

# An Automatic Approach to Treebank Error Detection Using a Dependency Parser

Bhasha Agrawal<sup>1</sup>, Rahul Agarwal<sup>1</sup>, Samar Husain<sup>2</sup>, and Dipti M. Sharma<sup>1</sup>

<sup>1</sup>IIT-Hyderabad, India, <sup>2</sup>University Of Potsdam, Germany

**Abstract.** Treebanks play an important role in the development of various natural language processing tools. Amongst other things, they provide crucial language-specific patterns that are exploited by various machine learning techniques. Quality control in any treebanking project is therefore extremely important. Manual validation of the treebank is one of the steps that is generally necessary to ensure good annotation quality. Needless to say, manual validation requires a lot of human time and effort. In this paper, we present an automatic approach which helps in detecting potential errors in a treebank. We use a dependency parser to detect such errors. By using this tool, validators can validate a treebank in less time and with reduced human effort.

## 1 Introduction

Treebanks are an essential resource for developing solutions to various NLP related problems. As a treebank provides important linguistic knowledge, its quality is of extreme importance. The treebank used for experiments in this work is a part of the new multi-layered and multi-representational Hindi Treebanking project [8, 19] (the dependency scheme is based on Computational Paninian Grammar (CPG) model, first proposed by [7]) whose target is 450k words. As is generally the case, most of the annotation process is either completely manual or semi-automatic, and the validation is completely manual.<sup>1</sup>

The process of developing a treebank involves manual or semi-automatic annotations of linguistic information. A semi-automatic procedure involves annotating the grammatical information using relevant NLP tools (eg. POS taggers, chunker, parsers, etc.). The output of these tools is then manually checked and corrected. Both these procedures may leave errors in the treebank on the first attempt. Therefore, there is usually another step called *validation* in which these errors are manually identified and corrected. But the validation process is as time-consuming as annotation process. As the data has already been annotated carefully, we need tools that can supplement the validators' task with a view of making the overall task fast, without compromising on reliability. Using such a

---

<sup>1</sup> Dependency annotation scheme of this treebank conforms to [6]

tool, a validator can directly go to error instances and correct them. So, the tool must have high recall. A human validator can reject unintuitive errors without much effort, so one can compromise a little bit on precision.

In this paper, we propose such a strategy. The identified errors are classified under two categories (*attachment error* and *labeling error*) for the benefit of the validators, who may choose to correct a specific type of error at one time. Our method to identify such errors involves using a dependency parser.

If we can demonstrate considerable benefit/relevance of such a strategy, it might then be a good idea to incorporate it in a treebank development pipeline.

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 describes data used in the experiments. In section 4, we describe our approach to error detection using a dependency parser. Here, we also show the results obtained using this approach. In section 5, we compare our results with older techniques. Section 6 concludes the paper.

## 2 Related Work

Error detection has become an active topic of research over the last decade due to increase in demand for high quality annotated corpora. Some earlier methods employed for error detection in syntactic annotation (mainly POS and chunk), are by [9] and [10]. [18] employed a method similar to [10], to automatically correct errors at the dependency level. Their main idea was to reproduce the gold standard data using MSTParser, MALTParser and MDParse. They use the outputs of any two parsers to detect error nodes, and mark a node as an error if tags predicted by the two parsers differ from the one in the gold data. In case the predicted tags are different, they use the output of the third parser to check if the tag predicted by the third parser matches with any of the tags predicted by the other two parsers. If the tag matches with any of the other two tags, changes are made in the gold data. Using large corpora, [17] and [13] employed error mining techniques. Other efforts towards detection of annotation errors in treebanks include [12] and [14]. Most of the aforementioned techniques work well with large corpora where the word-type frequencies are high. Thus, none of them account for data sparse conditions except for [13]. Moreover, the techniques employed by [17], [13] and [18] rely on the output of a reliable state-of-the-art parser.

Work by [3], [5] and [2] detects errors in a Hindi dependency treebank. They used a combination of both rule-based and hybrid systems to detect annotation errors. A rule-based system contributes towards increasing the precision of the system; it uses robust rules formed using annotation guidelines and the CPG framework, whereas a hybrid system is a combination of a statistical module with rule-based post-processing. The statistical module detects potential errors from the treebank and the rule-based post-processing module prunes out false positives, with the help of robust and efficient rules to increase the precision of the overall system.

[2] improved over the PBSM (Probability Based Statistical Module) used by [5]. They extended PBSM used by [5] by adding more linguistically motivated features like dependency labels of the parent, grandparent and count of its children. They also improved the hybrid system by placing a new module ‘Rule based correction’ before the statistical module.

Previous works show that parser outputs may help in detecting errors in Treebank [17, 13] and sometimes, automatic error correction as well [18]. Hence, in our approach, we use MaltParser<sup>2</sup> [16] to detect potential errors in Hindi Dependency Treebank. Baseline accuracy with MaltParser for Hindi are 77.58%, 88.97% and 80.48% for LAS<sup>3</sup>, UAS<sup>4</sup> and LA<sup>5</sup> respectively [4]. Using the parser output alone does not help much because the state-of-the-art parsers for free word order languages like Hindi perform lower than those for fixed word order language like English. So we also use an algorithm which uses the output produced by MaltParser to detect potential errors in the treebank.

### 3 Our Approach

In this section we describe our approach to error detection using a dependency parser. We detect errors at dependency level annotation. We use inter-chunk dependency trees rather than expanded trees, i.e., dependency relations are marked only among chunk heads in a sentence and dependency relations within a chunk<sup>6</sup> are not considered.

#### 3.1 Using Parser Output to Detect Potential Errors

In this strategy, we use the output of the parser to detect potential errors in the annotated data. We make an assumption that most of the decisions taken by the parser and the humans are similar, i.e., similar types of errors are made by both. Both, the parser and the annotators find it difficult to parse certain type of structures and both are good at other constructions.

The question may arise why we are using a parser to detect errors made by humans when human performance is better, as compared to a parser’s in all respects? The answer is simple. Decisions taken by human beings may sometimes vary but a statistical parser (e.g. MaltParser) gives consistent decisions. Even if the parser gives a wrong output, it will always give the same wrong result for a specific case/instance.

Reasons of human error can be:

---

<sup>2</sup> MaltParser (version 1.4.1)

<sup>3</sup> LAS - Labelled Attachment score (Percentage of words that are assigned both correct head and correct label)

<sup>4</sup> UAS - Unlabelled Attachment Score (Percentage of words that are assigned correct head)

<sup>5</sup> LA - Labelled Accuracy (Percentage of words that are assigned correct label)

<sup>6</sup> A typical chunk consists of a single content word surrounded by a constellation of function words [1]. Chunks are normally taken to be a ‘correlated group of words’.

1. *Random errors*: Decisions of annotators may change according to their state of mind. They may make random errors if they are not able to concentrate on their work. Though they know the correct decision, they may unknowingly make some errors because of some psychological conditions. Everything being right, an annotator can also by mistake make wrong decisions.
2. *Errors due to unawareness of a concept/Misinterpretation of guidelines decision*: Annotators may make the same mistake again and again if they do not know the concept properly or do not know the correct decision to be taken. Misinterpretation of certain guidelines decision may also cause consistent errors.
3. *Errors in the guidelines*: Some sections of the guidelines are always evolving. Some other sections might have errors due to oversight. Annotators will make consistent mistakes in such a case unknowingly.

The types of arcs for which the parser (MaltParser) consistently gives wrong decisions are<sup>7</sup>:

1. *Long distance arcs*: MaltParser is greedy in marking dependencies and searches for the head of a word in the vicinity of the word. It adds a node as the parent of another node as soon as it finds a suitable candidate for it. Hence, it generally marks long distance dependencies also as short distance dependencies.
2. *Non-projective arcs*: The algorithm used in MaltParser always produces projective graphs and cannot handle non-projective dependencies. [15]

The idea here is to use consistent decisions of the parser to mark the human errors. If the parser performs well in some constructions, it will, most of the time, give a correct decision and we can compare annotator's decision with that of the parser and if the decisions made by them are different, we can infer that the decision taken by the annotator may be incorrect (as the parser's decision was right and annotator's decision does not match). Even if the parser takes wrong decisions in other types of constructions, it will consistently mark them wrong. If we know these cases, we can again compare such decisions with annotator's decisions and if they match, we can say that decision taken by the annotator is incorrect (as the parser was wrong and annotator's decision is similar to the parser's). Here, we are only marking the potential errors, so even if we mark a correct decision as an incorrect one, it will not create much problem as it can be checked further during the process of validation. But we must not leave any error unmarked because this would lead to an erroneous treebank.

Our classification of parser errors is taken from [11]. Here, parser errors were classified based on:

---

<sup>7</sup> These errors are because of the algorithm used (Arc-eager [15]) for parsing. We have used this algorithm because it outperformed other algorithms for hindi. [4]

1. *Edge<sup>8</sup> Type and Non-Projective Edge*: The edges were categorized based on the distinct dependency labels on the edges.
2. *Edge Depth*: Depth of a tree is the total number of levels of links that it has. Consequently, the depth of an edge is basically the level at which it exists in the tree.

Examples of edge types from [11] are:

1. Main
2. Intra Clausal
  - a) Verb Argument Structure
    - i) Complement
    - ii) Adjunct
  - b) Non-verbal
    - i) Noun-modifier
    - ii) Adjective-modifier
    - iii) Apposition
    - iv) Genitive
  - c) Others
    - i) Co-ordination
    - ii) Complex Predicate
    - iii) Others
3. Inter Clausal
  - a) Co-ordination
  - b) Sub-ordination
    - i) Conjunction
    - ii) Relative Clause
    - iii) Clausal Complement
    - iv) Apposition
    - v) Verb Modifier

First of all, Maltparser was trained using manually validated training data and development data was parsed using the trained model. Then using the above mentioned classification, the data was classified according to different categories (edge type and edge depth). After that, the parsed output was analyzed by comparing it with gold data to find out the classes of arcs for which the parser works well and for which it doesn't. Using this analysis, a knowledge base was prepared, which contains the information about the arcs being correctly or incorrectly parsed for various edge type and edge depth.

With the help of this knowledge base, confidence level for different categories of arcs (different edge-types and different edge-depths) was set as 1, if the parser was confident on giving correct decisions for that type of arcs and 0 if it was less confident of giving correct decisions for that type of arcs. For example:

---

<sup>8</sup> An edge in a dependency tree is the arc that relates two nodes (which are basically words). These are binary asymmetric relations with labels that specify the relation type.

- The parser gives wrong attachments and labels for most of the inter-clausal arcs. They are parsed correctly only if they are at depth 1 or 2. e.g. ‘Inter Clausal Relative Clause’ is only parsed correctly if it occurs at depth 1, so confidence level for this class of arcs is set to 1 if it occurs at depth 1 and is set to 0 if the arc occurs at any other depth.
- Most of the Intra Clausal arcs are parsed correctly but only if the sentence is not very long. Intra Clausal arcs in general occur at the leaves of a dependency tree. For Intra Clausal arcs, confidence level is 1 for short depths and 0 when depth is high. e.g. Confidence level is 1 for ‘Intra Clausal, Verb Argument Structure (Complement)’ upto depth 10 and confidence level is 0 when this class of arcs occur at a depth greater than 10.

For boundary conditions, confidence level was set using the results obtained from development data. e.g. for arcs of category ‘Intra Clausal, Verb Argument Structure (Complement)’, confidence level was set 1 for depth 9, 1 for depth 10, 0 for depth 11 and 0 for depth 12 because confidence level 1 for depth 9 gave better results for development data than confidence level 0. Confidence level 0 for depth 11 was better than 1 and so on. In this manner, database for confidence level (knowledge base) for each category of arcs was prepared.

After preparing the confidence level database (knowledge base) using the development data, testing data was also parsed using the trained model and was flagged with potential errors with the help of confidence level mentioned in the knowledge base. Testing was performed as follows:

For each arc, its category was checked. For that category, confidence level was taken from the knowledge base. If confidence level was 1 and parser output was different from the annotated output, the arc was marked to be a potential error. Again, if confidence level is 0 and parser output matches annotated output, the arc was marked as a potential error. This can be explained in detail as follows:

1. We extract cases for which the parser is highly confident (confidence level 1) on giving correct labels and attachments.

Here, the parser gives correct decisions and if parser output doesn’t match the annotated one, there is a possibility that the annotator annotated it wrong. So, we mark that node as a potential error node.

For example, let there be a category of arcs containing the labels ‘k1’, ‘k2’, ‘k4’, etc. and the parser is confident enough on giving correct attachment and label for this class of arcs. Then, if an arc  $x \rightarrow y$  is labeled as ‘k1’ by parser and the annotators marked it ‘k2’, it is flagged as a potential error.

2. Next, we extract cases for which the parser is less confident (confidence level 0) on giving correct labels.

In this, the decision taken by the parser might be incorrect and if the parser output matches the annotated output, there is a possibility that the annotator also made the same mistake. Hence, we flag that node as a potential error node.

For example, again let there be a category of arcs containing the labels ‘k1’, ‘k2’, ‘k4’, etc. and the parser is less confident here, of giving correct

attachment and label for this class of arcs. Then, if an arc  $x \rightarrow y$  is labeled as ‘k1’ by the parser and the annotators also mark it ‘k1’, it is flagged as a potential error.

After this precision and recall are calculated which are shown in Table 1.

**Table 1.** Results of validation using parser output

<b>Data</b>	<b>Attachment Labeling</b>	
<b>Precision/Recall</b>		
<b>Development</b>	56.72/79.72	60.38/82.41
<b>Test</b>	44.56/78.44	63.45/89.71

## 4 Data Used in Our Approach

The data used for the experiments is part of a larger Hindi Dependency Treebank. The size of training, development and testing data is 47k, 5k and 7k respectively. The training data used was manually validated and hence we assume it to be free of any errors. Hence the patterns learnt by the dependency parser will generally be consistent, thereby improving parser accuracy.

## 5 Comparison with Other Techniques

Here we compare the results of our strategy with one of the earlier tools [2]. They detected errors in Hindi Dependency Treebank using a hybrid approach, as described earlier in Section 2. As shown in Table 3, our approach significantly outperforms [2] both in terms of precision and recall. Also, a detailed comparison between their approach and our approach is shown in Table 2. There is an overlap of 73.46% of the total errors between our approach and [2].

As is visible from Table 2, the earlier approach [2] flagged a large number of nodes as error nodes (1724 nodes were flagged as error nodes while actual errors were 490) out of which 395 errors were correct. So, the precision of this approach was little low. Our approach flagged 672 errors, out of which 434 were correct. This leads to high precision and recall. Correct errors common to both the approaches were 360. Further, both these approaches when used together were able to detect a total of 483 out of 490 errors. It might be a good idea to consider errors flagged by both approaches to cover almost all the errors in the Treebank. Also, [2] tried to capture only labelling errors while we capture attachment errors too, which are difficult to capture as compared to labelling errors.

Our approach outperformed [2] though it was a hybrid approach, because they mostly concentrated on capturing as much errors as possible while we have tried to maintain balance between both precision and recall. However, it is very

likely that the approach by [2] might not have performed well because of lack of a good size of training data and their rule based approach to increase precision could not cover all concepts of the language.

**Table 2.** Comparison of present approach with [2]

Total Original Errors	490
Total Errors Flagged by [2]	1724
Total Errors Flagged by our approach	672
Total Common Errors Flagged by our approach and [2]	522
[2] Errors Correct	395
our approach Errors Correct	434
Overlap between [2] and our approach	360
Total(Both methods) Flagged Errors	2959
Total(Both methods) Correct Errors	483

**Table 3.** Comparison of overall (attachment+labeling) results of our approach with previous one [2]

Approach	Precision(%)	Recall(%)
<b>Our approach</b>	<b>64.58</b>	<b>88.57</b>
[2]	23.24	80.61

## 6 Discussion

In this paper, we presented an automatic approach to detect potential errors in an annotated treebank to provide aid in its validation task. Precision and recall of the approach are *64.58% and 88.57%* respectively. Unlike the previous approach reported in [2], our approach is language independent. Given a manually validated treebank, all one needs is to prepare a knowledge-base using parser error patterns. We expect, it will be relatively easy to adopt our approach for similar dependency treebanks.

Certain types of errors cannot be detected by our method. For categories where the parser is less confident on taking correct decision, a node was flagged as a potential error only when the annotators decision matched the parsers decision. There are chances that for a class of arcs, both, the parser and the annotators take different decisions but both the decisions are incorrect. In such a case, since the annotator’s output doesn’t match the parser’s output, we do not flag it as an error even though it is an erroneous node.

The power of our validation approach is bounded by the performance of the parser used. If we have better parsers, we can flag potential errors with more

confidence and accuracy. Also, to decide the confidence level for various classes of arcs, recall was used as a metric but there was a trade-off between precision and recall. Hence, if we use F-score as a metric in place of recall, we may achieve better results.

## 7 Acknowledgement

We would like to thank Himani Chaudhry and Riyaz Ahmad Bhat for their valuable comments which helped us to improve the quality of the paper.

The work reported in this paper is supported by the NSF grant (Award Number: CNS 0751202; CFDA Number: 47.070). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

1. Abney, S.: Parsing by Chunks. *Principle-based parsing* 44, 257–278 (1991)
2. Agarwal, R., Ambati, B., Sharma, D.: A Hybrid Approach to Error Detection in a Treebank and its Impact on Manual Validation Time. *Linguistic Issues in Language Technology* 7(1) (2012)
3. Ambati, B.R., Gupta, M., Husain, S., Sharma, D.M.: A High Recall Error Identification Tool for Hindi Treebank Validation. In: *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*. European Language Resources Association (ELRA), Valletta, Malta (may 2010)
4. Ambati, B.R., Husain, S., Nivre, J., Sangal, R.: On the Role of Morphosyntactic Features in Hindi Dependency Parsing. In: *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*. pp. 94–102. SPMRL '10, Association for Computational Linguistics, Stroudsburg, PA, USA (2010)
5. Ambati, B., Agarwal, R., Gupta, M., Husain, S., Sharma, D.: Error Detection for Treebank Validation. *The 9th International workshop on Asian Language Resources (ALR)*. Chiang Mai, Thailand (2011)
6. Begum, R., Husain, S., Dhvaj, A., Misra, D., Bai, L., Sangal, R.: Dependency Annotation Scheme for Indian Languages (2008)
7. Bharati, A., Chaitanya, V., Sangal, R., Ramakrishnamacharyulu, K.: *Natural Language Processing: A Paninian Perspective*. Prentice-Hall of India (1995)
8. Bhatt, R., Narasimhan, B., Palmer, M., Rambow, O., Sharma, D.M., Xia, F.: A Multi-Representational and Multi-Layered Treebank for Hindi/Urdu. In: *Proceedings of the Third Linguistic Annotation Workshop*. pp. 186–189. ACL-IJCNLP '09, Association for Computational Linguistics, Stroudsburg, PA, USA (2009)
9. Eskin, E.: Automatic Corpus Correction with Anomaly Detection. In: *North American Chapter of the Association for Computational Linguistics* (2000)
10. van Halteren, H.: The Detection of Inconsistency in Manually Tagged Text. In: *Proceedings of LINC-00*. Luxembourg (2000)
11. Husain, S., Agrawal, B.: Analyzing Parser Errors to Improve Parsing Accuracy and to Inform Treebanking Decisions. *Linguistic Issues in Language Technology* 7(1) (2012)

12. Kaljurand, K.: Checking Treebank Consistency to Find Annotation Errors (2004)
13. de Kok, D., Ma, J., van Noord, G.: A Generalized Method for Iterative Error Mining in Parsing Results. In: Proceedings of the 2009 Workshop on Grammar Engineering Across Frameworks. pp. 71–79. GEAF '09, Association for Computational Linguistics, Stroudsburg, PA, USA (2009)
14. Kordoni, V.: Strategies for Annotation of Large Corpora of Multilingual Spontaneous Speech Data. In: The workshop on Multilingual Corpora: Linguistic Requirements and Technical Perspectives held at Corpus Linguistics. Citeseer (2003)
15. Nivre, J.: Incrementality in Deterministic Dependency Parsing. In: Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together. pp. 50–57. Association for Computational Linguistics (2004)
16. Nivre, J., Hall, J.: Maltparser: A Language-Independent System for Data-Driven Dependency Parsing. In: In Proc. of the Fourth Workshop on Treebanks and Linguistic Theories. pp. 13–95 (2005)
17. van Noord, G.: Error Mining for Wide-Coverage Grammar Engineering. In: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics. ACL '04, Association for Computational Linguistics, Stroudsburg, PA, USA (2004)
18. Volokh, A., Neumann, G.: Automatic Detection and Correction of Errors in Dependency Tree-Banks. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers - Volume 2. pp. 346–350. HLT '11, Association for Computational Linguistics, Stroudsburg, PA, USA (2011)
19. Xia, F., Rambow, O., Bhatt, R., Palmer, M., Sharma, D.: Towards a Multi-Representational Treebank. In: The 7th International Workshop on Treebanks and Linguistic Theories. Groningen, Netherlands. Citeseer (2009)