satisfy projectivity and also those nodes which have been connected through a non-proj operation.

In this work, we apply the bidirectional parsing approach described in Shen and Joshi (2008) for Indian Languages such as Hindi, Bangla and Telugu using the two operations defined above. The original implementation of the parsing algorithm by Libin Shen performed unlabeled LTAG dependency parsing by training on LTAG-spinal treebank. Later is has been extended to do labeled dependency parsing by Mannem (2009b). Since the size of the annotated treebank is considerably not large and due to the presence of more dependency labels (>30), the labeled bidirectional dependency parsing algorithm extended by Mannem (2009b) doesnot perform well. So, we used LIBSVM tools (Wu et al., 2004) for labeling.

Mannem et al. (2009a) extracted LTAG-spinal treebank for Hindi from a dependency treebank by using the adjunction operation to handle non-projective structures and attachment to handle projective relations other than coordination structures. They trained the bidirectional LTAG dependency parser (Shen and Joshi, 2008) on the extracted Hindi LTAG-spinal treebank with good results.

3 Unlabeled Bidirectional Dependency Parser (UBDP)

Shen and Joshi (2008) proposed a bidirectional incremental parsing algorithm which searches the sentence for the best score dependency hypothesis in both the directions (left-to-right & right-to-left). The search can begin at any position and can expand the intermediate results in any direction. The order of search is learned automatically.

The implementation of this parsing framework by Libin Shen could only do unlabeled LTAG dependency parsing and the training data was LTAG-spinal treebank in LTAG-spinal format. We use the same algorithm by doing minor changes to it.

In the rest of the section, we give an overview of the parsing algorithm along with the training and inference processes presented in (Shen and Joshi, 2008). We mention our extensions to the original parsing framework wherever appropriate.

3.1 Parsing Algorithm

We are given a linear graph \( G = (V, E) \) with vertices \( V = \{v_i\} \) and \( E(v_{i-1}, v_i) \) with a hidden structure \( U = \{u_k\} \). The hidden structure \( u_k = (v_{s_k}, v_{e_k}, l_k) \), where vertex \( v_{s_k} \) depends on vertex \( v_{s_k} \) with label \( l_k \in L \) is what we want to find (the parse tree). \( L \) is the full list of dependency labels occurring in the corpus. A sentence is a linear graph with an edge between the adjacent words. A fragment is a connected sub-graph of \( G(V, E) \). Each fragment \( x \) is associated with a set of hypothesized hidden structures, or fragment hypotheses for short: \( Y^x = \{y^1_x, \cdots, y^2_x\} \). Each \( y^x \) is a possible fragment hypothesis of \( x \). A fragment hypothesis represents a possible parse analysis for a fragment. Initially, each word with its POS tag comprises a fragment.

Let \( x_i \) and \( x_j \) be two fragments, where \( x_i \cap x_j = \emptyset \) and are directly connected via an edge in \( E \). Let \( y^{x_i} \) be a fragment hypothesis of \( x_i \) and \( y^{x_j} \) a fragment hypothesis of \( x_j \). We can combine the hy-
hypotheses for two nearby fragments with one of the labels from \( L \). Suppose we choose to combine \( y^x_i \) and \( y^x_j \) with an operation \( R_{\text{type},\text{label},\text{dir}} \) to build a fragment hypothesis for \( x_k = x_i \cup x_j \). The output of the operation is

\[
y^x_k = R_{\text{type},\text{label},\text{dir}}(x_i, x_j, y^x_i, y^x_j) \supseteq y^x_i \cup y^x_j
\]

where \( \text{type} \in \{ \text{proj}, \text{non-proj} \} \) is the type of operation, \( \text{label} \) is the dependency label from \( L \) and \( \text{dir} \in \{ \text{left}, \text{right} \} \), representing whether the left or the right fragment is the parent. \( y^x_i \) and \( y^x_j \) stand for the fragment hypotheses of the left and right fragments \( x_i \) and \( x_j \).

An operation \( R \) on fragment hypotheses \( R.y^x_i \) and \( R.y^x_j \) generates a new hypotheses \( y(R) \) for the new fragment which contains both the fragments \( R.x_i \) and \( R.x_j \). The score of an operation is defined as

\[
s(R) = W \cdot \phi(R)
\]

where \( s(R) \) is the score of the operation \( R \), which is calculated as the dot product of a weight vector \( W \) and \( \phi(R) \), the feature vector of \( R \). The score of the new hypothesis is the sum of the scores of the operation and the involving fragment hypotheses.

\[
\text{score}(y(R)) = s(R) + \text{score}(R.y^x_i) + \text{score}(R.y^x_j)
\]

The feature vector \( \phi(R) \) is defined on \( R.y^x_i \) and \( R.y^x_j \), as well as the context hypotheses. If \( \phi(R) \) only contains information in \( R.y^x_i \) and \( R.y^x_j \), its called \textit{level-0 feature dependency}. If features contain information of the hypotheses of nearby fragments, its called \textit{level-1 feature dependency}. A \textit{chain}, is used to represent a set of fragments, such that hypotheses of each fragment always have feature dependency relations with some other fragments within the same chain. Furthermore, each fragment can only belong to one chain. A set of related fragment hypotheses is called a \textit{chain hypothesis}. For a given chain, each fragment contributes a fragment hypothesis to build a chain hypothesis. Beam search is used with a predefined beam width to keep the top \( k \) chain hypotheses for each chain. The score of a chain hypothesis is the sum of the scores of the fragment hypotheses in this chain hypothesis. For chain hypothesis \( c \),

\[
\text{score}(c) = \sum_{y^x \in \text{fragment hypothesis } y^x \text{ of } c} \text{score}(y^x)
\]

A \textit{cut} \( T \) of a given sentence, \( T = \{ c_1, c_2, \ldots, c_m \} \), is a set of chains satisfying

- exclusiveness: \( \cup c_i \cap \cup c_j = \emptyset \), \( \forall i, j \), and
- completeness: \( \bigcup (\cap T) = V \).

Furthermore, \( H^T = \{ H^r | c_i \in T \} \) is used to represent the sets of fragment hypotheses for all the fragments in cut \( T \). At every point in the parsing process, a priority queue of candidate operations \( Q \) is maintained. \( Q \) contains all the possible operations for the fragments and their hypotheses in cut \( T \). \( s(R) \) is used to order the operations in \( Q \).

With the above formal notations, we now list the inference and learning algorithms in Algorithm 1 and Algorithm 2.

\begin{algorithm} 
\caption{UBDP: Inference Algorithm}
\begin{algorithmic}
\STATE \textbf{INPUT:} graph \( G(V, E) \) and weight vector \( W \); 
\STATE \textbf{INITIATE} cut \( T \), hypotheses \( H^T \), queue of candidate operations \( Q \); 
\WHILE {\( Q \) is not empty} 
\STATE operation \( y \leftarrow \arg \max_{\text{op} \in Q} \text{score (op, W)} \); 
\STATE \textbf{UPDATE} \( T \), \( H^T \), \( Q \) with \( y \); 
\ENDWHILE
\end{algorithmic}
\end{algorithm}

### 3.2 Decoding

Algorithm 1 describes the procedure of building hypotheses incrementally on a given linear graph \( G = (V, E) \). Parameter \( k \) is used to set the beam width of search. Weight vector \( w \) is used to compute the score of an operation. First, the cut \( T \) is initiated by treating each vertex in \( V \) as a fragment and a chain. Then the initial hypotheses for each vertex/fragment/chain are set with the POS tag for each word. The priority queue \( Q \) is used to collect all the possible operations over the initial cut \( T \) and hypotheses \( H^T \). Whenever \( Q \) is not empty, the chain hypothesis with highest score on operation according to a given weight vector \( w \) is searched for and the cut along with its hypotheses are updated with that chain hypothesis. The candidate queue \( Q \) is then updated by removing operations depending on the chain hypotheses that have been removed from \( H^T \), and adding new operations depending on those chain hypotheses.

### 3.3 Training

For each given training sample \( (G_r, H_r) \), where \( H_r \) is the gold standard hidden structure of graph \( G_r \), cut \( T \), its hypotheses \( H^T \) and candidate queue \( Q \) are initialized. The gold standard \( H_r \) is used to guide the search. The candidate \( (x, y) \) with the highest operation score in \( Q \) is selected. If \( y' \) is
Algorithm 2 UBDP: Training Algorithm

\[ W \leftarrow 0; \]
\[ \text{for } \text{round} = 1..T, \ i = 1..n \ \text{do} \]
\[ \text{LOAD graph } G_r(V, E), \text{ hidden structure } H_r; \]
\[ \text{Initiate cut } T, \text{ hypotheses } H^T, \text{ queue } Q; \]
\[ \text{while } Q \text{ is not empty do} \]
\[ \text{operation } y \leftarrow \arg_{op, Q} \text{ max score } (\text{op, } W); \]
\[ \text{if } \text{compatible}(H_r, y) \text{ then} \]
\[ \text{Update } T, \ H^T, \ Q \text{ with } y; \]
\[ \text{else} \]
\[ y^* \leftarrow \text{searchCompatible}(Q, y); \]
\[ \text{promote}(y^*) \]
\[ \text{demote}(y) \]
\[ \text{UPDATE } Q \text{ with } W; \]
\[ \text{end if} \]
\[ \text{end while} \]
\[ \text{end for} \]

compatible with \( H_r \), the cut \( T \), hypotheses \( H^T \) and \( Q \) are updated. If \( y^* \) is incompatible with \( H_r \), \( y^* \) is treated as a negative sample, and a positive sample \( y^* \) compatible with \( H_r \) is searched for in \( Q \). If such compatible hypothesis doesn’t exist, the hypothesis with highest score in \( Q \) and compatible with \( H_r \) is searched. Then, the weight vector \( w \) is updated with \( y \) and \( y^* \). At the end, the candidate queue \( Q \) is updated with the new weights \( w \) to compute the score of operation. Perceptron learning with margin is used in the training and averaged Perceptron for inference. Algorithm 2 lists the training procedure.

3.4 UBDP + SVM classifier

The unlabeled dependency tree given by UBDP is used by SVM (Support Vector Machines) classifier for labeling. We have performed experiments using various features with different options provided by SVM tool.

The best accuracy for Telugu and Bangla are obtained using the feature set described in Table 1. .cnk denotes the chunk label. .litem denotes the word/root form. .afx denotes the affix information. .pa denotes the head word and .ch denotes the dependent. Apart from the features mentioned above, the features like number of children, number of siblings, difference in positions of the child and parent and pos list from the root of a sentence to the dependent are also considered. For Hindi, we have also added structural features involving part of speech tags of all the words involved in the partial trees while building the tree incrementally. Due to the availability of smaller treebanks for both Telugu and Bangla, adding these structural features didn’t help in improving the accuracy.

<table>
<thead>
<tr>
<th>(a) Unigram features involving pa and ch</th>
</tr>
</thead>
<tbody>
<tr>
<td>.pa.litem</td>
</tr>
<tr>
<td>.pa.pos</td>
</tr>
<tr>
<td>.pa.cnk</td>
</tr>
<tr>
<td>.pa.afx</td>
</tr>
<tr>
<td>(pa.litem + pa.pos) (ch.litem + ch.pos)</td>
</tr>
<tr>
<td>(pa.litem + pa.cnk) (ch.litem + ch.cnk)</td>
</tr>
<tr>
<td>(pa.litem + pa.afx) (ch.litem + ch.afx)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Bigram features involving pa and ch</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pa.litem + pa.pos + ch.litem + ch.pos)</td>
</tr>
<tr>
<td>(pa.litem + pa.afx + ch.litem + ch.afx)</td>
</tr>
<tr>
<td>(pa.pos + pa.afx + ch.pos + ch.afx)</td>
</tr>
<tr>
<td>(pa.cnk + pa.afx + ch.cnk + ch.afx)</td>
</tr>
<tr>
<td>(pa.litem+pa.afx+pa.pos+ch.litem+ch.afx+ch.pos)</td>
</tr>
<tr>
<td>(pa.litem+pa.afx+pa.cnk+ch.litem+ch.afx+ch.cnk)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c) Features involving all the siblings sb of ch</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pa.pos + ch.pos + sb.pos)</td>
</tr>
<tr>
<td>(pa.litem + pa.pos + ch.pos + sb.pos)</td>
</tr>
<tr>
<td>(pa.litem + pa.afx + ch.pos + sb.pos)</td>
</tr>
<tr>
<td>(pa.pos + pa.afx + ch.pos + sb.afx + sb.pos)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(d) Features involving the context words b.i i could range from -2 to +2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pa.pos + ch.pos + b.i.pos)</td>
</tr>
<tr>
<td>(pa.pos + ch.pos + b.i.cnk)</td>
</tr>
<tr>
<td>(pa.pos + ch.pos + b.i.afx)</td>
</tr>
<tr>
<td>(pa.litem + pa.pos + ch.pos + b.i.pos)</td>
</tr>
<tr>
<td>(pa.litem + pa.pos + ch.pos + b.i.cnk)</td>
</tr>
<tr>
<td>(pa.litem + pa.pos + ch.pos + b.i.afx)</td>
</tr>
</tbody>
</table>

Table 1: Features used by the parser for all the three languages

4 Data

Annotated training and development data for Hindi, Telugu and Bangla were released as part of the contest (ICON, 2010). Some sentences have been discarded due to the presence of errors in the treebank. The final data (training+development) contained 3515 Hindi, 1450 Telugu and 1129 Bangla sentences. 68 Bangla (6.02%), 659 (18.75%) Hindi and 12 (0.82%) Telugu sentences were non-projective in the entire corpus.
For the testing phase of the contest, the parser was trained on the entire released data with the best performing feature set and the unannotated test data was parsed with the model obtained.

5 Experiments and Results

The Unlabeled Bidirectional Dependency Parser (UBDP) with the features listed in Table 1 was used for developing models for the three languages. This feature set was arrived at by doing a 5-fold cross validation on the released data (training+development) for Hindi. Sentence context (SC) and local context (LC) were varied over a range of 0 to +/-3. The best accuracy over the 5-fold cross validation was reported with a SC of 2 and LC of 1. This feature set was used for Telugu and Bangla for the accuracies reported in this section. For Hindi, combining more structural features as mentioned in the previous section with this feature set performed well. The results shown here are different from the ones submitted. This is due to the selection of incorrect models for testing while submitting the outputs.

The annotated data for all the three languages was released with fine-grained and coarse-grained dependency labels separately by the organizers. The unannotated test data released by the organizers had 321 sentences for Hindi and 150 sentences for Bangla and Telugu. The accuracies achieved by our system on the data with fine-grained dependency labels are in Table 2. UAS is the Unlabeled Attachment Score, LAS is the Labeled Attachment Score and LA is the Labeled Accuracy. UAS, LAS and LA are standard evaluation metrics in the literature for dependency parsing (Nivre et al., 2007a).

![Table 2: Accuracies on data annotated with fine-grained dependency labels](image)

Though the available annotated data for all the three languages is small, the average UAS (88.02%) is high. One reason is because of the use of gold-standard chunks and part of speech tags during training and testing. The other reason is due to the presence of large number of intra chunk relations when compared to inter chunk relations. The average LAS is however considerably low at 72.18% when compared to the avg. UAS (88.02%). This is because of the small size of the available treebanks and also due to the large number of dependency labels used in the annotation (>30). The dependency labels used in the treebanks are syntactico-semantic in nature, marking labels is even more difficult than the one with pure syntactic labels.

The organizers also released treebanks for the three languages with coarse-grained dependency labels. Table 3 shows the performance of the parser on this data. The avg. LAS has increased to 77.06% because of the reduction in the number of dependency labels.

![Table 3: Accuracies on data annotated with coarse-grained dependency labels](image)

6 Error Analysis

For all languages, in case of labeled accuracy, the classifier is not able to predict the correct label among the labels k1, k2 and pof. It is due to the wrong dependencies produced by the unlabeled parser and also due to the absence of specific case markers which makes hard to disambiguate k1 and k2. While in case of unlabeled parsing, the frequent errors are the inter chunk relations involving nouns and conjuncts. For Bangla and Telugu, lower labeled accuracy is due to the availability of smaller treebank and the classifier doesn’t have enough samples to learn properly. Since the number of dependency labels are large, the number of hypotheses generated are even larger which allows the classifier to take an improper decision.

7 Conclusion and Future Work

In this work, we used the approach to dependency parsing which builds the tree using two actions proj and non-proj. non-proj is used to connect nodes that lead to non-projective arcs. A node connected with this operation is available/visible
for combination beyond its projective scope. \textit{proj} is used to connect the rest of the nodes. A bidirectional dependency parsing algorithm is used with these two operations for our system. The unlabeled dependency tree generated is used by SVM (Support Vector Machines) classifier for labeling. The parser was trained on the entire released data and we got LAS of 84.79\%, 69.09\% and 68.95\% for Hindi, Bangla and Telugu respectively for the coarse-grained dependency tagset. While using fine-grained dependency labels, the LAS for the three languages are 83.12\%, 65.97\% and 67.45\% respectively. Our immediate future work consists of experiments for proper selection of features and also extending bidirectional parser to provide features easily.

**References**


