The Application of Constraint Rules to Data-driven Parsing

Sardar Jaf
The University of Manchester
jafs@cs.man.ac.uk

Allan Ramsay
The University of Manchester
ramsaya@cs.man.ac.uk

Abstract

In this paper, we show an approach to extracting different types of constraint rules from a dependency treebank. Also, we show an approach to integrating these constraint rules into a dependency data-driven parser, where these constraint rules inform parsing decisions in specific situations where a set of parsing rule (which is induced from a classifier) may recommend several recommendations to the parser. Our experiments have shown that parsing accuracy could be improved by using different sets of constraint rules in combination with a set of parsing rules. Our parser is based on the arc-standard algorithm of MaltParser but with a number of extensions, which we will discuss in some detail.

1 Introduction

In this paper we present a new implementation of the arc-standard algorithm of MaltParser (Joakim, 2003; Nivre, 2006; Nivre, 2008). The key features of this implementation are that (i) it includes a new approach to handling non-projective trees (Section 3); (ii) it allows the inclusion of information about local subtrees as an extra guide to parsing (Section 8); (iii) the assignment of labels to arcs is carried out as a separate phase of analysis rather than during the determination of dependency relations between words (Section 5). We compare the performance of the arc-standard version of MaltParser with four different versions of our parser in Section 9.

2 Deterministic Shift-reduce Parsing

The arc-standard algorithm deterministically generates dependency trees using two data-structures: a queue of input words, and a stack of items that have been looked at by the parser. Three parse actions are applied to the queue and the stack: SHIFT, LEFT-ARC and RIGHT-ARC. SHIFT moves the head of the queue onto the top of the stack, LEFT-ARC makes the head of the queue a parent of the topmost item on the stack and pops this item from the stack, and RIGHT-ARC makes the topmost item on the stack a parent of the head of the queue, removing the head of the queue and moving the topmost item on the stack back to the queue. At each parse transition the parser uses a classifier trained on a dependency treebank for predicting the next parse action given the current state of the parser.

3 Non-projective Parsing

The arc-standard version of the MaltParser fails to deal with non-projective trees.

Figure 1 shows a well-known example of a Czech sentence with a non-projective dependency tree. Figure 2 shows the problem with the basic algorithm. In step *8* from Figure 2 the parser may perform either LEFT-ARC, RIGHT-ARC, or SHIFT, but none of these operations lead to producing a tree matching the original non-projective tree. According to the dependency relations that are extracted from the tree (as shown at the top of Figure 2), LEFT-ARC is not allowed. On the one hand, if the parser performs LEFT-ARC then this will lead to the production of a tree that will not match the original tree because that will make *5* the parent of *3*, which does not match any relations in the original tree. On the other hand, performing RIGHT-ARC, which is allowed , will make *3* the parent of *5*. However, performing RIGHT-ARC at this stage is not an ideal operation because *5* will not be available in subsequent stages when it is required to become the parent of *1*, which remains on the queue[1]. This means that *1* will subse-

[1]LEFT-ARC and RIGHT-ARC remove the dependent

Proceedings of Recent Advances in Natural Language Processing, pages 232–238,
Hissar, Bulgaria, Sep 7–9 2015.

quently receive the wrong parent, which will produce a tree that does not match the original tree. SHIFT will move *5* to the top of the stack, which means that both *5* and *3* will be on the stack and hence they will never be in a state where *3* can become the parent of *5*, therefore the parser will not produce a tree that matches the original tree.

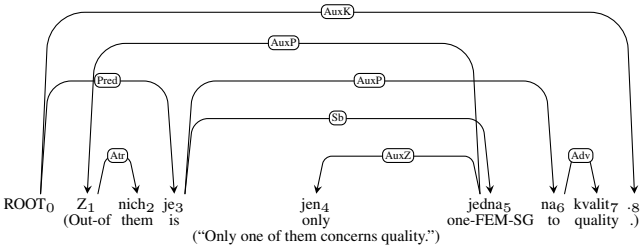

Figure 1: Non-projective dependency graph for a Czech sentence from the Prague Dependency Treebank (Nivre, 2008).

Dependency relations: (0,Pred,3)(0,AuxK,8)(1,Attr,2)(3,sb,5)
(3,AuxP,6)(5,AuxP,1)(5,AuxZ,4)(6,Adv,7)

Step	Action	Queue	Stack	Arcs
1	θ	[0,1,...]	[]	θ
2	SHIFT	[1,2,...]	[0]	θ
3	SHIFT	[2,3,...]	[1,0]	θ
4	RIGHT-ARC	[1,3,...]	[0]	A1=(1,Atr,2)
5	SHIFT	[3,4,...]	[1,0]	A1
6	SHIFT	[4,5,...]	[3,1,0]	A1
7	LEFT-ARC	[5,6,...]	[4,3,1,0]	A2=A1∪(5,AuxZ,4)
8	_	[5,6,...]	[3,1,0]	A2

Figure 2: Parsing the sentence in Figure 1 using the original arc-standard algorithm.

In order to overcome the limitation of the arc-standard algorithm of MaltParser, we allow for combining the head of the queue with an item on the stack that may or may not be the topmost item. Here, we introduce LEFT-ARC(N) and RIGHT-ARC(N) where N is any non-zero integer: LEFT-ARC(N) says 'Make the head of the queue the parent of the Nth item on the stack and pop the item from the stack', RIGHT-ARC(N) says 'Make the head of the queue a daughter of the Nth item on the stack, and roll the stack back onto the queue until you reach the Nth item'. LEFT-ARC(1) and RIGHT-ARC(1) are the arc-standard LEFT-ARC and RIGHT-ARC operations.

As part of this implementation we can reproduce the non-projective graph shown in Figure 1 given the dependency relations extracted from the graph. The parse transitions of the extended algorithm, as shown in Figure 3, reproduce the non-projective graph shown in Figure 1. The line in bold in steps 9 from Figure 3 shows the parse transition that the original algorithm would not have performed. In step 9, the extended algorithm performs the LEFT-ARC(2) operation. It makes the head of the queue (5) the parent of the second item on the stack (1)[2].

Dependency relations: (0,Pred,3)(0,AuxK,8)(1,Attr,2)(3,sb,5)
(3,AuxP,6)(5,AuxP,1)(5,AuxZ,4)(6,Adv,7)

Step	Action	Queue	Stack	Arcs
1	θ	[0,1,...]	[]	θ
2	SHIFT	[1,2,...]	[0]	θ
3	SHIFT	[2,3,...]	[1,0]	θ
4	RIGHT-ARC(1)	[1,3,...]	[0]	A1=(1,Atr,2)
5	SHIFT	[3,4,...]	[1,0]	A1
6	SHIFT	[4,5,...]	[3,1,0]	A1
7	SHIFT	[5,6,...]	[4,3,1,0]	A1
8	LEFT-ARC(1)	[5,6,...]	[3,1,0]	A2=A1∪(5,AuxZ,4)
9	**LEFT-ARC(2)**	**[5,6,...]**	**[3,0]**	**A3=A2∪(5,AuxP,1)**
10	RIGHT-ARC(1)	[3,6,...]	[0]	A4=A3∪(3,Sb,5)
11	SHIFT	[6,7,8]	[3,0]	A4
12	SHIFT	[7,8]	[6,3,0]	A4
13	RIGHT-ARC(1)	[6,8]	[3,0]	A5=A4∪(6,Adv,7)
14	RIGHT-ARC(1)	[3,8]	[0]	A6=A5∪(3,AuxP,6)
15	RIGHT-ARC(1)	[0,8]	[]	A7=A6∪(0,Pred,3)
16	SHIFT	[8]	[0]	A7
17	RIGHT-ARC(1)	[0]	[]	A8=A7∪(0,AuxK,8)
18	SHIFT	[]	[0]	A8
19	θ	[]	[0]	A8

Figure 3: Parsing the sentence in Figure 1 using the extended version of the arc-standard algorithm.

A similar technique for processing non-projective sentences is proposed by Kuhlmann and Nivre (2010), which is the non-adjacent arc transitions. This technique allows for creating arcs between non-neighbouring arcs. This is achieved by extending the arc-standard to do the followings:

LEFT-ARC-2$_l$: This operation creates an arc by making the topmost item on the stack the parent of the third topmost item on the stack, and removes the topmost item.

RIGHT-ARC(2)$_l$: This operation creates an arc by making the third topmost item on the stack the parent of the topmost item on the stack, and removes the topmost item.

Although Attardi (2006) claims that LEFT-ARC(2)$_l$ and RIGHT-ARC-2$_l$ are sufficient for producing every non-projective tree Kuhlmann and Nivre (2010, p. 6) argues to the contrary.

Our re-implementation of the arc-standard algorithm, which is a generalisation of propos-

item from the queue or the stack so they will not be available in subsequent steps.

[2]The head of the queue was not combined with the topmost item on the stack in step 9 because that would have removed 5 from the queue, which will be needed later to be used as the parent of 1.

als by Kuhlmann and Nivre (2010) and Attardi (2006), will handle all possible cases of non-projectivity because we allow N in LEFT-ARC(N) and RIGHT-ARC(N) to be a positive number larger than 2 if necessary. However, it contrasts the approach used by Kuhlmann and Nivre (2010) in that we combine the head of the queue with any item on the stack rather than combining the top items on the stack. Unfortunately, our approach broadens the range of possibilities available to the parser at each stage of the parsing process, and hence learning parse rules for enabling the parser to make the right choice at each stage becomes more difficult.

4 Assigning Scores to Parse States

Our parser generates one or more parse states from a given state. If the queue consists of one or more items and the stack is empty then the parser produces one state by performing SHIFT. For example, if the queue is `[1, 2, 3, 4]` and the stack is `[]` then the parser cannot recommend LEFT-ARC(N) or RIGHT-ARC(N) because these two operations require an item on the stack to be made the parent or the daughter of the head of the queue respectively.

If the queue consists of one or more items and the stack consists of one item only, then there are three possible moves: SHIFT, LEFT-ARC(1), and RIGHT-ARC(1). However, the parse model, which is based on a classification algorithm, will recommend only one operation (SHIFT, LEFT-ARC(1), or RIGHT-ARC(1)). Although in this kind of state our parser generates three states only one state will be given a positive score, which is based on recommendation of the parsing rules.

If the queue consists of one or more items and the stack consists of more than one item, then our parser may generate more than three states because it checks for relations between the head of the queue and any items on the stack; i.e., states that are generated by LEFT-ARC(N+1) and RIGHT-ARC(N+1), where N is a positive number.

In order to use a state from the newly generated states we assign a score to each new state, which is computed by using two different scores: (i) a score that is based on the recommendation made by the parsing rules. For example, we give a score of 1 for a SHIFT operation if it is recommended by a parsing rule, otherwise we give it a score of 0 (and the same applies to LEFT-ARC(N) and RIGHT-ARC(N)). Also (ii) we add the score from (i) to the score of the current state (which is the state that the new parse state is generated from). The sum of these two scores is assigned to the newly generated parse state(s).

There are two advantages of assigning a score to each parse state: (i) we can manipulate the assignment of various other scores to newly generated parse state(s), such as scores for the application of constraint rules to parse states, and (ii) we can rank a collection of parse states by using their scores and then process the state with the highest score, which we consider the most plausible state.

We store the states with various scores in an agenda ranked based on their scores, and the state with the highest score is explored by the parser.

5 Labelled Attachment Score

In this section we show the way we obtain labelled attachment scores, which is largely different from the way this is implemented in the original algorithm. As in the arc-standard algorithm, for each dependency relation between two words, a syntactic label is attached to indicate the syntactic role of the daughter item with its parent. However, the way we assign labels to dependency relations during parsing is that we extract patterns from the training data during training phase. This contrasts with the approach used in MaltParser whereby labels are predicted with the LEFT-ARC and RIGHT-ARC actions of the parser which are learned during training phase.

Each pattern or rule consists of a dependency parent, a list of n part-of-speech (POS) tagged items, a dependency daughter, a label, and the frequency of the pattern in the training data. A schema of a pattern is shown in Figure 4. The first element of the pattern is a parent item, the second element is a list of up to n POS tagged items between a parent item and its daughter in the original text, the third element is the daughter of a parent item, the fourth element is the label for the dependency relation and the last element is the frequency of the pattern recorded during the training phase. Figure 4 shows the rule format where `PARENT` is assigned as the parent of `DAUGHTER` and that there are up to n POS tagged items between them and the dependency label between the parent item and daughter item is `LABEL` where the last element indicates that the pattern occurred `j` times during the training phase.

PARENT,[POS_1,...,POS_n],DAUGHTER,LABEL,j

Figure 4: A schema of a pattern for a label.

6 Dataset

The kind of data that is suitable for developing a data-driven parser is an annotated treebank. There are a number of treebanks available for inducing a dependency parser for a number of natural languages. Some of the most popular treebanks for Arabic are: Penn Arabic Treebank (PATB) (Maamouri and Bies, 2004), Prague Arabic dependency treebank (PADT) (Smrž and Hajič, 2006), and Columbia Arabic treebank (CATiB) (Habash and Roth, 2009).

The linguistic information in PATB is sufficient for inducing a parser. However, the limitation for using this treebank directly for generating a parse model is that its annotation schemata is based on a phrase structure format, which cannot be used for dependency parsing. However, we have converted the phrase structure trees of the PATB to dependency structure trees using the standard conversion algorithm for transforming phrase structure trees to dependency trees, as described is detail by Xia and Palmer (2001).

Because we do not have access to the PATD and CATiB treebanks, we have used the PATB[3] part 1 version 3 for training and testing the arc-standard version of MaltParser and various versions of our parser.

In order to perform a 5-fold validation, we have systematically generated five sets of testing data and five sets of training data from the treebank, where the testing data is not part of the training data. The training data for each fold contains approximately 112,800 words while the testing data for each fold contains approximately 28,000 words. The average length of sentences is 29 words and the total number of testing sentences in each fold is about 970 sentences while the total number of sentences in the training data in each fold is about 3880 sentence. We use the training data for generating a set of parsing rules and for extracting a set of constraint rules; this way we are retrieving two different kinds of information from the training data.

[3]Catalogue number LDC2005T02 from the Linguistic Data Consortium (LDC). Available at: http://www.ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2005T02

7 The Role of Constraint Rules in Parsing

Each intermediate state that is produced by following recommended parse operations by the parse model is checked to see whether it is plausible. We consider a state to be plausible if it obeys the constraint rules.

A parse state is assigned a score based on the recommendation of the parse model (see Section 4 for more details). We attempt to use constraint rules to assign an additional score to a state if the recommended parse operation by the parsing rules does not violate the constraint rules. This means that recommendations made by the parsing rules are validated by using a set of constraint rules to check whether they produce acceptable analyses. This way the parser benefits from the information provided by the parsing rules and from the information provided by the constraint rules.

The role of the constraint rules is particularly evident when the parser produces more than three states from one state. In situations where the parser is presented with a state whereby the queue contains one or more items and the stack contains more than one item, then the parser generates more than three states because it checks for relations between the head of the queue and any items on the stack. In this kind of situation, two or more parse operations may be recommended by the parsing rules; i.e., two or more states may be given a positive score. To determine which of the equally scored states should be explored next, the score given by the constraint rules to a parse state will influence the parser's decision. For example the lines in bold from Figure 5 where we assumed that the parsing rules recommended LEFT-ARC(1) (making 3 the parent of 2) and also LEFT-ARC(2) (making 3 the parent of 1) they are both given a score of 1, as shown in bold in Figure 5. Also, we assumed that the constraint rules encouraged the recommendation of the parse model and that they gave their scores to the two recommended operations, where LEFT-ARC(1) is given 0.25 and LEFT-ARC(2) is given 0.5. In this situation, LEFT-ARC(2), with a total score of 1.5, plus the score for the currently explored state (In this example the current score is set to 1), will be placed on the top of the agenda because it will have the highest score (2.5). In a situation like this, the constraint rules influence the decision of the parser whereby LEFT-ARC(2) is performed

instead of LEFT-ARC(1).

States	Action	Queue	Stack	arcs	Curr. Sc	Sc	C. Sc	T. Sc
Current	θ	[3,4]	[2,1]	θ	1	θ	θ	1
New	SHIFT	[4]	[3,2,1]	θ	1	0	0	1
	RA(1)	[2,4]	[1]	2>3	1	0	0	1
	RA(2)	[1,2,4]	[]	1>3	1	0	0	1
	LA(1)	**[3,4]**	**[1]**	**3>2**	**1**	**1**	**0.25**	**2.25**
	LA(2)	**[2,3,4]**	**[]**	**3>1**	**1**	**1**	**0.5**	**2.5**

Figure 5: The generation of more than three states, LA = LEFT-ARC, RA = RIGHT-ARC, Curr = current, Sc = Score, C = Constraint, T = Total.

In Figure 5 we have shown the way the constraint rules may influence parse decisions. In the following sections, we describe different types of constraint rules that can be extracted automatically from a dependency treebank where we integrate them into our parser.

8 Extracting Constraint Rules from PATB

The main type of relations that are accounted for in dependency parsing are the parent-daughter relations between different words in a sentence. We devote the following sections to describing two different types constraint rules extracted from a set of dependency trees.

8.1 Parent-daughter Relations Extraction with Local Contextual Information

In the training phase, we use the dependency tree of each sentence as a grammar for parsing the sentence. During each LEFT-ARC(N) or RIGHT-ARC(N), the dependency relation between a parent and its daughter is recorded. The recorded relations contain different information: (i) the parent item (ii) the daughter item, (iii) a set of up to *n* POS tagged items from the queue and up to *n* POS tagged items from the stack[4], and (iv) the frequency of each rule. The frequency of each rule is used for computing the probability of the rule during parsing. The probability computation of a rule is calculated in three steps (i) obtaining the frequency of a rule, (ii) obtaining the sum of the frequency of all the rules with the same parent and daughter relation (regardless of the *n* POS tagged items that appear between them), (iii) dividing the number obtained in step (i) by the number obtained in step (ii). The probability of each rule is then used as a score for encouraging a parse operation suggested by the parse model.

The conditional probability for the constraint rule in Figure 6 is shown in equation (1), where r_i is a distinct rule with the same parent and daughter but a different set of intermediate items.

$$P(r_j) = \frac{|r_j|}{\sum_{i=1}^{n} |r_i|} \quad (1)$$

In Figure 6 we show an example of a constraint rule with a window size of up to two items on the queue and up to two items on the stack. The rule in Figure 6 shows that a `VERB` is the parent of a `NOUN` if the first item in the queue is a `VERB`, the second item in the queue is a `PREP`, and there is only one item on the stack which is a `NOUN`. Since there is no second item on the stack the symbol `'-'` is used for representing unavailable items. The final element (`j`) of the rule represents the frequency of the rule during training.

```
r = (VERB,NOUN,[VERB,PREP,NOUN,-],j)
```

Figure 6: Dependency relations with local information.

We have evaluated our parser using this type of constraint rules where the best parsing performance is achieved when we recorded four items from the queue and three items from the stack for each dependency relation. The parsing performance is shown in Table 1.

8.2 Subtrees

Since LEFT-ARC(N) and RIGHT-ARC(N) result in the removal of a daughter item from the stack or queue, which may be required in subsequent parsing stages, it is vital to ensure that the daughter has collected all and only its daughters. Thus, subtrees can be used to encourage the parser to remove a daughter item only if there is evidence that it has collected all and only its daughters, this corresponds to completeness and cohesion in Lexical Functional Grammar (LFG) (Bresnan and Kaplan, 1982). This check is performed in two steps by using the subtrees: (i) collecting all the daughters of the dependent item from the tree that have been built by the parser, and (ii) finding a subtree (from a set of subtrees collected during training phase) that is headed by the dependent item with the same set of daughters that are collected in (i).

[4]The number of items collected from the queue and the stack may vary between `1 ... n`.

If a matching subtree is found then the parse operation can be encouraged by giving it a score. As shown in Figure 7, each daughter in a subtree is associated with a score, which represents the frequency of the subtree during training. The score is used for computing the probability of the subtree with a specific set of daughters, which is computed by dividing the frequency of the subtree by the total associated frequencies of all other daughters headed by the same item, this process resembles the approach used by Charniak (1996). The computed probability is then used for encouraging the parse operation.

Figure 7 shows two subtrees headed by a `VERB` where the first one has a `NOUN` as its daughter and it occurred `5` times during training while in the second rule the `VERB` has two `NOUN`s as its daughter and it occurred `10` times during training.

```
r = VERB,(5,[NOUN])
r = VERB,(10,[NOUN,NOUN])
```

Figure 7: Examples of unlexicalised subtree

The conditional probability for the subtrees in Figure 7 is shown in equation (2) where each r_k^f is a distinct rule.

$$P(r_k) = \frac{r_k^f}{\sum_{i=1}^{n} r_i^f} \quad (2)$$

9 Evaluation

In this section we compare the result we have obtained for testing the arc-standard algorithm of MaltParser[5] with different versions of our re-implementation of this algorithm: (i) DDParser, which is our re-implementation of the arc-standard of MaltParser; (ii) CDDParser, which is DDParser supplemented by parent-daughter constraint rules, i.e., the parsing rules and a set of parent-daughter constrain rules are used during parsing, (iii) SDDParser, which is DDParser supplemented by local subtrees, i.e., the parsing rules and a set of subtrees are used during parsing, and (iv) S-CDDDParser, which is DDParser supplemented by a combination of subtrees and parent-daughter constrain rules. The performance of each parser is shown in Table 1.

We can note from Table 1 that DDParser is 43.8% more efficient than MaltParser. Although the unlabelled attachment score (UAS) of DDParser is slightly lower than that of MaltParser (0.7%) the labelled attachment score (LAS) and the labelled accuracy (LA) are more accurate than MaltParser by 1% and 1.4% respectively. We believe that this improved accuracy of LAS and LA occurred because we have used a different approach from MaltParser for assigning labels to dependency relations (see Section 5 for more details on our approach to label assignment).

Parsers	UAS (%)	LAS (%)	LA (%)	second/relation
MaltParser	75.2	70.0	92.2	0.144
DDParser	74.5	71.0	93.6	0.081
CDDParser	76.2	72.7	94.85	0.145
SDDParser	75.9	72.4	94.84	0.133
S-CD-DDParser	75.3	71.8	94.82	0.127

Table 1: Performance of MaltParser and our parsers.

The use of constraint rules has improved the parsing accuracy of DDParser but it has noticeably degraded its speed. This clearly indicates that the use of constraint rules improves parsing accuracy at the expense of speed. Having said that, the use of parent-daughter constraint rules improved the accuracy of our parser over the accuracy of MaltParser by 1% for UAS, 2.7% for LAS and 2.65% for LA while the parser remained as efficient as MaltParser.

The use of local subtrees as constraint rules also improved the accuracy of our parser over the accuracy of MaltParser by 0.7% for UAS, 2.4% for LAS and 2.64% for LA while its speed is quicker than MaltParser by 7.6%. These results show that the application of different types of constraint rules to a data-driven parser affects parsing performance differently. We have shown here that we can trade off parsing speed for parsing accuracy by using different constraint rules.

Additionally, we have combined the constraint rules and subtrees and applied them to DDParser. Applying both extensions to the parser did not lead to better results than using them individually. However, applying both extensions lead to better parsing accuracy than using none of them but the parsing speed degraded by about 36%.

It is worth noting that the training time of our parser, including the automatic extraction of constraint rules from the training data, was much shorter than the training time of the original algorithm. The training time for the original algorithm took approximately four hours. While the training time for our parser took approximately thirty

[5] Available at: http://www.maltparser.org/download.html

minutes. We assume that our training time was shorter because we have used the J48 classification algorithm (which is the Weka's[6] implementation of C4.5 (Quinlan, 1996)) instead of LiBSVM (Chang and Lin, 2011), which is used by the original algorithm[7].

In conclusion, from the experiments that we have conducted in this paper, we can note that applying constraint rules to a data-driven parser may improve the parsing accuracy but the parsing speed may degrade.

10 Future Work

Since there are a number of treebanks for different natural languages and that our method is language independent, we would like to evaluate our parser on different languages and examine its extendibility to other languages.

For this study, we have extracted a set of constraint rules from the same training data that we have used for generating a parse model. In the future, we would like to obtain a set of linguistic grammatical rules and apply them to our parser for validating operations recommended by the parse model.

11 Summary

In this paper we have shown an extension to the arc-standard algorithm of MaltParser. We have also shown a method to automatically extracting different kinds of constraint rules from a dependency treebank.

Our re-implementation of the arc-standard algorithm of MaltParser allows us to integrate different kinds of constraint rules to it. We have shown that the application of these constraint rules have improved the parsing accuracy at the expense of parsing speed. Although the application of constraint rules to parsing degraded the parsing speed the parser remained as efficient as the original algorithm.

Acknowledgements

This work was funded by the Qatar National Research Fund (grant NPRP 09-046-6-001).

[6] Available publicly at: http://www.cs.waikato.ac.nz/ml/weka/index.html

[7] We have experimented with a large number of classification algorithms with various features and settings for training our parser, but we cannot present them in this paper due to space limitation. See (Sardar, 2015) for more details on experiments on using different classifiers for parsing.

References

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL-X '06, pages 166–170, Stroudsburg, PA, USA. Association for Computational Linguistics.

Joan Bresnan and Ronald Kaplan. 1982. Lexical function grammar. In J.W. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–281, Cambridge, MA. MIT Press.

Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May.

Eugene Charniak. 1996. Tree-bank Grammars. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1031–1036.

Nizar Habash and Ryan M. Roth. 2009. Catib: The columbia arabic treebank. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, pages 221–224.

Nivre Joakim. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*, pages 149–160. Citeseer.

Marco Kuhlmann and Joakim Niver. 2010. Transition-based techniques for non-projective dependency parsing. *Northern European Journal of Language Technology*, 2:1–19.

Mohamed Maamouri and Ann Bies. 2004. Developing an Arabic treebank: methods, guidelines, procedures, and tools. In *Proceedings of the Workshop on Computational Approaches to Arabic Script-based Languages*, pages 2–9, Geneva.

Joakim Nivre. 2006. *Inductive dependency parsing*, volume 34 of *Text, Speech and Language Technology*. Springer.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:513–553.

John R. Quinlan. 1996. Improved use of continuous attributes in c4.5. *Journal of Artificial Intelligence Research*, 4:77–90.

Jaf Sardar. 2015. *The Application of Constraint Rules to Data-driven Parsing*. PhD Thesis, School of Computer Science, The University of Manchester.

Otakar Smrž and Jan Hajič. 2006. The other arabic treebank: Prague dependencies and functions. *Arabic Computational Linguistics: Current Implementations. CSLI Publications*, 104.

Fei Xia and Martha Palmer. 2001. Converting dependency structures to phrase structures. In *1st Human Language Technology Conference (HLT-2001)*, pages 1–5, San Diego.