Automatic Discovery of Feature Sets for Dependency Parsing

Peter Nilsson **Pierre Nugues**
Department of Computer Science
Lund University
peter.nilsson.lund@telia.com Pierre.Nugues@cs.lth.se

Abstract

This paper describes a search procedure to discover optimal feature sets for dependency parsers. The search applies to the shift–reduce algorithm and the feature sets are extracted from the parser configuration. The initial feature is limited to the first word in the input queue. Then, the procedure uses a set of rules founded on the assumption that topological neighbors of significant features in the dependency graph may also have a significant contribution. The search can be fully automated and the level of greediness adjusted with the number of features examined at each iteration of the discovery procedure.

Using our automated feature discovery on two corpora, the Swedish corpus in CoNLL-X and the English corpus in CoNLL 2008, and a single parser system, we could reach results comparable or better than the best scores reported in these evaluations. The CoNLL 2008 test set contains, in addition to a *Wall Street Journal* (WSJ) section, an out-of-domain sample from the Brown corpus. With sets of 15 features, we obtained a labeled attachment score of 84.21 for Swedish, 88.11 on the WSJ test set, and 81.33 on the Brown test set.

1 Introduction

The selection of relevant feature sets is crucial to the performance of dependency parsers and this process is still in large part manual. Moreover, feature sets are specific to the languages being analyzed and a set optimal for, say, English can yield poor results in Chinese. With dependency parsers being applied today to dozens of languages, this makes the parametrization of a parser both a tedious and time-consuming operation. Incidentally, the advent of machine-learning methods seems to have shifted the tuning steps in parsing from polishing up grammar rules to the optimization of feature sets. And as with the writing of a grammar, the selection of features is a challenging task that often requires a good deal of effort and inspiration.

Most automatic procedures to build feature sets resort to greedy algorithms. Forward selection constructs a set by adding incrementally features from a predetermined superset while backward elimination removes them from the superset (Attardi et al., 2007). Both methods are sometimes combined (Nivre et al., 2006b). The selection procedures evaluate the relevance of a candidate feature in a set by its impact on the overall parsing score: Does this candidate improve or decrease the performance of the set?

Greedy search, although it simplifies the design of feature sets, shows a major drawback as it starts from a closed superset of what are believed to be the relevant features. There is a broad consensus on a common feature set including the words close to the top of the stack or the beginning of the queue, for the shift–reduce algorithm, but no clear idea on the limits of this set.

In this paper, we describe an automatic discovery procedure that is not bounded by any prior knowledge of a set of potentially relevant features. It applies to the shift–reduce algorithm and the initial feature consists solely of the first word of the queue. The search explores nodes along axes of the parser's data structures and the partially built graph using proximity rules to uncover sequences of relevant, efficient features. Using this procedure on the Swedish corpus in CoNLL-X and the English corpus in CoNLL 2008, we built feature sets that enabled us to reach a labeled attachment score of 84.21 for Swedish, 88.11 on the *Wall Street Journal* section of CoNLL 2008, and 81.33 on the Brown part of it with a set cardinality of 15.

2 Transition-based Parsing

Transition-based methods (Covington, 2001; Nivre, 2003; Yamada and Matsumoto, 2003; Zhang and Clark, 2009) have become a popular approach in multilingual dependency parsing because of their speed and performance. Transition-based methods share common properties and build a dependency graph from a sequence of actions, where each action is determined using a feature function. In a data-driven context, the function is typically implemented as a classifier and the features are extracted from the partially built graph and the parser's data structures, most often a queue and a stack.

2.1 Parser Implementation

In this study, we built a parser using Nivre's algorithm (Nivre, 2003). The parser complexity is linear and parsing completes in at most $2n+1$ operations, where n is the length of the sentence. Table 1 shows the transitions and actions to construct a dependency graph.

Given a sentence to parse, we used a classifier-based guide to predict the transition sequence to apply. At each step, the guide extracts features from the parser configuration and uses them as input to a classifier to predict the next transition. Before training the classification models, we projectivized the corpus sentences (Kunze, 1967; Nivre and Nilsson, 2005). We did not attempt to recover nonprojective sentences after parsing.

2.2 Training and Parsing Procedure

We extracted the features using a gold-standard parsing of the training set. We organized the classification, and hence the feature extraction, as a two-step process. The first step determines the action among *LeftArc*, *RightArc*, *Reduce*, and *Shift*; the second one, the grammatical function, if the action is either a left arc or a right arc.

Once the features are extracted, we train the corresponding models that we apply to the test corpus to predict the actions and the arc labels.

Action	Parser configuration
Init.	$\langle nil, W, \emptyset\rangle$
End	$\langle S, nil, G\rangle$
LeftArc	$\langle n\vert S, n'\vert Q, G\rangle \rightarrow \langle S, n'\vert Q, G\cup\{\langle n', n\rangle\}\rangle$
RightArc	$\langle n\vert S, n'\vert Q, G\rangle \rightarrow \langle n'\vert n\vert S, Q, G\cup\{\langle n, n'\rangle\}\rangle$
Reduce	$\langle n\vert S, Q, G\rangle \rightarrow \langle S, Q, G\rangle$
Shift	$\langle S, n\vert Q, G\rangle \rightarrow \langle n\vert S, Q, G\rangle$

Table 1: Parser transitions (Nivre, 2003). W is the input, G, the graph, S, the stack, and Q, the queue. The triple $\langle S, Q, G\rangle$ represents the parser configuration and n, n', and n'' are lexical tokens. $\langle n', n\rangle$ represents an arc from n' to n.

3 Feature Discovery

We designed an automatic procedure to discover and select features that is guided by the structure of the graph being constructed. The search algorithm is based on the assumption that if a feature makes a significant contribution to the parsing performance, then one or more of its topological neighbors in the dependency graph may also be significant. The initial state, from which we derive the initial feature, consists of the first word in the queue. There is no other prior knowledge on the features.

3.1 Node Attributes

In the discovery procedure, we considered the nodes of four data structures: the queue, the stack, the sentence, and the graph being constructed. We extracted three attributes (or fields) from each node: two static ones, the lexical value of the node and its part of speech, and a dynamic one evaluated at parse time: the dependency label of the arc linking the node to its head, if it exists. We denoted the attributes of node w, respectively,

$LEX(w)$, $POS(w)$, and $DEP(w)$. These attributes are used as input by most dependency parsers, whatever the language being parsed.

3.2 Search Axes

The feature search covers three different axes: the parser's data structures – the queue and the stack –, the graph being constructed, and the sentence. Given a feature set at step n of the discovery procedure, we defined a successor function that generates the set of topological neighbors of all the members in the feature set along these three axes. For a particular feature:

The data structure axis consists of the nodes in the stack and the queue. The immediate neighbors of a node in the stack are the adjacent nodes above and below. In the queue, these are the adjacent nodes before and after it. The top node on the stack and the next node in the queue have a special connection, since they are the ones used by the parser when creating an arc. Therefore, we considered them as immediate neighbors to each other. For a node that is neither in the stack, nor in the queue, there is no connection along this axis.

The graph axes traverse the partially constructed graph horizontally and vertically. The horizontal axis corresponds to the sibling nodes connected by a common head (Figure 1). The immediate neighbors of a node are its nearest siblings to the left and to the right. The vertical axis corresponds to the head and child nodes. The immediate neighbors are the head node as well as the leftmost and rightmost child nodes. There is no connection for nodes not yet part of the graph.

The sentence axis traverses the nodes in the order they occur in the original sentence. The immediate neighbors of a node are the previous and next words in the sentence.

4 Representing Features and Their Neighbors

We represented features with a parameter format partly inspired by MaltParser (Nivre et al., 2006a).

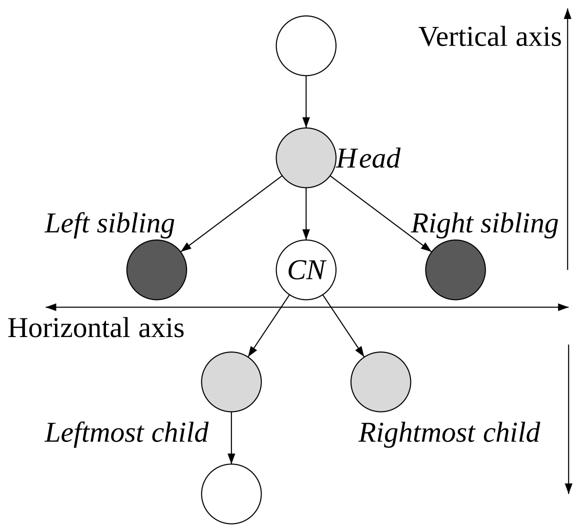

Figure 1: The vertical and horizontal axes, respectively in light and dark gray, relative to *CN*.

Each parameter consists of two parts. The first one represents a node in a data structure ($STACK$ or $QUEUE$) and an attribute:

The nodes are identified using a zero-based index. Thus $STACK_1$ designates the second node on the stack.

The attribute of a node is one of part of speech (POS), lexical value (LEX), or dependency label (DEP), as for instance $LEX(QUEUE_0)$ that corresponds to the lexical value of the first token in the queue.

The second part of the parameter is an optional navigation path that allows to find other destination nodes in the graph. It consists of a sequence of instructions to move from the start node to the destination node. The list of possible instructions are:

- h: head of the current node;
- lc/rc: leftmost/rightmost child of the node;
- pw/fw: previous/following word of the node in the original sentence.

An example of a feature obtained using the navigation part is $POS(STACK_1\ lc\ pw)$, which is interpreted as: start from $STACK_1$. Then, using the instructions lc and pw, move to the left child of the start node and to the previous word of this child in the sentence. The requested feature is the part of speech of the destination node.

5 Initial State and Successor Function

The feature discovery is an iterative procedure that grows the feature set with one new feature at each iteration. We called *generation* such an iteration, where generation 1 consists of a single node. We denoted $FeatSet_i = \{f_1, f_2, ..., f_i\}$ the feature set obtained at generation i.

Although the features of a classifier can be viewed as a set, we also considered them as a tuple, where $Feat_i = \langle f_1, f_2, ..., f_i \rangle$ is the i-tuple at generation i and f_k, the individual feature discovered at generation k with $1 \leqslant k \leqslant i$. This enables us to keep the order in which the individual features are obtained during the search.

5.1 Initial State

We start the feature search with the empty set, $\emptyset$, that, by convention, has one neighbor: the first node in the queue $QUEUE_0$. We chose this node because this is the only one which is certain to exist all along the parsing process. Intuitively, this is also obvious that $QUEUE_0$ plays a significant role when deciding a parsing action. We defined the successor function of the empty set as: $SUCC(\emptyset) = \{POS(QUEUE_0), LEX(QUEUE_0)\}$.

5.2 Successors of a Node

The successors of a node consist of itself and all its topological neighbors along the three axes with their three possible attributes: part of speech, lexical value, and dependency label. For a particular feature in *FeatSet*, the generation of its successors is carried out through the following steps:

1. Interpret the feature with its possible navigation path and identify the destination node n.
2. Find all existing immediate neighboring nodes of n along the three search axes.
3. Assign the set of attributes – POS, LEX, and DEP – to n and its neighboring nodes.

If at any step the requested node does not exist, the feature evaluates to $NOTHING$.

5.3 Rules to Generate Neighbors

The generation of all the neighbors of the features in *FeatSet* may create duplicates as a same node can sometimes be reached from multiple paths. For instance, if we move to the leftmost child of a node and then to the head of this child, we return to the original node.

To compute the successor function, we built a set of rules shown in Table 2. It corresponds to a subset of the rules described in the axis search (Sect. 3.2) so that it omits the neighbors of a node that would unavoidably create redundancies. The third column in Table 2 shows the rules to generate the neighbors of $POS(QUEUE_0)$. They correspond to the rows:

PL. This stands for the POS and LEX attributes of the node. We only add $LEX(QUEUE_0)$ as we already have $POS(QUEUE_0)$.

PLD lc *and* PLD rc. POS, LEX, and DEP of the node's leftmost and rightmost children.

PLD pw. POS, LEX, and DEP of the previous word in the original string. The following word is the same as the next node in the queue, which is added in the next step. For that reason, *following word* is not added.

PL $QUEUE_1$. POS and LEX of $QUEUE_1$.

PLD $STACK_0$. POS, LEX, and DEP of $STACK_0$. This rule connects the queue to the top node of the stack.

Table 3 summarizes the results of the rule application and shows the complete list of successors of $POS(QUEUE_0)$. In this way, the search for a node's neighbors along the axes is reduced to one direction, either left or right, or up or down, that will depend on the topological relation that introduced the node in the feature set.

6 Feature Selection Algorithm

At each generation, we compute the Cartesian product of the current feature tuple $Feat_i$ and the set defined by its neighbors. We define the set of candidate tuples $CandFeat_{i+1}$ at generation $i+1$ as:

$$CandFeat_{i+1} = \{Feat_i\} \times SUCC(Feat_i),$$

where we have $Card(CandFeat_{i+1}) = Card(SUCC(Feat_i))$.

The members of $CandFeat_{i+1}$ are ranked according to their parsing score on the development

Data structures				Navigation paths					
$STACK_0$	*$STACK_n, n > 0$*	*$QUEUE_0$*	*$QUEUE_n, n > 0$*	*h*	*lc, rc*	*ls*	*rs*	*pw*	*fw*
PLD	PLD	PL	PL	h				h	h
PLD h	PLD h				lc	lc	lc	lc	lc
PLD lc	PLD lc	PLD lc			rc	rc	rc	rc	rc
PLD rc	PLD rc	PLD rc		ls	ls	ls		ls	ls
PLD ls	PLD ls			rs	rs		rs	rs	rs
PLD rs	PLD rs			pw	pw	pw	pw	pw	
PLD pw	PLD pw	PLD pw		fw	fw	fw	fw		fw
PLD fw	PLD fw								
PLD $STACK_1$	PLD $STACK_{n+1}$	PL $QUEUE_1$	PL $QUEUE_{n+1}$						
PL $QUEUE_0$		PLD $STACK_0$							

Table 2: Rules to compute the successors of a node. For each node category given in row 2, the procedure adds the features in the column headed by the category. PLD stands for the *POS*, *LEX*, and *DEP* attributes. In the right-hand side of the table, the category corresponds to the last instruction of the navigation path, if it exists, for instance *pw* in the feature $POS(STACK_1\ lc\ pw)$. We read the six successors of this node in the fifth column headed by *pw*: $STACK_1\ lc\ pw\ h$, $STACK_1\ lc\ pw\ lc$, $STACK_1\ lc\ pw\ rc$, $STACK_1\ lc\ pw\ ls$, $STACK_1\ lc\ pw\ rs$, and $STACK_1\ lc\ pw\ pw$. We then apply all the attributes to these destination nodes to generate the features.

Initial feature	POS	QUEUE	0	
Successors	LEX	QUEUE	0	
	PLD	QUEUE	0	lc
	PLD	QUEUE	0	rc
	PLD	QUEUE	0	pw
	PL	QUEUE	1	
	PLD	STACK	0	

Table 3: Features generated by the successor function $SUCC(\{POS(QUEUE_0)\})$. PLD stands for the three attributes *POS*, *LEX*, and *DEP* of the node; PL for *POS* and *LEX*.

set and when applying a greedy best-first search, $Feat_{i+1}$ is assigned with the tuple yielding the highest score:

$$Feat_{i+1} \leftarrow eval\_best(CandFeat_{i+1}).$$

The procedure is repeated with the immediate neighbors of $Feat_{i+1}$ until the improvement of the score is below a certain threshold.

We extended this greedy version of the discovery with a beam search that retains the *N*-best successors from the candidate set. In our experiments, we used beam widths of 4 and 8.

7 Experimental Setup

In a first experiment, we used the Swedish corpus of the CoNLL-X shared task (Buchholz and Marsi, 2006). In a second experiment, we applied the feature discovery procedure to the English corpus from CoNLL 2008 (Surdeanu et al., 2008), a dependency corpus converted from the Penn Treebank and the Brown corpus. In both experiments, we used the LIBSVM package (Chang and Lin, 2001) with a quadratic kernel, $\gamma = 0.2$, $C = 0.4$, and $\varepsilon = 0.1$. These parameters are identical to Nivre et al. (2006b) to enable a comparison of the scores.

We evaluated the feature candidates on a development set using the labeled and unlabeled attachment scores (LAS and UAS) that we computed with the `eval.pl` script from CoNLL-X. As there was no development set for the Swedish corpus, we created one by picking out every 10th sentence from the training set. The training was then carried out on the remaining part of the set.

8 Feature Discovery on a Swedish Corpus

In a first run, the search was optimized for the UAS. In a second one, we optimized the LAS. We also report the results we obtained subsequently on the CoNLL-X test set as an indicator of how well the training generalized.

8.1 The First and Second Generations

Table 4 shows the feature performance at the first generation sorted by UAS. The first row shows the two initial feature candidates, $\langle POS(QUEUE_0)\rangle$ and $\langle LEX(QUEUE_0)\rangle$. The third row shows the score produced by the initial features alone. The next rows show the unlabeled and labeled attachment scores with feature pairs combining one of the initial features and the one listed in the row. The combination of $POS(QUEUE_0)$ and $POS(STACK_0)$ yielded the best UAS: 74.02. The second feature improves the performance of $POS(QUEUE_0)$ by more than 30 points from 43.49.

For each generation, we applied a beam search. We kept the eight best pairs as starting states for the second generation and we added their neighboring nodes. Table 5 shows the eight best results out of 38 for the pair $\langle POS(QUEUE_0), POS(STACK_0)\rangle$.

Parent state: $\langle POS(QUEUE_0), POS(STACK_0)\rangle$				
Dev set		Test set		
UAS	LAS	UAS	LAS	Successors
79.50	65.34	79.07	65.86	P QUEUE 1
78.73	66.98	76.04	64.51	L STACK 0 fw
77.42	63.08	74.63	61.86	L QUEUE 1
77.06	64.54	75.28	62.90	L QUEUE 0 pw
76.83	66.01	73.61	63.77	L QUEUE 0
76.63	63.62	74.75	63.17	P STACK 0 fw
76.44	64.24	74.09	62.02	L STACK 0
76.39	63.12	73.99	61.16	L QUEUE 0 lc

Table 5: Ranking the successors of $\langle POS(QUEUE_0), POS(STACK_0)\rangle$ on the Swedish corpus. Out of the 38 successors, we show the eight that yielded the best results. P stands for *POS*, L for *LEX*, and D for *DEP*.

8.2 Optimizing the Unlabeled Attachement Score

We iterated the process over a total of 16 generations. Table 6, left-hand side, shows the list of the best scores for each generation. The scores on the development set increased steadily until generation 13, then reached a plateau, and declined around generation 15. The test set closely followed the development set with values about 1% lower. On this set, we reached a peak performance at generation 12, after which the results decreased.

Table 6, right-hand side, shows the features producing the final score in their order of inclusion in the feature set. As we applied a beam search, a feature listed at generation *i* does not necessary correspond to the highest score for this generation, but belongs to the feature tuple producing the best result at generation 16.

8.3 Optimizing the Labeled Attachement Score

We also applied the feature discovery with a search optimized for the labeled attachment score. This time, we reduced the beam width used in the search from 8 to 4 as we noticed that the candidates between ranks 5 and 8 never contributed to the best scoring feature set for any generation.

We observed a score curve similar to that of the UAS-optimized search. The train set followed the development set with increasing values for each generation but 1-2% lower. The optimal value was obtained at generation 15 with 84.21% for the test set. Then, the score for the test set decreased.

9 Feature Discovery on a Corpus of English

The training and development sets of the CoNLL 2008 corpus contain text from the *Wall Street Journal* exclusively. The test set contains text from the Brown corpus as well as from the *Wall Street Journal*. Table 7 shows the results after 16 generations. We used a beam width of 4 and the tests were optimized for the unlabeled attachment score. As for Swedish, we reached the best scores around generation 14-15. The results on the in-domain test set peaked at 90.89 and exceeded the results on the development set. As expected, the results for the out-of-domain corpus were lower, 87.50, however the drop was limited to 3.4.

10 Discussion and Conclusion

The results we attained with feature set sizes as small as 15 are competitive or better than figures

Parent state		$\langle POS(QUEUE_0)\rangle$			$\langle LEX(QUEUE_0)\rangle$
UAS	LAS	Successors	UAS	LAS	Successors
43.49	26.45	None	42.76	23.56	None
74.02	59.67	POS STACK 0	**65.86**	52.18	POS STACK 0
67.77	54.50	LEX STACK 0	**58.59**	45.51	LEX STACK 0
58.37	41.83	POS QUEUE 0 pw	**51.98**	37.70	POS QUEUE 0 pw
55.28	38.49	LEX QUEUE 0 pw	50.44	29.71	POS QUEUE 1
51.53	30.43	POS QUEUE 1	50.38	35.24	LEX QUEUE 0 pw
51.05	32.66	LEX QUEUE 0 lc	49.37	32.27	POS QUEUE 0
49.71	31.54	POS QUEUE 0 lc	48.91	27.77	LEX QUEUE 1
49.49	29.18	LEX QUEUE 1	48.66	29.91	LEX QUEUE 0 lc
49.37	32.27	LEX QUEUE 0	47.25	28.92	LEX QUEUE 0 rc
48.68	29.34	DEP STACK 0	47.09	28.65	POS QUEUE 0 lc
48.47	30.84	LEX QUEUE 0 rc	46.68	27.08	DEP QUEUE 0 lc
46.77	26.86	DEP QUEUE 0 lc	45.69	27.83	POS QUEUE 0 rc
46.40	29.95	POS QUEUE 0 rc	44.77	26.17	DEP STACK 0
42.27	25.21	DEP QUEUE 0 pw	44.43	26.47	DEP QUEUE 0 rc
41.04	26.56	DEP QUEUE 0 rc	41.87	23.04	DEP QUEUE 0 pw

Table 4: Results of the beam search on the Swedish corpus at the first generation with the two initial feature candidates, $\langle POS(QUEUE_0)\rangle$ and $\langle LEX(QUEUE_0)\rangle$, respectively on the left- and right-hand side of the table. The third row shows the score produced by the initial features alone and the next rows, the figures for the candidate pairs combining the initial feature and the successor listed in the row. The eight best combinations shown in bold are selected for the next generation.

Generation	Dev set		Test set		Features
	UAS	LAS	UAS	LAS	
1	43.49	26.45	45.93	30.19	POS QUEUE 0
2	74.02	59.67	71.60	58.37	POS STACK 0
3	79.50	65.34	79.07	65.86	POS QUEUE 1
4	83.58	71.76	82.75	70.98	LEX STACK 0 fw
5	85.96	76.03	84.82	74.75	LEX STACK 0
6	87.23	77.32	86.34	76.52	LEX QUEUE 0 lc
7	88.42	80.00	87.67	78.99	POS STACK 1
8	89.43	81.56	88.09	80.26	LEX QUEUE 1
9	89.84	83.20	88.69	82.33	LEX QUEUE 0
10	90.23	83.89	89.17	83.31	DEP STACK 0 lc
11	90.49	84.31	89.58	**83.85**	POS STACK 0 fw
12	90.73	84.47	**89.66**	83.83	LEX STACK 0 fw ls
13	90.81	84.60	89.52	83.75	LEX STACK 0 fw ls lc
14	90.81	**84.70**	89.32	83.73	POS STACK 1 h
15	**90.85**	84.67	89.13	83.21	LEX STACK 1 rs
16	90.84	84.68	88.65	82.75	POS STACK 0 fw ls rc

Table 6: Best results for each generation on the Swedish corpus, optimized for UAS. Figures in bold designate the best scores. The right-hand side of the table shows the feature sequence producing the best result at generation 16.

Generation	Dev set		Test set WSJ		Test set Brown		Features
	UAS	LAS	UAS	LAS	UAS	LAS	
1	45.25	33.77	45.82	34.49	52.12	40.70	POS QUEUE 0
2	64.42	55.64	64.71	56.44	71.29	62.41	LEX STACK 0
3	78.62	68.77	78.99	70.30	78.67	65.17	POS QUEUE 1
4	81.83	76.67	82.46	77.82	80.57	72.95	LEX STACK 0 fw
5	84.43	79.78	84.89	80.88	84.03	76.99	POS STACK 0
6	85.95	81.60	86.61	82.93	84.55	77.80	DEP QUEUE 0 lc
7	86.95	82.73	87.73	84.09	85.26	78.48	LEX STACK 1
8	88.03	83.62	88.52	84.74	85.66	78.73	LEX QUEUE 1
9	88.61	84.97	89.15	86.20	86.29	79.86	LEX QUEUE 0
10	89.09	85.43	89.47	86.60	86.43	80.02	POS QUEUE 2
11	89.54	85.87	90.25	87.40	87.00	80.75	POS STACK 0 pw
12	89.95	86.21	90.63	87.77	86.87	80.46	POS QUEUE 3
13	90.26	86.56	90.64	87.80	87.35	80.86	POS STACK 1 pw
14	90.54	86.81	90.71	87.88	**87.50**	81.30	POS QUEUE 0 pw
15	90.61	86.94	**90.89**	**88.11**	87.47	**81.33**	LEX STACK 0 lc
16	**90.65**	**87.00**	90.88	88.09	87.42	81.28	POS STACK 0 pw ls

Table 7: Best results for each generation. English corpus. Selection optimized for UAS.

reported by state-of-the-art transition-based systems. We reached a UAS of 89.66 on the CoNLL-X Swedish corpus. On the same corpus, the top scores reported in the shared task were slightly lower: 89.54 and 89.50. Our best LAS was 84.21, and the two best scores in CoNLL-X were 84.58 and 82.55. Our results for the English corpus from CoNLL 2008 were optimized for an unlabeled attachment score and we obtained 90.89 for the in-domain test set and 87.50 for the out-of-domain one. Our best LAS were 88.11 and 81.33. Official results in CoNLL 2008 only reported the labeled attachment scores, respectively 90.13 and 82.81[1].

We believe these results remarkable. We used a single-parser system as opposed to ensemble systems and the results on the Brown corpus show an excellent resilience and robustness on out-of-domain data. The automatic discovery produced results matching or exceeding comparable systems, although no prior knowledge of the language being analyzed was used and no feature set was provided to the parser.

Although, a systematic search requires no intuitive guessing, it still consumes a considerable machine time. Due to the learning algorithm we use, SVM, training a model takes between 1 and 130 hours depending on the size of the corpus. The number of models to train at each generation corresponds to the number of feature candidates times the beam width. The first generation contains about 15 feature candidates per feature set and since features are only added, the number of candidates can grow to 100 at generation 10.

We believe there is a margin for improvement both in the parsing scores and in the time needed to determine the feature sets. Our scores in Swedish were obtained with models trained on 90% of the training set. They could probably be slightly improved if they had been trained on a complete set. In our experiments, we used three attributes: the part of speech, lexical value, and dependency label of the node. These attributes could be extended to lemmas and grammatical features. SVMs yield a high performance, but they are slow to train. Logistic regression with, for instance, the LIBLINEAR package (Fan et al., 2008) would certainly reduce the exploration time.

[1]Results are not exactly comparable as we used the CoNLL-X evaluation script that gives slightly higher figures.

Acknowledgments

The research leading to these results has received funding from the European community's seventh framework program FP7/2007-2013, challenge 2, cognitive systems, interaction, robotics, under grant agreement No 230902—ROSETTA.

References

Attardi, Giuseppe, Felice Dell'Orletta, Maria Simi, Atanas Chanev, and Massimiliano Ciaramita. 2007. Multilingual dependency parsing and domain adaptation using DeSR. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 1112–1118, Prague, Czech Republic, June. Association for Computational Linguistics.

Buchholz, Sabine and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 149–164, New York City, June. Association for Computational Linguistics.

Chang, Chih-Chung and Chih-Jen Lin. 2001. LIBSVM: a library for support vector machines. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Covington, Michael A. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, Athens, Georgia.

Fan, Rong-En, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874.

Kunze, Jürgen. 1967. Die Behandlung nicht-projektiver Strukturen bei der syntaktischen Analyse und Synthese des englischen und des deutschen. In *MASPEREVOD-67: Internationales Symposium der Mitgliedsländer des RGW*, pages 2–15, Budapest, 10.–13. Oktober.

Nivre, Joakim and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 99–106, Ann Arbor, June.

Nivre, Joakim, Johan Hall, and Jens Nilsson. 2006a. Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC2006)*, pages 2216–2219, Genoa, May 24-26.

Nivre, Joakim, Johan Hall, Jens Nilsson, Gülsen Eryigit, and Svetoslav Marinov. 2006b. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 221–225, New York, June, 8-9.

Nivre, Joakim. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 149–160, Nancy, 23-25 April.

Surdeanu, Mihai, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. 2008. The CoNLL 2008 shared task on joint parsing of syntactic and semantic dependencies. In *CoNLL 2008: Proceedings of the 12th Conference on Computational Natural Language Learning*, pages 159–177, Manchester, August.

Yamada, Hiroyasu and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 195–206, Nancy, 23-25 April.

Zhang, Yue and Stephen Clark. 2009. Transition-based parsing of the Chinese treebank using a global discriminative model. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT 09)*, pages 162–171, Paris, October.