The Selection of Classifiers for a Data-driven Parser

Sardar Jaf[1], Allan Ramsay[1]

[1]The University of Manchester, Faculty of Engineering and Physical Sciences, School of Computer Science, Manchester, United Kingdom
{sardar.jaf, allan.ramsay}@manchester.ac.uk

Abstract. There is a large number of classifiers that can be used for generating a parse model; i.e., as an oracle for guiding data-driven parsers when parsing natural languages. In this paper we present a general and simple approach for generating a parse model. Additionally, we present a large number of experiments on various classifiers. We also present the effect of various parse models, which are generated from different classifiers, on a data-driven parser to see the way each model contributes to parsing performance.

1 Introduction

The objective of this study is to present an approach for generating different parse models, which are used for guiding parsers during natural language parsing, from different machine learning classifiers. There are various classification algorithms that can be used for this purpose. However, different classifiers may learn from a set of data differently, which means that they may affect parsing performance in different ways. In Section 2 we present a data-driven parser that we have used for examining the effectiveness of different parse models, which are generated from different classifiers. In Section 3 we show a simple approach for generating a parse model from the J48 classifier while in Section 5 we show the accuracy of a large number of classifiers. Section 6 covers the effect of each parse model on parsing performance. Finally, in Section 7 we compare our parser with the arc-standard algorithm of MaltParser.

2 A Data-driven Shift-Reduce Parser

Our parser is based on the arc-standard algorithm of MaltParser (Kuhlmann and Nivre, 2010). This 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 stack: SHIFT, LEFT-ARC and RIGHT-ARC (we will write LA and RA for LEFT-ARC and RIGHT-ARC respectively to save space). SHIFT moves the head of the queue onto the top of the stack, LA makes the head of the queue a parent of the topmost item on the stack and pops this item from the stack, and RA makes the topmost item on the stack a parent of the head of the queue; RA removes the head of the queue and moves the topmost item on the stack back to the queue. MaltParser uses a support vector machine classifier for

generating a parse model from a set of parsed trees, which is used for predicting the next parse action given the current state of the parser.

We will call our parser NDParser. At each parse step, we generate a state for LA, RA, and SHIFT, and we will assign different scores to each state. For example, a score is computed for each newly generated state by computing two different scores: (i) a score that is based on the recommendation made by a parse model. For instance, when generating a SHIFT state the parser gives a score of 1 if a SHIFT operation is recommended by the model. Otherwise a score of 0 is given (and the same applies to LA and RA). (ii) The score of the state that the new state is derived from. The sum of these two scores is then assigned to the newly generated state. The advantage of assigning a score to a parse state is that we can rank a collection of parse states by using their scores and then process the state with the highest score. In order to efficiently process a potentially large set of states, we use dynamic programming for ranking competing states with respect to their plausibility (the plausibility of a state is based on its score.) The ranked states are then stored in a chart table (Kay, 1973) and the most plausible state is explored by the parser, where new states are generated by using SHIFT, LA, and LR operations. This way we combine features of chart parsing with shift-reduce parsing.

3 The Generation of a Parse Model

The effectiveness of a parse model largely depends on the classifier's ability to correctly classify a Treebank. The Penn Arabic Treebank (Maamouri and Bies, 2004), which we converted it to Dependency format, was used for parser training and testing. We have experimented with several classifiers that are available in the `WEKA' toolkit (Hall et al., 2009) for classifying a set of training data. The output of each classifier is then used for generating a parse model, which is then used for examining the effect of each model on parsing performance. The following steps explain a general and simple approach for generating a parse model from a dependency Treebank:

Step 1. Forced parsing: we use a shift-reduce parser for parsing the training data, which contains the parsed trees of each sentence. The parsed tree of each sentence are used as a grammar to parse the sentence, which is used as a guide to parse the sentence and record parse states during training.

Step 2. Collecting parse states: during training we obtain a set of parse states, i.e., state:action pairs where the condition is the state of the queue and stack (i.e., the items on the queue and stack) and the action is the parse operation that the parser performed (which is either SHIFT, LA, or RA). Consider, for instance, the parsed tree in Fig. 1 for the sentence '*the cat sat on the mat*'.

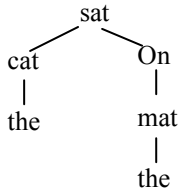

Fig. 1. Dependency tree for the sentence '*the cat sat on the mat'*

Fig. 2 shows the transitions that the parser uses for producing the tree for the sentence '*the cat sat on the mat'*. Note that whilst constructing the training data we will not perform any LA or RA operations if a dependency daughter is the head of an item that is not inspected yet, as illustrated from step 6 of Fig. 2 where we perform SHIFT instead of RA, since performing RA at this point would make it impossible to later make *on* the head of *mat* because RA would remove *on* from the queue, which would prevent it from becoming the head of *mat* which is still on the queue.

Dependency relations: (sat>cat) (sat>on) (cat>the) (on>mat) (mat>the)

Steps	Action	Queue	Stack	Arcs
1	-	[the,cat,sat,on,the,mat]	[]	-
2	SHIFT	[cat,sat,on,the,mat]	[the]	-
3	LA	[cat,sat,on,the,mat]	[]	A1=(cat>the)
4	SHIFT	[sat,on,the,mat]	[cat]	A1
5	LA	[sat,on,the,mat]	[]	A2=A1+(sat>cat)
6	**SHIFT**	**[on,the,mat]**	**[sat]**	**A2**
7	SHIFT	[the,mat]	[on,sat]	A2

Fig. 2. Parse states when parsing the sentence '*the cat sat on the mat*'

We can treat the sequences shown in Fig. 2 as a set of data-points which indicate what the parser should do in a given state -- for instance, in a situation like in step 6 in Fig. 2 the parser should use SHIFT instead of RA for the reason explained above.

Given a set of such data-points, it is possible to extract and record the parse states and train a classifier for building a parse model, which can be used for predicting parse operation; i.e., it can be used for guiding the parser. The task here is to classify intermediate states of the parser into three groups: cases where SHIFT should be performed, cases where LA should be performed, and cases where RA should be performed.

Step 3. Preparing recorded parse states for classification: from the set of parse states that we obtained in step 2, we populate an *.arff* file with the correct data format, i.e., the format that is accepted by WEKA. An example of a set of WEKA-style data format is shown in Fig. 3, which is based on the parse states shown in Fig. 2. Here we have extracted the word forms as a feature for learning but it is possible to use a number of different features (such as POS tags, word position etc.) as values for the queue and the stack attribute parameters.

```
@relation states
@attribute queue_word_pos_1 {'the', 'cat', 'sat', 'on', 'mat', '-' }
@attribute queue_word_pos_2 {'cat', 'sat', 'on', 'the', 'mat', '-'}
@attribute stack_word_pos_1 {'-','the', 'cat', 'sat', 'on'}
@attribute stack_word_pos_2 {'-', 'sat', 'on'}
@attribute parse_action {'SHIFT', 'LEFT-ARC', 'RIGHT-ARC'}
@data
```

```
‘the’, ‘cat’, ‘-’, ‘-’, ‘SHIFT’
‘cat’, ‘sat’, ‘the’, ‘-’, ‘LEFT-ARC’
‘cat’, ‘sat', ‘-’,‘-’’SHIFT ’
‘sat’, ‘on’, ‘cat', ‘-’, ‘LEFT-ARC’
 …
```

Fig. 3. An example of data for an *.arff* file

Additionally, one can use many different window sizes for the queue and the stack in the data selection as instances for the classification algorithms to learn from. In Fig. 3. we use a window size of two items for the queue and two items for the stack, while the dash mark (‘-’) represents an empty item where the queue or the stack did not contain an item in the given position.

Step 4. Training a classifier using the *.arff* file: we supply WEKA with the data prepared in step 3 (i.e., the *.arff* file) and then we select a classification algorithm for learning. Fig. 4. is an example of the J48 classification algorithm output from WEKA.

Step 5. Generating a parse model from the classification output: finally, we convert the output produced by the classification algorithm to an appropriate state-action model, which is used for guiding the parser to parse new sentences. Fig. 5. is a sample of some states and actions we have extracted from the J48 (Quinlan, 1992) classifier's output.

```
Satck_word_pos_1 = ?: SHIFT (8430.0)
Stack_word_pos_1 = ABBREV
|    queue_word_pos_1 = ABBREV
|    |    queue_word_pos_2 = ?: RIGHT-ARC (6.0)
|    |    queue_word_pos_2 = ABBREV
|    |    |    queue_word_pos_3 = ?: RIGHT-ARC (5.0)
|    |    |    queue_word_pos_3 = ABBREV: RIGHT-ARC (2.0)
…
```

Fig. 4. An example of the J48 algorithm output using WEKA

```
states(
QUEUE,
STACK, [
word_pos(STACK, 1, ‘-’), ‘SHIFT’,
word_pos(STACK, 1, ‘ABBREV’), [
word_pos(QUEUE, 1, ‘ABBREV’), [
word_pos(QUEUE, 2, ‘-’), ‘RIGHT-ARC’,
… ] ] ] ] ] ).
```

Fig. 5. An example of a state-action model

4 Label Assignment to Dependency Relations

In this section we show the way we assign labels to dependency relations, which is largely different from the way this is implemented in the standard implementation of MaltParser. As in the arc-standard algorithm, for each dependency relation between two words, a label is attached to indicate the grammatical function 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 the training phase. This contrasts with the approach used in MaltParser whereby labels are predicted with the LA and RA actions of the parser which are learned during the training phase.

Each pattern 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 Fig. 6. The first element of the pattern is a parent item, the second is a list of up to *n* POS tagged items between a parent item and its daughter in the original text, the third 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 training. Fig. 6. shows the pattern when PARENT is assigned as the parent of DAUGHTER where there are up to *n* POS tagged items between them then their dependency label is LABEL, and the last element indicates that the pattern occurred j times during training.

PARENT, [POS1,...,POS*n*], DAUGHTER, LABEL, j

Fig. 6. A schema of a pattern for a label

During the evaluation phrase, we show three different parsing accuracy measures, those are: (i) Labelled Attachment Scores (LAS), which is the percentage of the correct dependency relations with the correct labels of the dependency relations (DEPREL) between tokens; (ii) Unlabelled Attachment Score (UAS), which is the percentage of correct dependency relation (i.e., the percentage of tokens with correct heads) regardless of the DEPREL; and (iii) Labelled Scores (LS) which is the percentage of tokens with the correct dependency label.

5 Evaluating Different Classifiers

Table 1 contains the accuracy of various classifiers that were used for classifying the training data that we have mentioned in Step 2 of Section 3. We consider a classifier appropriate for producing a parse model if it meets two requirements: (i) it produces good classification accuracy. Although the accuracy of the classifiers that are presented in Table 1 may not directly reflect the accuracy of a parser that uses its recommendations but, a classifier that produces a high level of accuracy is more likely to assist a parser to make more informed parse decisions at each parse step than a classifier that produces a low level of accuracy; and (ii) its output can be used for generating a parse model which can be used for making recommendations to a data-

driven parser, for example, what action (SHIFT, LA, or RA) the parser should take in a specific situation. We have used various features for training different classification algorithms. These features included POS tags, word forms, word locations in sentences, their spans (i.e., their start and end positions in sentences). Additionally, we have used a combination of these features such as word forms with POS tags, word forms with word location or word spans, and similar combination of POS tags with other features. Also, various window sizes are used for the queue and stack, ranging between two items to four items. The use of these features for training each classifier along with the classification accuracy is presented in Table 1. Previous experiments by Jaf and Ramsay (2013) indicated that using a window size of more than four items on the queue or stack did not yield better results, hence we have used up to four items in this experiment.

Table 1. Classification accuracy with various feature and setting. *W* = Word, Loc = *Item location in sentence*, *POS = part-of-speech tags*, and *Span = start and end position of a word*

J48							
Items on Queue	2	3	3	3	4	4	4
Items on Stack	4	2	3	4	2	3	4
W (%)	68.24	68.29	68.37	68.53	68.56	68.67	68.81
W + Loc (%)	71.92	72.23	71.88	71.67	72.41	72.11	71.80
W + Loc + span (%)	71.73	72.76	72.25	72.00	72.87	72.45	72.17
W + span (%)	70.17	70.81	70.64	70.43	70.83	70.79	70.56
POS (%)	84.94	85.63	85.77	85.80	85.89	86.05	86.04
POS + Loc (%)	85.27	85.89	85.91	85.92	86.96	86.08	86.09
POS + Loc + span (%)	85.23	85.81	85.84	85.92	85.95	85.97	85.99
POS + span (%)	85.00	85.71	85.69	85.67	85.88	85.88	85.88
W + POS (%)	85.28	86.23	86.24	86.24	86.46	86.47	86.39
W + POS + Loc (%)	85.83	86.57	86.53	86.45	86.63	86.57	86.54
W + POS + Loc + span (%)	85.83	86.48	86.54	86.47	86.49	86.55	86.44
Word + POS + span (%)	85.79	86.50	86.45	86.43	86.53	86.49	86.36
LiBSVM							
Items on Queue	2	3	3	3	4	4	4
Items on Stack	4	2	3	4	2	3	4
W (%)	-	-	-	-	-	-	-
W + Loc(%)	-	-	-	-	-	-	-
W + Loc + span (%)	-	-	-	-	-	-	-
W + span (%)	-	-	-	-	-	-	-
POS (%)	74.62	75.41	75.43	75.39	75.62	75.63	75.55
POS + Loc (%)	-	-	-	-	-	-	-
POS + Loc + span (%)	-	-	-	-	-	-	-
POS + span (%)	-	-	-	-	-	-	-
W + POS (%)	-	-	-	-	-	-	-
W + POS + Loc (%)	-	-	-	-	-	-	-
W + POS + Loc + span (%)	-	-	-	-	-	-	-
W + POS + span (%)	-	-	-	-	-	-	-
Id3							
Items on Queue	2	3	3	3	4	4	4
Items on Stack	4	2	3	4	2	3	4
W (%)	67.65	67.94	67.77	67.68	67.97	67.79	67.62
W + Loc (%)	62.37	65.22	63.04	61.89	64.41	62.55	61.49

W + Loc + span (%)	62.69	64.85	63.18	62.26	64.23	62.79	62.00
W + span (%)	61.21	63.60	61.84	60.71	62.81	61.25	60.36
POS (%)	81.64	83.41	81.78	80.54	81.47	79.70	78.81
POS + Loc (%)	74.57	75.48	75.04	74.95	74.83	74.65	74.61
POS + Loc + span (%)	74.51	75.42	74.92	74.83	74.85	74.63	74.57
POS + span (%)	74.28	75.17	74.71	74.59	74.64	74.41	74.33
W + POS (%)	81.02	82.89	81.29	80.36	81.04	79.67	79.09
W + POS + Loc (%)	74.96	75.73	75.39	75.33	75.29	75.07	75.04
W + POS + Loc + span (%)	74.79	75.64	75.22	75.15	75.21	75.00	74.93
W + POS + span (%)	74.55	75.45	75.04	74.97	75.04	74.81	74.73
RandomTree							
Items on Queue	2	3	3	3	4	4	4
Items on Stack	4	2	3	4	2	3	4
W (%)	68.00	68.27	68.26	68.32	68.47	68.51	68.50
W + Loc (%)	70.25	70.64	69.35	68.67	70.19	69.36	69.83
W + Loc + span (%)	-	70.27	-	-	69.97	-	-
W + span (%)	67.46	68.99	68.25	67.39	68.67	67.79	67.89
POS (%)	83.71	85.28	84.71	84.26	84.78	84.31	83.69
POS + Loc (%)	79.18	81.41	80.15	78.84	80.09	78.18	80.12
POS + Loc + span (%)	76.32	79.41	78.02	76.26	78.79	77.42	76.47
POS + span (%)	79.19	80.89	78.69	77.78	79.93	77.28	76.95
W + POS (%)	83.62	85.37	84.34	83.57	84.46	83.33	83.28
W + POS + Loc (%)	80.10	81.84	80.62	79.17	80.27	79.03	77.99
W + POS + Loc + span (%)	77.3	79.59	77.50	76.33	78.09	76.77	75.02
W + POS + span (%)	78.42	80.58	77.87	77.34	78.95	76.87	77.17
NaiveBayes							
Items on Queue	2	3	3	3	4	4	4
Items on Stack	4	2	3	4	2	3	4
W (%)	60.13	65.95	65.48	63.37	65.06	64.68	64.60
W + Loc (%)	57.02	64.12	57.03	55.68	62.21	54.67	52.74
W + Loc + span (%)	49.98	47.29	44.51	47.60	45.28	42.79	45.28
W + span (%)	53.47	55.32	48.75	51.09	52.30	46.46	48.57
POS (%)	70.42	76.78	76.19	74.02	76.01	75.38	74.17
POS + Loc (%)	64.05	74.70	71.13	67.13	72.39	70.68	67.1
POS + Loc + span (%)	58.43	65.72	58.22	57.11	61.27	55.49	54.24
POS + span (%)	61.15	70.93	65.13	61.31	67.62	63.37	59.63
W + POS (%)	66.00	74.67	73.66	71.04	72.12	72.21	71.02
W + POS + Loc (%)	62.25	72.50	69.20	63.57	69.13	67.89	62.70
W + POS + Loc + span (%)	57.62	64.58	56.72	55.55	59.78	53.95	52.73
W + POS + span (%)	59.98	69.69	62.84	59.08	65.33	60.50	56.89

During the evaluation of the classifiers, some widely used classifiers did not yield encouraging results. For example, the LiBSVM classifier (Chang, 2001) which is used in MaltParser did not perform well with the set of features that we have supplied. It only managed to learn successfully from one feature (POS tags), while the accuracy was well below the accuracy of some of the other classifiers. The entries for LiBSVM in Table 1 are incomplete because training takes so long (3 days per case) that future experiments seemed infeasible. However, the fact that it produces no better classification than the J48 classifier in the cases that we have looked at suggests that it is unlikely to substantially outperform it in the remaining cases.

From the large number of experiments we have conducted on several classifiers, we will evaluate NDParser on them in the following section.

6 Evaluating NDParser with Various Classifiers

As presented in Table 2 the classification accuracy varies because each classifier learns differently from the set of training data. In this section, we investigate the effect of different classifiers on parsing. Our objective is to identify the classifiers that help the parser perform best in terms of accuracy and speed (We measure speed as the number of seconds per dependency relation). These experiments also highlight whether different parsing models, which are generated by using different classifiers, contribute in different ways to parsing performance. The optimal classification of accuracy may not necessarily lead to optimal parsing performance. Hence, it is necessary to investigate the effectiveness of different classifiers parsing performance.

Table 2. Parser evaluation with different classifiers, features and settings. *Q* = Queue size, *S* = Stack size, *POS* = part-of-speech tags, *RT* = RandomTree

Classifier	Features	Q	S	UAS (%)	LAS(%)	LS(%)	Spee
J48	**POS**	**4**	**3**	**74.5**	**71.0**	**93.6**	**0.081**
J48	POS	4	4	74.1	70.5	93.6	0.086
J48	POS + location	4	2	70.3	67.0	93.3	0.146
J48	POS + location + span	4	4	69.2	65.9	93.3	0.161
J48	POS + span	4	2	70.6	67.2	93.3	0.145
J48	POS + span	4	3	70.8	67.4	93.3	0.150
J48	POS + span	4	4	70.9	67.5	93.3	0.142
J48	Words + POS	4	3	71.4	67.9	93.5	0.096
J48	Words + POS + location	4	2	69.9	66.5	93.4	0.140
J48	Words + POS + location + span	4	3	68.0	64.8	93.1	0.183
J48	Words + POS + span	4	2	69.8	66.5	93.3	0.161
RT	POS	3	2	70.9	69.4	93.3	0.141
RT	POS + Location	2	1	68.6	65.5	92.9	0.154
RT	POS + Location + span	2	1	67.8	68.1	92.3	0.181
RT	POS + span	2	1	68.2	65.8	92.3	0.181
RT	Words + POS	2	2	70.0	66.6	92.3	0.196
RT	Words + POS + location	2	1	68.7	65.3	92.4	0.198
RT	Words + POS + location + span	2	1	66.8	63.6	92.1	0.196
RT	Words + POS + span	2	1	68.6	65.3	92.3	0.184
Id3	POS	3	1	70.6	67.2	93.4	0.083
Id3	Words + POS	2	2	68.1	64.8	93.3	0.099

From Table 1 we can identify the classification algorithms with the highest degree of accuracy. In this section, we trained NDParser using J48, RandomTree, and Id3 algorithms since they all classified the same set of training data with over 80% accuracy. For each of these algorithms we use the same settings that produced the optimal accuracy. For example, based on the results in Table 1 we will use the POS tags as a

training feature for the J48 algorithm with four items on the queue and three items on the stack because with this setting the algorithm produced 86.05% accuracy, while if we are using POS tags and their locations in a sentence as training features for J48 algorithm then we will use four items on the queue and two items on the stack because the algorithm performs best with this setting, which produced 86.96% accuracy.

The results of the evaluation of our parser are presented in Table 2. The best parsing performance is achieved when training the parser using the J48 classification algorithm on only POS tags as a feature and the window size for the queue and stack is four and three items respectively. The experiments in Table 1 show that training a classifier using a small set of features produces relatively similar classification accuracy to using a larger set of features. However, using a smaller set of features improves the parsing accuracy and speed.

7 A Comparison with the State-of-the-art Parser

In this section, we compare our parser with MaltParser, as shown in Table 3 where we have conducted a 5-fold cross validation on both parsers. We have trained our parser using the J48 classifier with POS tags as features and a window size of four items on the queue and three items on the stack. We can note that our parser is 43% more efficient than MaltParser. Although the unlabelled attachment score of our parser is slightly lower than that of MaltParser (0.7%), the labelled attachment score and the labelled accuracy is more accurate than MaltParser by 1% and 1.4% respectively. We believe that this improved accuracy of labelled attachment score and labelled score is because our parser have information about intermediate items between parent and daughter, which are collected during training (see Section 4 for more details) where such information is not available to MaltParser. MaltParser learns models that contain information about parent and daughter relations and their labels during the training phase where the information about the intermediate items between parents and daughters that we use is not recorded.

Table 3. Parser performance of MaltParser and NDParser

Parser	UAS(%)	LAS(%)	LS(%)	Speed
MaltParser	75.2	70.0	92.2	0.144
NDParser	74.5	71.0	93.6	0.081

The results for MaltParser in Table 3 were obtained by training and testing it on the dependency trees that we extracted from the PATB. The structure of these trees depends on the head-percolation table that is used during the conversion process. It is likely that this underlies the differences between our results for MaltParser and the results published by Nivre (2008) (77.76% for UAS, 65.79% for LAS, and 79.30% for LS), where Niver's result for UAS is slightly better than the one we obtained in this study, for LAS and LS they are slightly worse.

8 Conclusions

In this paper we have presented a simple approach for evaluating and generating parse models from various machine learning classifiers. We have shown that generating a parse model, which is used for guiding data-driven parsers, from different classifiers affects parsing performance in different ways. We have discovered that generating a parse model using a small set of features and settings improves parsing accuracy and speed compared with using large features and settings. We have presented a basic shift-reduce parser, which is based on the arc-standard algorithm of MaltParser, and we have evaluated our parser with various parse models that were generated from different classification algorithms, features and settings.

Acknowledgment

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

References

Chang, C.-c. and Lin, C.-J. (2001), Libsvm: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 3(2):1-27.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Written, I. H., (2009), The weka data mining software: An update, *SIGKDD Explorations*. *Newsl*, 11(1):10-18.

Kuhlmann, M. and Nivre, J., (2010), Transition-Based Techniques for Non-Projective Dependency Parsing, *Northern European Journal of Language Technology,* 2(1):1-19.

Nivre, J. (2008), Algorithms for deterministic incremental dependency parsing, *Computational Linguistics,* 34(4): 513-553.

Kay, M. (1973) The MIND System. In Rustin R. (Ed) Natural *Language Processing*, pp. 155–188. Algorithmics Press, New York.

Jaf, S. F. and Ramsay, A. (2013), Towards the Development of a Hybrid Parser for Natural Languages. In Jones, A. V. and Ng, N., editors, *2013 Imperial College Computing Student Workshop*, volume 35 of *Open Access Series in Informatics (OASIcs)*, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Pp. 49–56.

Maamouri, M. and Bies, A. (2004) Developing an Arabic treebank: methods, guidelines, procedures, and tools. In Proceedings of the Workshop on Computational Approaches to Arabic Script-based Languages. Geneva, pp. 2–9.

Nivre, J., Hall, J. and Nilsson, J. (2006), MaltParser: A Data-Driven Parser-generator for Dependency Parsing, In Proceedings of LREC.

Nivre, J., Rimell, L., McDonald, R. and Gomez-Rodr´ıguez, C. (2010), Evaluation of dependency parsers on unbounded dependencies. In Proceedings of the 23rd International Conference on Computational Linguistics, COLING'10. Beijing, pp. 833–841.

Quinlan, J. R., (1992), Learning with continuous classes, In Proceedings of the 5th Australian Joint Conference on Artificial Intelligence. Sydney. pp. 343–348.