MaltOptimizer: A System for MaltParser Optimization

Miguel Ballesteros† **Joakim Nivre‡**

†Universidad Complutense de Madrid, Spain
miballes@fdi.ucm.es

‡Uppsala University, Sweden
joakim.nivre@lingfil.uu.se

Abstract

Freely available statistical parsers often require careful optimization to produce state-of-the-art results, which can be a non-trivial task especially for application developers who are not interested in parsing research for its own sake. We present MaltOptimizer, a freely available tool developed to facilitate parser optimization using the open-source system MaltParser, a data-driven parser-generator that can be used to train dependency parsers given treebank data. MaltParser offers a wide range of parameters for optimization, including nine different parsing algorithms, two different machine learning libraries (each with a number of different learners), and an expressive specification language that can be used to define arbitrarily rich feature models. MaltOptimizer is an interactive system that first performs an analysis of the training set in order to select a suitable starting point for optimization and then guides the user through the optimization of parsing algorithm, feature model, and learning algorithm. Empirical evaluation on data from the CoNLL 2006 and 2007 shared tasks on dependency parsing shows that MaltOptimizer consistently improves over the baseline of default settings and sometimes even surpasses the result of manual optimization.

Keywords: Dependency Parsing, MaltParser, Optimization

1. Introduction

The performance of statistical parsers for natural language has improved tremendously during the last two decades, and there are now a number of different systems that can be used to develop parsers for new languages and applications. This includes constituency-based parsers like the Stanford Parser (Klein and Manning, 2002) and the Berkeley Parser (Petrov et al., 2006) as well as dependency parsers such as MaltParser (Nivre et al., 2004) and MSTParser (McDonald et al., 2005). However, the development of accurate parsers for new languages may require careful optimization, a task that is often non-trivial especially for application developers that may lack both the competence and the motivation to perform extensive parsing experiments. As an illustration of the importance of optimization, Hall et al. (2007) report differences of over 3 percent absolute in labeled attachment score between the baseline and the optimized system for some languages in the CoNLL 2007 shared task on dependency parsing. It is worth noting that these differences are greater than those typically reported when comparing different parsers on the same data sets.

MaltParser (Nivre et al., 2006a) is an open-source system for data-driven dependency parsing that offers a wide range of parameters for optimization. First of all, it implements nine different transition-based parsing algorithms, each of which has its own parameters. Secondly, for each parsing algorithm it is possible to define arbitrarily complex feature models using an expressive feature specification language. Finally, any combination of parsing algorithm and feature model can be combined with a number of different machine learning algorithms available in LIBSVM (Chang and Lin, 2001) and LIBLINEAR (Fan et al., 2008). Just running the system with default settings when training a new parser is likely to result in suboptimal performance, with respect to parsing accuracy as well as efficiency, but finding a good combination of all parameters can be a daunting task even for experienced researchers.

To facilitate MaltParser optimization, we propose a system that automates the search for optimal parameters based on an analysis of the training set and on optional input from the user at various points. Although the system is not guaranteed to find truly optimal settings, our experiments indicate that it invariably improves over the default settings and often approaches (or even surpasses) the results obtained through careful manual optimization. In this way, we believe that it can be a useful tool both for application developers who do not want to get involved in extensive parsing experiments and for expert users who quickly want to find a good starting point for advanced optimization. The system, which is called MaltOptimizer, is available for download under an open-source license together with a user guide at *http://nil.fdi.ucm.es/maltoptimizer* and appears in the system demonstration session at EACL 2012 (Ballesteros and Nivre, 2012).

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the MaltParser system and the parameters that need to be optimized, while Section 3 describes the three-phase optimization process implemented in MaltOptimizer. Section 4 reports experiments on data sets from the CoNLL 2006 and 2007 shared tasks on dependency parsing, Section 5 discusses related work, and Section 6 concludes.

2. MaltParser

MaltParser[1] is a freely available implementation of the parsing models described in Nivre (2006), Nivre (2007) and Nivre (2009). These models are often characterized as *transition-based*, because they reduce the problem of

[1] http://www.maltparser.org

parsing a sentence to the problem of finding an optimal path through an abstract transition system, or state machine. This is sometimes equated with shift-reduce parsing, but in fact includes a much broader range of transition systems (Nivre, 2008). Transition-based parsers learn models that predict the next state given the current state of the system, including features over the history of parsing decisions and the input sentence. At parsing time, the parser starts in an initial state and greedily moves to subsequent states – based on the predictions of the model – until a terminal state is reached. The greedy, deterministic parsing strategy results in highly efficient parsing, with run-times often linear in sentence length, and also facilitates the use of arbitrary non-local features, since the partially built dependency tree is fixed in any given state. However, greedy inference can also lead to error propagation if early predictions place the parser in incorrect states (McDonald and Nivre, 2007).

When optimizing MaltParser for a new language or domain, there are essentially three aspects of the system that need to be optimized:

1. Parsing algorithm
2. Feature model
3. Learning algorithm

We now describe each of these aspects in turn.

2.1. Parsing Algorithm

Selecting a parsing algorithm essentially means selecting a transition system together with certain constraints on search in that transition system. The main parsing algorithms available in MaltParser can be grouped into three families:[2] (i) Nivre's algorithms include the arc-eager and arc-standard versions of the algorithm described in Nivre (2003) and Nivre (2004); (ii) Covington's algorithms include the projective and non-projective versions of the algorithm described by Covington (2001) and adapted by Nivre (2008); (iii) Stack algorithms include the projective and non-projective versions of the algorithm described in Nivre (2009) and Nivre et al. (2009). While both the Covington and the Stack family contains algorithms that can handle non-projective trees, the Nivre family does not. However, any projective parsing algorithm can be used to parse non-projective trees through the techniques known as pseudo-projective parsing (Nivre and Nilsson, 2005).

2.2. Feature Model

One of the advantages of the transition-based approach to dependency parsing is that it enables rich history-based feature models for predicting the next transition, and MaltParser provides an expressive specification language for defining feature models of arbitrary complexity. Features are defined relative to tokens in the main data structures for a given parsing algorithm, which normally include at least a *stack* holding partially processed tokens and a *buffer* holding remaining input tokens. The actual feature values are normally linguistic attributes of one or more tokens. The following attributes are available in the CoNLL-X format assumed by the parser:[3]

1. FORM: Word form.
2. LEMMA: Lemma.
3. CPOSTAG: Coarse-grained part-of-speech tag.
4. POSTAG: Fine-grained part-of-speech tag.
5. FEATS: List of morphosyntactic features (e.g., case, number, tense, etc.)
6. DEPREL: Dependency relation to head.

The attributes LEMMA and FEATS are not available in all data sets, and the CPOSTAG and POSTAG tags are sometimes identical. Note also that the DEPREL attribute is only available dynamically in the partially built dependency tree. We will refer to features that are defined relative to the partial parse as dependency tree features.

The default model for a MaltParser parsing algorithm (which is what is used when the system is run with default settings) includes the following groups of features:

1. A wide window of POSTAG features over the stack and buffer (typically of length 6).
2. A narrower window of FORM features over the stack and buffer (typically of length 3).
3. A small set of DEPREL features over dependents (and heads) of the most central tokens on the stack and in the buffer (typically of size 4).
4. A small set of combinations of the above features, in particular POSTAG n-grams and pairs of POSTAG and FORM features.

As we shall see later, optimizing the feature model implies both tuning the size of these feature groups and exploring additional features such as CPOSTAG, LEMMA and FEATS features.

2.3. Learning Algorithm

MaltParser comes with two libraries for machine learning: LIBSVM (Chang and Lin, 2001) and LIBLINEAR (Fan et al., 2008). The LIBSVM package enables the use of support vector machines with kernels, which facilitates feature selection but has the drawback of being rather inefficient both during training and parsing. The LIBLINEAR package only supports plain linear classifiers, which makes training and parsing very fast but put higher demands on feature selection. Both packages contain a number of specific algorithms each with their own hyperparameters that need to be optimized. For the development of MaltOptimizer we have restricted out attention to the LIBLINEAR package in the interest of efficiency.

[2] More algorithms have been added in recent versions but are not handled by MaltOptimizer and are therefore omitted from the presentation here.

[3] http://nextens.uvt.nl/depparse-wiki/DataFormat

3. MaltOptimizer 1.0

MaltOptimizer is a software tool written in Java that implements a stepwise optimization procedure for MaltParser based on the heuristics described in Nivre and Hall (2010). The system takes as input a training set, consisting of sentences annotated with dependency trees in the CoNLL-X data format (Buchholz and Marsi, 2006). The optimization process has three phases with optional input from the user after each phase:

1. Data validation and analysis
2. Parsing algorithm selection
3. Feature selection and hyper-parameter optimization

The machine learning algorithm used throughout the process is the multiclass support vector machine of Crammer et al. (2006) as implemented in LIBLINEAR (Fan et al., 2008). We now describe each of the three phases in turn.

3.1. Phase 1: Data Analysis

MaltOptimizers starts by validating that the data is correctly formatted, using the official validation script from the CoNLL-X shared task *validateFormat.py*.[4] If fatal errors are found, MaltOptimizer stops and informs the user to check the output of the script. If only warnings occur, the system proceeds with the data analysis but recommends the user to check the script output since the warnings may indicate inconsistent annotation.

In the data analysis, MaltOptimizer gathers information about the following properties of the training set:

1. Number of words/sentences.
2. Percentage of non-projective arcs/trees.
3. Existence of "covered roots" (arcs spanning tokens with HEAD = 0).
4. Frequency of labels used for tokens with HEAD = 0.
5. Existence of non-empty feature values in the LEMMA and FEATS columns.
6. Identity (or not) of feature values in the CPOSTAG and POSTAG columns.

Property 1 is used to suggest the best validation strategy (simple train-devtest split or 5-fold cross-validation); properties 2–4 are relevant for the choice of parsing algorithm in Phase 2 as well as for some parameters of pre- and post-processing; properties 5–6 are important for the feature selection experiments in Phase 3. When the data analysis is completed, MaltOptimizer creates a baseline option file to be used as the starting point for optimization. The user is given the opportunity to edit this option file before optimization continues and may also choose to stop the process and continue with manual optimization.

Based on the size of the data set, MaltOptimizer recommends the user to choose on of the two built-in validation methods during Phase 2 and 3: (i) simple train-devtest split (80% for training, 20% for development testing) or (ii) 5-fold cross validation (20% in each fold). In both cases, the system performs stratified sampling to ensure a similar distribution of data in all subsets. Simple train-devtest split is recommended for large data sets, where cross-validation would be time consuming and a single devtest set is large enough to give reliable estimates. Cross-validation is recommended for smaller data sets, where a single evaluation estimate might be unreliable and the extra time needed to run cross-validation is tolerable. In either case, the user can override the system's recommendation.

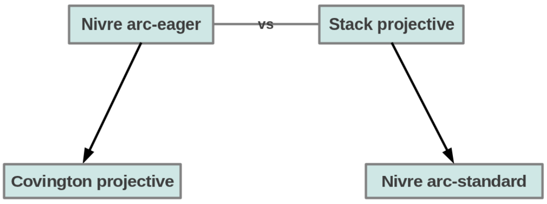

Figure 1: Decision tree for best projective algorithm.

3.2. Phase 2: Parsing Algorithm

In the second phase, MaltOptimizer explores a subset of the parsing algorithms implemented in MaltParser, based on the results of the data analysis. In particular, if there are no non-projective dependencies in the training set, then only projective algorithms are explored, including the arc-eager and arc-standard versions of Nivre's algorithm (Nivre, 2003; Nivre, 2004), Covington's projective parsing algorithm (Covington, 2001; Nivre, 2008), and the projective Stack algorithm (Nivre, 2009). By contrast, if the training set contains a substantial amount of non-projective dependencies, then MaltOptimizer instead tests Covington's non-projective algorithm (Covington, 2001; Nivre, 2008), the non-projective Stack algorithms (Nivre, 2009; Nivre et al., 2009) as well as projective algorithms in combination with pseudo-projective parsing (Nivre and Nilsson, 2005). If the amount of non-projective dependencies is intermediate, both groups of algorithms are explored.

In order to reduce the number of tests needed, we have came up with two different decision trees based on previous experience (Nivre and Hall, 2010). The first one, shown in Figure 1, tests only projective algorithms in such a way that the maximum number of tests is 3, and the procedure avoids unnecessary tests such as testing the Nivre arc-standard algorithm when the Nivre arc-eager algorithm provides better results than the projective Stack algorithm. The reason we can omit this test is that the Nivre arc-standard algorithm uses the same parsing order as the projective Stack algorithm. The second decision tree, shown in Figure 2, is for non-projective algorithms and results in a maximum of 5 tests using similar considerations.

After traversing one or both of these decision trees with default settings, MaltOptimizer tunes the parameters of the best performing algorithm and creates a new option file for the best configuration so far. The user is again given the opportunity to edit the option file (or stop the process) before optimization continues.

[4] http://ilk.uvt.nl/conll/software.html

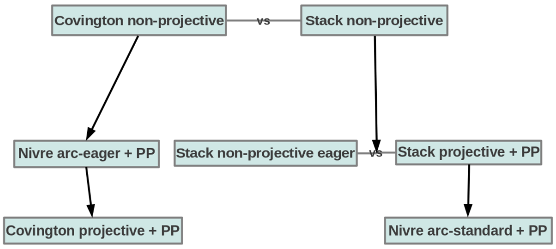

Figure 2: Decision tree for best non-projective algorithm (+PP for pseudo-projective parsing).

3.3. Phase 3: Feature Selection

In the third phase, MaltOptimizer tries to optimize the feature model given the parameters chosen so far (in particular the parsing algorithm). It first performs backward selection experiments to ensure that all features in the default model for the given parsing algorithm actually make a contribution. It then proceeds with forward selection experiments, trying potentially useful features one by one and in combination. An exhaustive search for the best possible feature model is practically impossible, so the optimization strategy is based on heuristics derived from proven experience (Nivre and Hall, 2010). The major steps of the forward selection experiments are the following:[5]

1. Tune the window of POSTAG (n-gram) features over the stack and buffer.
2. Tune the window of (lexical) FORM features over the stack and buffer.
3. Tune dependency tree features using DEPREL and POSTAG features.
4. Add predecessor and successor features for salient tokens using POSTAG and FORM features.
5. Add CPOSTAG, FEATS, and LEMMA features if available.
6. Add conjunctions of POSTAG and FORM features.

Our algorithm traverses all the 6 steps adding one feature at a time and keeping the feature set that provides the best result so far in a greedy fashion. We speed up the process by applying the following heuristics:

1. If backward selection provides improvements for a specific window, we do not try forward selection for this window.
2. As soon as forward selection is unsuccessful for a specific window, we do not try further forward selection experiments for this window.

It is worth to mentioning that these six steps are slightly different depending on which algorithm is the best with default settings, because the MaltParser algorithms have different parsing orders and use different data structures, but the steps are roughly equivalent at a certain level of abstraction. After the feature selection experiments are completed, MaltOptimizer creates a new option file and a new feature specification file. The user is given the opportunity to edit both of these files (or stop the process) before optimization continues.

At the end of the third phase, MaltOptimizer tunes the C parameter of the multiclass SVM using a simple grid search. After this optimization is completed, MaltOptimizer creates the final option file. The user may now continue to do further optimization manually.

4. Experiments

In order to assess the usefulness and validity of the optimization procedure, we have run all three phases of the optimization on all the data sets from the CoNLL 2006 and 2007 shared tasks on dependency parsing (Buchholz and Marsi, 2006; Nivre et al., 2007). We used 5-fold cross-validation for all training sets smaller than 90,000 words and a simple train-devtest split for the larger training sets. Table 1 shows the labeled attachment scores with default settings and after each of the three optimization phases, as well as the difference between the final configuration and the default. The last two columns compare the accuracy obtained on the final test set (Test-MO) with the best score obtained with manual optimization of MaltParser (Test-MP) in the original shared tasks (Nivre et al., 2006b; Hall et al., 2007).

The first thing to note is that the optimization improves parsing accuracy for all languages without exception, although the amount of improvement varies considerably from about 1 percentage point for Chinese, Japanese and Swedish to 7–10 points for Basque, Dutch, Czech, Hungarian and Turkish. Part of the explanation for these differences is the fact that some data sets include rich linguistic annotation, which makes it beneficial to enrich the feature model. This is also why, for most languages, the greatest improvement comes from feature selection in phase 3. However, we also see significant improvement from phase 2 for languages with a substantial amount of non-projective dependencies, such as Czech, Dutch and Slovene, where the selection of parsing algorithm is quite important.

Turning to the final test results, we see that MaltOptimizer fares quite well compared to the manually optimized version of MaltParser, with an average difference of only 0.61 and a maximum (negative) difference of 1.87 (for Catalan 2007). Moreover, we see that in 5 cases out of 23, MaltOptimizer actually improves on the old results, and in a few cases with a quite substantial margin. In fact, in the 2006 shared task, MaltOptimizer would have finished third, beaten only by MSTParser (McDonald et al., 2006) and MaltParser (Nivre et al., 2006b).

The time needed to run the optimization depends primarily on the size of the training set and the validation method chosen. With a simple train-devtest split, it ranges from about half an hour for the smallest data set (Slovene) to about one day for the largest one (Czech 2006). With 5-fold cross-validation, it will basically take five times longer.

[5] For an explanation of the different feature columns such as POSTAG, FORM, etc., see Buchholz and Marsi (2006) and Section 2.

CoNLL-X Shared Task							
Language	**Default**	**Phase 1**	**Phase 2**	**Phase 3**	**Diff**	**Test-MO**	**Test-MP**
Arabic*	63.02	63.03	64.03	66.37	3.35	66.20	**66.71**
Bulgarian	83.19	83.19	84.00	86.03	2.84	86.44	**87.41**
Chinese	84.14	84.14	84.95	84.95	0.81	85.49	**86.92**
Czech	69.94	70.14	72.44	78.04	8.10	**80.46**	78.42
Danish	81.01	81.01	81.34	83.86	2.85	83.41	**84.77**
Dutch	74.77	74.77	78.02	82.63	7.86	77.23	**78.59**
German	82.36	82.36	83.56	85.91	3.55	85.24	**85.82**
Japanese	89.70	89.70	90.92	90.92	1.22	90.39	**91.65**
Portuguese	84.11	84.31	84.75	86.52	2.41	85.85	**87.60**
Slovene*	66.08	66.52	67.86	72.29	6.21	**73.66**	70.30
Spanish*	76.45	76.45	76.64	79.65	3.20	80.18	**81.29**
Swedish	83.34	83.34	83.50	84.09	0.75	83.81	**84.58**
Turkish*	57.79	57.79	58.33	67.11	9.32	64.85	**65.68**

CoNLL 2007 Shared Task							
Language	**Default**	**Phase 1**	**Phase 2**	**Phase 3**	**Diff**	**Test-MO**	**Test-MP**
Arabic	67.71	67.75	67.75	70.77	3.06	73.22	**74.75**
Basque*	67.69	67.83	68.29	75.05	7.36	72.19	**74.99**
Catalan	83.07	83.07	83.13	84.89	1.82	85.87	**87.74**
Chinese	84.04	84.04	85.03	86.21	2.17	82.58	**83.51**
Czech	70.25	70.51	72.49	77.71	7.46	**78.03**	77.22
English	83.84	83.84	85.34	86.61	2.77	85.17	85.81
Greek*	71.01	71.09	72.41	75.12	4.11	**74.50**	74.21
Hungarian	66.42	66.42	68.21	76.53	10.11	77.17	**78.09**
Italian*	79.07	79.07	79.45	81.53	2.46	**82.79**	82.48
Turkish*	67.45	68.38	70.67	76.91	9.46	78.93	**79.24**

Table 1: Labeled attachment score per phase compared to default settings for all training sets from the CoNLL-X shared task (Buchholz and Marsi, 2006) and the CoNLL 2007 Shared task. Languages marked * have a training set smaller than 90,000 tokens and have been optimized using 5-fold cross-validation; the remaining languages have been optimized using a simple train-devtest split. The last two columns report labeled attachment score on the final test sets for the best model found by MaltOptimizer (Test-MO) and the best MaltParser model in the original shared tasks (Test-MP) as reported in Nivre et al. (2006b) and Hall et al. (2007).

5. Related Work

Automatic feature selection for transition-based dependency parsing was recently explored in a study by Nilsson and Nugues (2010). Starting from a much more reduced feature model than the MaltParser default model, they add features incrementally using a notion of topological neighbors in the feature space. They also experiment with different levels of greediness and report competitive results on three different data sets.

In natural language processing more generally, optimization problems have been studied by Kool et al. (2000) and Daelemans et al. (2003), who use genetic algorithms for model selection in the context of part-of-speech tagging, grapheme-to-phoneme conversion with stress assignment, and word sense disambiguation. They study feature selection together with parameter optimization, trying to reach joint optima. A tool developed specifically for the optimization of learning algorithm parameters in the context of natural language processing is Paramsearch (Van den Bosch, 2004).

In machine learning more generally, the feature selection problem has been the object of numerous studies (Guyon and Elisseeff, 2003; McCallum, 2003), and greedy methods like the ones employed by MaltOptimizer are well represented in the literature. For instance, Korycinski et al. (2003) use an adaptive greedy feature selection technique to solve the hyperspectral data analysis in the optical engineering area. Doraisamy et al. (2008) present a comparative study on feature selection techniques applied to automatic genre classification, including best first, greedy stepwise and genetic adaptative search, and conclude that all these methods can significantly improve the performance of general machine learning models. Pahikkala et al. (2010) show how to speed up forward feature selection by applying greedy search, using a strategy similar to that employed by MaltOptimizer. Finally, Das and Kempe (2011) demonstrate that greedy algorithms perform well even when the features are highly correlated, which is something that definitely happens in transition-based dependency parsing.

6. Conclusion

We have presented MaltOptimizer, an optimization tool for MaltParser that can support developers in adapting the system to new languages or domains. We have demonstrated that by using MaltOptimizer it is possible to get substantial improvements over the default settings, thereby allowing non-experts in dependency parsing to achieve high (if not optimal) accuracy, which is usually not possible when using the system “out of the box”. In addition to application developers, MaltOptimizer should also be useful for people doing parsing research who want to use MaltParser as a point of comparison for their own systems, since comparisons with default settings can be highly misleading. MaltOptimizer is an interactive system that allows the user to influence the optimization process at various points, which should make the system potentially useful also for expert users. Future directions may involve the development of a more advanced optimization strategy that interleaves the optimization of parsing algorithm, feature model and learning algorithm instead of using a greedy stepwise approach.

Acknowledgments

The first author is funded by the Spanish Ministry of Education and Science (TIN2009-14659-C03-01 Project), Universidad Complutense de Madrid and Banco Santander Central Hispano (GR58/08 Research Group Grant) and he is under the support of Natural Interaction Based on Language Group from the same University.

7. References

Miguel Ballesteros and Joakim Nivre. 2012. MaltOptimizer: An Optimization Tool for MaltParser. In *Proceedings of the System Demonstration Session of the Thirteenth Conference of the European Chapter of the Association for Computational Linguistics (EACL)*.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 149–164.

Chih-Chung Chang and Chih-Jen Lin, 2001. *LIBSVM: A Library for Support Vector Machines*. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

Michael A. Covington. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585.

Walter Daelemans, Véronique Hoste, Fien De Meulder, and Bart Naudts. 2003. Combined optimization of feature selection and algorithm parameters in machine learning of language. In *Proceedings of the 14th European Conference on Machine Learning (ECML)*, pages 84–95.

Abhimanyu Das and David Kempe. 2011. Submodular meets spectral: Greedy algorithms for subset selection, sparse approximation and dictionary selection. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 1057–1064.

Shyamala Doraisamy, Shahram Golzari, Noris Mohd. Norowi, Md Nasir Sulaiman, and Nur Izura Udzir. 2008. A study on feature selection and classification techniques for automatic genre classification of traditional malay music. In *Proceedings of the Ninth International Conference on Music Information Retrieval (ISMIR)*, pages 331–336.

Rong-En Fan, 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.

Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182.

Johan Hall, Jens Nilsson, Joakim Nivre, Gülsen Eryiğit, Beáta Megyesi, Mattias Nilsson, and Markus Saers. 2007. Single malt or blended? A study in multilingual parser optimization. In *Proceedings of the CoNLL Shared Task of EMNLP-CoNLL 2007*, pages 933–939.

Dan Klein and Christopher D. Manning. 2002. Conditional structure versus conditional estimation in NLP models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9–16.

Anne Kool, Jakub Zavrel, and Walter Daelemans. 2000. Simultaneous feature selection and parameter optimization for memory-based natural language processing. In *Proceedings of the Tenth Belgian-Dutch Conference on Machine Learning (BENELEARN)*, pages 93–100.

Donna Korycinski, Melba Crawford, J. Wesley Barnes, and Joydeep Ghosh. 2003. Adaptive feature selection for hyperspectral data analysis. In *Proceedings of the SPIE Conference on Image and Signal Processing for Remote Sensing IX*, pages 213–225.

Andrew McCallum. 2003. Efficiently inducing features of conditional random fields. In *Proceedings of the Conference on Uncertainty in AI*, pages 403–410.

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 91–98.

Ryan McDonald, Kevin Lerman, and Fernando Pereira. 2006. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 216–220.

Peter Nilsson and Pierre Nugues. 2010. Automatic discovery of feature sets for dependency parsing. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 824–832.

Joakim Nivre and Johan Hall. 2010. A quick guide to MaltParser optimization. Technical report, maltparser.org.

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

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL)*, pages 49–56.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006a. Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, pages 2216–2219.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülsen Eryiğit, and Svetoslav Marinov. 2006b. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 221–225.

Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task of EMNLP-CoNLL 2007*, pages 915–932.

Joakim Nivre, Marco Kuhlmann, and Johan Hall. 2009. An improved oracle for dependency parsing with online reordering. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09)*, pages 73–76.

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

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together (ACL)*, pages 50–57.

Joakim Nivre. 2006. *Inductive Dependency Parsing*. Springer.

Joakim Nivre. 2007. Incremental non-projective dependency parsing. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT)*, pages 396–403.

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

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL-IJCNLP)*, pages 351–359.

Tapio Pahikkala, Antti Airola, and Tapio Salakoski. 2010. Speeding up greedy forward selection for regularized least-squares. In *The Ninth International Conference on Machine Learning and Applications*, pages 325–330.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440.

Antal Van den Bosch. 2004. Wrapped progressive sampling search for optimizing learning algorithm parameters. In *Proceedings of the 16th Belgian-Dutch Conference on Artificial Intelligence*.