MaltOptimizer: An Optimization Tool for MaltParser

Miguel Ballesteros
Complutense University of Madrid
Spain
miballes@fdi.ucm.es

Joakim Nivre
Uppsala University
Sweden
joakim.nivre@lingfil.uu.se

Abstract

Data-driven systems for natural language processing have the advantage that they can easily be ported to any language or domain for which appropriate training data can be found. However, many data-driven systems require careful tuning in order to achieve optimal performance, which may require specialized knowledge of the system. We present MaltOptimizer, a tool developed to facilitate optimization of parsers developed using MaltParser, a data-driven dependency parser generator. MaltOptimizer performs an analysis of the training data and guides the user through a three-phase optimization process, but it can also be used to perform completely automatic optimization. Experiments show that MaltOptimizer can improve parsing accuracy by up to 9 percent absolute (labeled attachment score) compared to default settings. During the demo session, we will run MaltOptimizer on different data sets (user-supplied if possible) and show how the user can interact with the system and track the improvement in parsing accuracy.

1 Introduction

In building NLP applications for new languages and domains, we often want to reuse components for tasks like part-of-speech tagging, syntactic parsing, word sense disambiguation and semantic role labeling. From this perspective, components that rely on machine learning have an advantage, since they can be quickly adapted to new settings provided that we can find suitable training data. However, such components may require careful feature selection and parameter tuning in order to give optimal performance, a task that can be difficult for application developers without specialized knowledge of each component.

A typical example is MaltParser (Nivre et al., 2006), a widely used transition-based dependency parser with state-of-the-art performance for many languages, as demonstrated in the CoNLL shared tasks on multilingual dependency parsing (Buchholz and Marsi, 2006; Nivre et al., 2007). MaltParser is an open-source system that offers a wide range of parameters for optimization. It implements nine different transition-based parsing algorithms, each with its own specific parameters, and it has an expressive specification language that allows the user to define arbitrarily complex feature models. 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 therefore very likely to result in suboptimal performance. However, selecting the best combination of parameters is a complicated task that requires knowledge of the system as well as knowledge of the characteristics of the training data.

This is why we present MaltOptimizer, a tool for optimizing MaltParser for a new language or domain, based on an analysis of the training data. The optimization is performed in three phases: data analysis, parsing algorithm selection, and feature selection. The tool can be run in "batch mode" to perform completely automatic optimization, but it is also possible for the user to manually tune parameters after each of the three phases. In this way, we hope to cater for users without specific knowledge of MaltParser, who can use the tool for black box optimization, as well as expert users, who can use it interactively to speed up optimization. Experiments on a number of data sets show that using MaltOptimizer for completely automatic optimization gives consistent and often substantial improvements over the default settings for MaltParser.

The importance of feature selection and parameter optimization has been demonstrated for many NLP tasks (Kool et al., 2000; Daelemans et al., 2003), and there are general optimization tools for machine learning, such as Paramsearch (Van den Bosch, 2004). In addition, Nilsson and Nugues (2010) has explored automatic feature selection specifically for MaltParser, but MaltOptimizer is the first system that implements a complete customized optimization process for this system.

In the rest of the paper, we describe the optimization process implemented in MaltOptimizer (Section 2), report experiments (Section 3), outline the demonstration (Section 4), and conclude (Section 5). A more detailed description of MaltOptimizer with additional experimental results can be found in Ballesteros and Nivre (2012).

2 The MaltOptimizer System

MaltOptimizer is written in Java and implements an 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 CoNLL data format,[1] and outputs an optimized MaltParser configuration together with an estimate of the final parsing accuracy. The evaluation metric that is used for optimization by default is the labeled attachment score (LAS) excluding punctuation, that is, the percentage of non-punctuation tokens that are assigned the correct head and the correct label (Buchholz and Marsi, 2006), but other options are available. For efficiency reasons, MaltOptimizer only explores linear multiclass SVMs in LIBLINEAR.

2.1 Phase 1: Data Analysis

After validating that the data is in valid CoNLL format, using the official validation script from the CoNLL-X shared task,[2] the system checks the minimum Java heap space needed given the size of the data set. If there is not enough memory available on the current machine, the system informs the user and automatically reduces the size of the data set to a feasible subset. After these initial checks, MaltOptimizer checks the following characteristics of the data set:

1. Number of words/sentences.
2. Existence of “covered roots” (arcs spanning tokens with HEAD = 0).
3. Frequency of labels used for tokens with HEAD = 0.
4. Percentage of non-projective arcs/trees.
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.

Items 1–3 are used to set basic parameters in the rest of phase 1 (see below); 4 is used in the choice of parsing algorithm (phase 2); 5 and 6 are relevant for feature selection experiments (phase 3).

If there are covered roots, the system checks whether accuracy is improved by reattaching such roots in order to eliminate spurious non-projectivity. If there are multiple labels for tokens with HEAD=0, the system tests which label is best to use as default for fragmented parses.

Given the size of the data set, the system suggests different validation strategies during phase 1. If the data set is small, it recommends using 5-fold cross-validation during subsequent optimization phases. If the data set is larger, it recommends using a single development set instead. But the user can override either recommendation and select either validation method manually.

When these checks are completed, MaltOptimizer creates a baseline option file to be used as the starting point for further optimization. The user is given the opportunity to edit this option file and may also choose to stop the process and continue with manual optimization.

2.2 Phase 2: Parsing Algorithm Selection

MaltParser implements three groups of transition-based parsing algorithms:[3] (i) Nivre's algorithms (Nivre, 2003; Nivre, 2008), (ii) Covington's algorithms (Covington, 2001; Nivre, 2008), and (iii)

[1]http://ilk.uvt.nl/conll/#dataformat

[2]http://ilk.uvt.nl/conll/software.html#validate

[3]Recent versions of MaltParser contains additional algorithms that are currently not handled by MaltOptimizer.

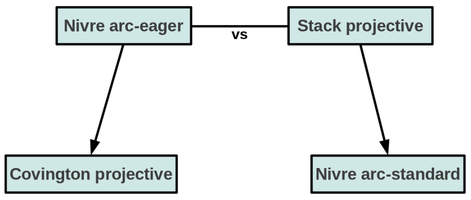

Figure 1: Decision tree for best projective algorithm.

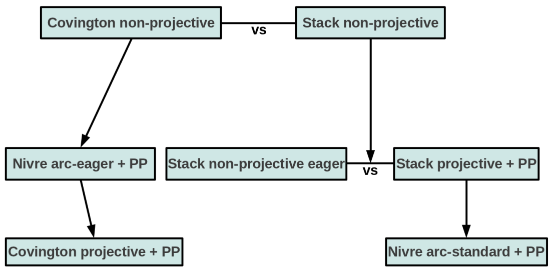

Figure 2: Decision tree for best non-projective algorithm (+PP for pseudo-projective parsing).

Stack algorithms (Nivre, 2009; Nivre et al., 2009) Both the Covington group and the Stack group contain algorithms that can handle non-projective dependency trees, and any projective algorithm can be combined with pseudo-projective parsing to recover non-projective dependencies in post-processing (Nivre and Nilsson, 2005).

In phase 2, MaltOptimizer explores the parsing algorithms implemented in MaltParser, based on the data characteristics inferred in the first phase. 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, the projective version of Covington's projective parsing algorithm and the projective Stack algorithm. The system follows a decision tree considering the characteristics of each algorithm, which is shown in Figure 1.

On the other hand, if the training set contains a substantial amount of non-projective dependencies, MaltOptimizer instead tests the non-projective versions of Covington's algorithm and the Stack algorithm (including a lazy and an eager variant), and projective algorithms in combination with pseudo-projective parsing. The system then follows the decision tree shown in Figure 2.

If the number of trees containing non-projective arcs is small but not zero, the system tests both projective algorithms and non-projective algorithms, following the decision trees in Figure 1 and Figure 2 and picking the algorithm that gives the best results after traversing both.

Once the system has finished testing each of the algorithms with default settings, MaltOptimizer tunes some specific 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.

2.3 Phase 3: Feature Selection

In the third phase, MaltOptimizer tunes the feature model given all the parameters chosen so far (especially the parsing algorithm). It starts with backward selection experiments to ensure that all features in the default model for the given parsing algorithm are actually useful. In this phase, features are omitted as long as their removal does not decrease parsing accuracy. The system then proceeds with forward selection experiments, trying potentially useful features one by one. In this phase, a threshold of 0.05% is used to determine whether an improvement in parsing accuracy is sufficient for the feature to be added to the model. Since an exhaustive search for the best possible feature model is impossible, the system relies on a greedy optimization strategy using heuristics derived from proven experience (Nivre and Hall, 2010). The major steps of the forward selection experiments are the following:[4]

1. Tune the window of POSTAG n-grams over the parser state.
2. Tune the window of FORM features over the parser state.
3. Tune DEPREL and POSTAG features over the partially built dependency tree.
4. Add POSTAG and FORM features over the input string.
5. Add CPOSTAG, FEATS, and LEMMA features if available.
6. Add conjunctions of POSTAG and FORM features.

These six steps are slightly different depending on which algorithm has been selected as the best in phase 2, because the algorithms have different parsing orders and use different data structures,

[4]For an explanation of the different feature columns such as POSTAG, FORM, etc., see Buchholz and Marsi (2006) or see http://ilk.uvt.nl/conll/#dataformat

Language	Default	Phase 1	Phase 2	Phase 3	Diff
Arabic	63.02	63.03	63.84	65.56	2.54
Bulgarian	83.19	83.19	84.00	86.03	2.84
Chinese	84.14	84.14	84.95	84.95	0.81
Czech	69.94	70.14	72.44	78.04	8.10
Danish	81.01	81.01	81.34	83.86	2.85
Dutch	74.77	74.77	78.02	82.63	7.86
German	82.36	82.36	83.56	85.91	3.55
Japanese	89.70	89.70	90.92	90.92	1.22
Portuguese	84.11	84.31	84.75	86.52	2.41
Slovene	66.08	66.52	68.40	71.71	5.63
Spanish	76.45	76.45	76.64	79.38	2.93
Swedish	83.34	83.34	83.50	84.09	0.75
Turkish	57.79	57.79	58.29	66.92	9.13

Table 1: Labeled attachment score per phase and with comparison to default settings for the 13 training sets from the CoNLL-X shared task (Buchholz and Marsi, 2006).

but the steps are roughly equivalent at a certain level of abstraction. After the feature selection experiments are completed, MaltOptimizer tunes the cost parameter of the linear SVM using a simple stepwise search. Finally, it creates a complete configuration file that can be used to train MaltParser on the entire data set. The user may now continue to do further optimization manually.

3 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 13 data sets from the CoNLL-X shared task on multilingual dependency parsing (Buchholz and Marsi, 2006). 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.[5]

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 8–9 points for Dutch, Czech and Turkish. For most languages, the greatest improvement comes from feature selection in phase 3, but 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 can be very important. The time needed to run the optimization varies from about half an hour for the smaller data sets to about one day for very large data sets like the one for Czech.

4 System Demonstration

In the demonstration, we will run MaltOptimizer on different data sets and show how the user can interact with the system while keeping track of improvements in parsing accuracy. We will also explain how to interpret the output of the system, including the final feature specification model, for users that are not familiar with MaltParser. By restricting the size of the input data set, we can complete the whole optimization procedure in 10–15 minutes, so we expect to be able to complete a number of cycles with different members of the audience. We will also let the audience contribute their own data sets for optimization, provided that they are in CoNLL format.[6]

5 Conclusion

MaltOptimizer is an optimization tool for MaltParser, which is primarily aimed at application developers who wish to adapt the system to a new language or domain and who do not have expert knowledge about transition-based dependency parsing. Another potential user group consists of researchers who want to perform comparative parser evaluation, where MaltParser is often used as a baseline system and where the use of suboptimal parameter settings may undermine the validity of the evaluation. Finally, we believe the system can be useful also for expert users of MaltParser as a way of speeding up optimization.

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). He is under the support of the NIL Research Group (http://nil.fdi.ucm.es) from the same university.

[5]Note that these results are obtained using 80% of the training set for training and 20% as a development test set, which means that they are not comparable to the test results from the original shared task, which were obtained using the entire training set for training and a separate held-out test set for evaluation.

[6]The system is available for download under an open-source license at http://nil.fdi.ucm.es/maltoptimizer

References

Miguel Ballesteros and Joakim Nivre. 2012. MaltOptimizer: A System for MaltParser Optimization. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC)*.

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.

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 Nada Lavrac, Dragan Gamberger, Hendrik Blockeel, and Ljupco Todorovski, editors, *Machine Learning: ECML 2003*, volume 2837 of *Lecture Notes in Computer Science*. Springer.

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

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

Peter Nilsson and Pierre Nugues. 2010. Automatic discovery of feature sets for dependency parsing. In *COLING*, 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. 2006. 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, 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. 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.

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*.