Parsing Hindi with MDParser

Alexander Volokh and Günter Neumann
DFKI, Stuhlsatzenhausweg 3, Campus D3_2, 66123 Saarbrücken

alexander.volokh@dfki.de, neumann@dfki.de

ABSTRACT

We describe our participation in the MTPIL Hindi Parsing Shared Task-2012. Our system achieved the following results: 82.44% LAS/90.91% UAS (auto) and 85.31% LAS/92.88% UAS (gold). Our parser is based on the linear classification, which is suboptimal as far as the accuracy is concerned. The strong point of our approach is its speed. For parsing development the system requires 0.935 seconds, which corresponds to a parsing speed of 1318 sentences per second. The Hindi Treebank contains much less different part of speech tags than many other treebanks and therefore it was absolutely necessary to use the additional morphosyntactic features available in the treebank. We were able to build classifiers predicting those, using only the standard word form and part of speech features, with a high accuracy.

KEYWORDS : Dependency Parsing, Hindi Dependency Treebank, Linear Classification

1 Introduction

In this paper we describe our participation in the MTPIL Hindi Parsing Shared Task-2012. We have participated in both *auto* and *gold* tracks and achieved the following results: 82.44% LAS/90.91% UAS (auto) and 85.31% LAS/92.88% UAS (gold). In the *gold track*, the input contains tokens with gold standard morphological analysis, part-of-speech tags, chunks and some additional features. In the *auto* track, the input contains tokens only with the part-of-speech tags from an automatic tagger. For both track the requirement was that the same approach/system is used.

The Hindi dependency treebank, which was provided to the participants, was of an average size: the training data contained 12041 sentences (268,093 words) and the development data had 1233 sentences (26416 words). However, the training data contained 91 different dependency edge labels, which is much more than many other treebanks, e.g. English has 56 or German has 46. At the same time Hindi has 34 different parts of speech, whereas as other treebanks usually have more: e.g. English – 48 and German – 56. Therefore, the additional features (morphological, chunk and sentence-level features) available in the treebank are of a very significant importance, since the dependency edges are harder to predict, because there are so many types and the information available, i.e. parts of speech, are not so diverse and thus less discriminative.

In this paper we will describe the system MDParser used for the participation. We will therefore provide details on the parsing and learning approaches used in the system, as well as discuss the features, especially the additional ones, that we have integrated into our models. We think it is also important to point out that we have no knowledge of Hindi, we were not able to read the script or analyse the quality of our output. Therefore that was more or less a blind unmodified application of our parser, which was primarily designed for English and German, to Hindi.

Proceedings of the Workshop on Machine Translation and Parsing in Indian Languages (MTPIL-2012), pages 149–154,
COLING 2012, Mumbai, December 2012.

2 Parsing Approach

Almost all dependency parsers can be roughly split into two groups: graph-based and transition-based. The graph-based parsers assign scores to all possible dependency edges and then search for the highest-scoring dependency graph. Transition-based systems start at some initial configuration and perform a sequence of transitions to some final configuration, such that the desired dependency graph is derived in the process. For languages like English and German both approaches seem to be quite similar as far as accuracies are concerned, however, the speed of transition-based systems is higher (Volokh and Neumann, 2012). Therefore we have developed a transition-based parser.

There are a lot of different transition-based algorithms, which are able to perform the task, and they differ in many properties. Some of the most important properties are the efficiency, complexity, projectivity, determinism and incrementality. We use an algorithm based on the Covington's parsing strategy (Covington, 2000), which to our mind has particularly appealing properties. It is deterministic, i.e. its decisions during the processing are always final and are never revised. It is incremental, i.e. there is no need for the whole input to be read in prior to the computation and only a small look-ahead is necessary for computations, on the contrary to some other approaches, which consider the whole sentence, when computing the solution. The projectivity can be allowed or disallowed by changing a single parameter and no additional solutions are necessary, as it is the case with some other algorithms, which are able to process projective structures only (e.g. pseudo-projective parsing (Nivre et al., 2005). In case of the Hindi treebank, which contained a significant amount of non-projective edges, we have allowed non-projectivity. The complexity of Covington's parsing strategy is $O(n^2)$, which is worse than of some other algorithms in the field, which have linear complexity, e.g. Nivre's arc-standard and arc-eager algorithms (Nivre, 2006). However, the efficiency of the Covington's algorithm is still higher, because the worst-case complexity occurs rarely and Covington's parsing strategy allows an extremely efficient feature extraction (Volokh and Neumann, 2012).

Since transition-based systems deliver their result after a series of transitions, it is important to know what the inventory of possible transition types is. According to this approach in every transition the system has to decide whether for a pair of words under consideration there is a dependency relation or not. This usually corresponds to two possible transitions in case there is a dependency relation, namely one when the *left* word is the head of the *right* word (we will this transition right-arc) and another one when the *right* word is the head of the *left* word (we will call this transition left-arc). In case there is no dependency relation there are usually also two possible transitions: one changes the *left* word to some next word and the other one changes the *right* word to the next word. Thus, overall there are four basic transition types.

However, since every dependency relation has to be subsequently labelled with a dependency type, there are a lot of additional transition types, which are responsible for the labelling process. There are two fundamental ways how the labelling is done in transition-based systems. The first one is to treat the tasks of finding dependency edges and labelling them as separate tasks. Therefore several models are trained in this case. The second one is to combine the tasks and use one model for both predicting the basic transitions and labelling the edges. Usually, many labels can be assigned only for left-arc or right-arc transitions, but not for both (e.g. suffixes are always to the right of their head or the conjunction is always to the right of the first conjunct, whereas the title is always to the left etc.). When labelling edges is an independent process, this kind of

information is lost, unless there are two models, one for labelling edges of each direction. However, this means that there are overall three models and three classifiers, which makes the system much slower than when the second option is used and everything is done in one step. We have used the latter option in our system. This resulted in overall 99 possible transitions for our parsing algorithm.

3 Learning Approach

During the training phase a transition-based system uses the available gold standard in order to construct the correct dependency tree and at the same time it learns in which state which transition is taken. In the application phase, when the gold standard is not available, the system uses this learned model in order to guide the algorithm during the computation of the result.

There are countless classification approaches, which can be applied to this task. The most important properties of classifiers used for dependency parsing are: a) whether they are binary or support real multi-class classification, i.e. do not simply apply a series of binary classifiers for solving a multi-class task and b) whether they are linear or non-linear.

Most classification approaches support only binary classification and multi-class classification is solved by constructing a complex classifier with one-vs-all or one-vs-one strategies. As already mentioned, dependency parsing not only is a multi-class classification task, but also has a very big number of classes, e.g. 99 in case of this shared task. Constructing a multi-class classifier out of binary classifiers is tedious and the application is slow. Therefore, it is better to use a real multi-class classification approach.

A linear classifier identifies the class by a linear combination of all features in a feature vector, which does not require a lot of computation. One can visualise its operation as dividing one class from another by drawing a line between them. Of course this assumes that such line can be drawn, i.e. that the data is linearly separable. Usually this is not the case. A non-linear classifier often makes use of the method called kernel trick (Aizerman et al., 1964). According to this method a linear classifier solves a non-linear problem by mapping the original observations into a higher-dimensional space, where the linear classifier is subsequently used. The mapping is achieved by applying a kernel function to the feature space. Wherever a dot product is used, it is replaced with the kernel function. This is much more expensive, but guarantees better separability.

Considering these properties we have chosen a classification approach, which satisfies our needs most. The learning strategy MCSVM_SC (linear multi-class support vector machines (Keerthi et al., 2008) from the package LibLinear (Lin et al., 2008) is to our mind particularly suitable for dependency parsing. It is particularly fast because it is a linear classifier, which supports real multi-class classification.

4 Features

Feature models are a major factor for both accuracy and efficiency of a parser. The more features are present in the training data the better the chance that the learner will find good discriminative features necessary for accurate predictions. However, the bigger the number of features the more intensive is the training and the slower the processing. The ideal scenario is thus that the training data should contain only a relatively small amount of very good features.

For many treebanks, e.g. English or German, word form and POS features are usually sufficient in order to achieve high accuracies. Hindi is different in this respect. The amount of different POS tags is much smaller and therefore a good system also has to make use of morphological and other features in order to compensate for that. The different auto and gold tracks visualise the gap very well: the performance of systems which have the additional features available is significantly higher.

Therefore it was obvious that we want to use these features. In the gold track it was straightforward, since the corresponding features are available. For the auto track we have constructed classifiers, which are able to predict the following features: pers, cat, num, case, gen, tam, vib, stype and voicetype. The classifiers used the word form and POS information about the token, for which the features were predicted, as well as the same information for three words before and after this token. The features usually could be predicted with an accuracy of 97-98% for all of the above-mentioned functions.

As far as the model for parser is concerned here is the complete list of features we have used (except for the already mentioned pers, cat, num, case, gen, tam,vib, stype and voicetype features):

1. wfj → returns the word form of the token j; 2. pj → returns the part of speech of the token j; 3. wfjp1 → returns the word form of the token j+1; 4. pjp1 → returns the part of speech of the token j+1; 5. wfjp2 → returns the word form of the token j+2; 6. pjp2 → returns the part of speech of the token j+2; 7. wfjp3 → returns the word form of the token j+3; 8. pjp3 → returns the part of speech of the token j+3; 9. wfi → returns the word form of the token i; 10. pi → returns the part of speech of the token i; 11. pip1 → returns the part of speech of the token i+1; 12. wfhi → returns the word form of the head of the token i; 13. phi → returns the part of speech of the head of the token i; 14. depi → returns the dependency label of the head of the token i; 15. depldi → returns the dependency label of the left-most dependent of the token i; 16. deprdi → returns the dependency label of the right-most dependent of the token i; 17. depldj → eturns the dependency label of the left-most dependent of the token j; 18. dist → returns the distance between the tokens j and i. For i=0 the feature returns 0, for the distance 1 the feature returns 1, for distances 2 or 3 the feature returns 2, for distances 4 or 5 the value 3 is returned, for distances 6, 7, 8 or 9 the value 4 and for all other distances the value 5 is returned; 19. merge2(pi,pip1) → returns the concatenation of pi and pip1 features. 20. merge2(wfi,pi) → returns the concatenation of wfi and pi features; 21. merge3(pjp1,pjp2,pjp3) → returns the concatenation of pjp1, pjp2 and pjp3 features; 22. merge2(depldj,pj) → returns the concatenation of depldj and pj features; 23. merge3(pi,deprdi,depldi) → returns the concatenation of pi, deprdi and depldi features; 24. merge2(depi,wfhi) → returns the concatenation of depi and wfhi features; 25. merge3(phi,pjp1,pip1) → returns the concatenation of phi, pjp1 and pip1 features; 26. merge3(wfj,wfi,pjp3) → returns the concatenation of wfj, wfi and pjp3 features, 27. merge3(dist,pj,wfjp1) → returns the concatenation of dist, pj and pjp1 features.

Indexes j and i refer to the right and left words, respectively, examined in each state of the Covington's algorithm.

5 Performance

As we have already mentioned our system has achieved the following results in the tracks: 82.44% LAS/90.91% UAS (auto) and 85.31% LAS/92.88% UAS (gold).

These results are quite worse than what the top systems were able to achieve (up to 93.99% UAS/87.84% LAS gold; 90.83% LAS/96.37% UAS auto). However, one should keep in mind that our parser is based on the linear classification, which is suboptimal as far as the accuracy is concerned. The strong point of our approach is its speed. For parsing development the system requires 0.935 seconds, which corresponds to a parsing speed of 1318 sentences per second. We have a used a machine with a dual-core 2.4 GHz processor for computing. The system supports multithreading and therefore both cores could be used. The speed with one thread only is 1.648 seconds.

Furthermore, probably the lack of knowledge of the language also negatively affected the result, because we did not understand the meaning of many morphosyntactic features and labels, and thus a better performance probably could have been achieved with our approach if applied properly.

6 Conclusion

We have applied our very fast parser MDParser to the Hindi Treebank. We were able to achieve a competitive result and a very good parsing speed. The MTPIL Hindi Parsing Shared Task-2012 was a great opportunity for us test our system in a completely new scenario and demonstrate that it is truly multilingual, since we have had absolutely no knowledge of Hindi. Furthermore, it was the first time when we had to run the system in the non-projective mode, because the languages with which we worked before, did not contain enough non-projective edges and it has never been worth it to increase the search space in order to capture them so far. It was also the first time we have worked with a language where, beyond the usual word form and POS features, the morphosyntactic information was necessary in order to achieve good results. We were able to automatically predict those additional features using the standard word form and POS features with a very good accuracy.

Acknowledgments

The work presented here was partially supported by a research grant from the German Federal Ministry of Education and Research (BMBF) to the DFKI project Deependance (FKZ. 01IW11003).

References

A. Aizerman, E. M. Braverma and L. I. Rozoner, 1964. *Theoretical foundations of the potential function method in pattern recognition learning*. Automation and Remote Control vol. 25, pp. 821—837.

Michael A. Covington, 2000. *A Fundamental Algorithm for Dependency Parsing*. In Proceedings of the 39th Annual ACM Southeast Conference.

C.-J. Lin, R.-E. Fan, K.-W. Chang, C.-J. Hsieh and X.-R. Wang. *LIBLINEAR: A library for large linear classification*. Journal of Machine Learning Research 9(2008), pp. 1871-1874.

Joakim Nivre and Jens Nilsson, 2005. *Pseudo-projective dependency parsing*. Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics 2005. pp. 99—106.

Nivre, J, 2006. *Inductive Dependency Parsing* (Text, Speech and Language Technology). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

S. Sathiya Keerthi,S. Sundararajan, Kai-Wei, Chang, Hsieh, Cho-Jui, Lin and Chih-Jen, 2008. A *sequential dual method for large scale multi-class linear SVMs*. Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 408—416.

Alexander Volokh and Günter Neumann, 2012. *Dependency Parsing with Efficient Feature Extraction*. KI 2012. pp. 253-256.