# A Quick Guide to MaltParser Optimization

Joakim Nivre Johan Hall

## 1 Introduction

MaltParser is a system for data-driven dependency parsing, which can be used to induce a parsing model from treebank data and to parse new data using an induced model. Parsers developed using MaltParser have achieved state-of-the-art accuracy for a number of languages (Nivre et al., 2006; Hall et al., 2007; Nivre et al., 2007). However, MaltParser is a fairly complex system with many parameters that need to be optimized. Simply using the system "out of the box" with default settings is therefore likely to result in suboptimal performance.

The purpose of this document is to give a practical guide to the steps needed to reach good performance, starting from the default settings and tuning (i) the parsing algorithm, (ii) the learning algorithm, and (iii) the feature model. It is by no means an exhaustive description of all the options available in MaltParser and is only meant as an introduction. For complete information about all parameters and settings in MaltParser, we refer to the documentation at `http://maltparser.org`. The MaltParser website also contains information about settings used in published experiments, notably in the CoNLL shared tasks 2006 and 2007 (Buchholz and Marsi, 2006; Nivre et al., 2007), as well as pretrained models for a number of languages.

## 2 Data Requirements

The training data for MaltParser consists of sentences annotated with dependency trees. If the source treebank does not contain dependency trees, there are usually standard procedures for conversion that can be used (with varying degrees of precision depending on the nature of the original annotation). MaltParser supports several input formats, but in this document we will assume that data is in the CoNLL representation format (Buchholz and Marsi, 2006), which is currently the de facto standard for data-driven dependency parsing.

Optimization requires repeated testing on held-out data using either cross-validation or a separate development test set. Based on our own experience, we recommend using cross-validation for training sets up to 200,000 tokens. For larger training sets, a single training-test split is usually sufficient provided that at least 10% of the data is set aside for development testing. Regardless of whether cross-validation or a development test set is used for optimization, it is good experimental practice to reserve a separate data set for final evaluation.

# 3 Parsing Algorithm

MaltParser is based on the transition-based approach to dependency parsing, which means that a parser consists of a *transition system* for deriving dependency trees, coupled with a *classifier* for deterministically predicting the next transition given a feature representation of the current parser configuration (Nivre, 2008). In order to derive training data for the classifier, an *oracle* is used to reconstruct a valid transition sequence for every dependency structure in the training set. A transition system together with an oracle defines a *parsing algorithm*. MaltParser 1.4.1 is equipped with nine different parsing algorithms, which differ not only in terms of transitions and oracles but also with respect to the class of dependency structures they can handle. Table 1 lists the option values for the option `parsing_algorithm` (`-a`) for all nine algorithms, together with information about the class of dependency structures they cover and pointers to the literature.

Table 1: Parsing algorithms in MaltParser 1.4.1.

| **parsing_algorithm (–a)** | **Structures** | **References** |
|---|---|---|
| nivreeager | Projective | Nivre (2003), Nivre (2008) |
| nivrestandard | Projective | Nivre (2004), Nivre (2008) |
| covproj | Projective | Nivre (2006), Nivre (2008) |
| covnonproj | Non-Projective | Nivre (2006), Nivre (2007), Nivre (2008) |
| stackproj | Projective | Nivre (2009) |
| stackeager | Non-Projective | Nivre (2009) |
| stacklazy | Non-Projective | Nivre et al. (2009) |
| planar | Planar | Gómez-Rodríguez and Nivre (2010) |
| 2-planar | 2-Planar | Gómez-Rodríguez and Nivre (2010) |

## 3.1 Choosing a Parsing Algorithm

A useful first step in the optimization of MaltParser to a new language or data set is to try out different parsing algorithms, by varying the value of the option `parsing_algorithm` (`-a`). Although the performance of a given algorithm is dependent also on the choice of learning algorithm and feature model, a comparison of parsing algorithms under otherwise default settings is usually indicative of which algorithms are likely to give good performance on the given data set. Minimally, the default algorithm `nivreeager` should be compared to `stackproj`, since one of these two algorithms often perform best.

**Note:** Some of the parsing algorithms have options of their own that can be tuned. See the MaltParser documentation for more information.

### 3.2 Handling Non-Projective Structures

If the training data contains (a non-negligible proportion of) non-projective dependency structures, parsing accuracy can usually be improved in one of two ways.

- Use a parsing algorithm for non-projective structures (`covnonproj`, `stackeager`, `stacklazy`, `2planar`). The rule of thumb here is that if `nivreeager` performs better than `stackproj` in the first comparison, then `covnonproj` is likely to be the best choice; if the results are reversed, then `stacklazy` is probably the better option.
- Use a parsing algorithm for projective structures (`nivreeager`, `nivrestandard`, `covproj`, `stackproj`) but combine it with pseudo-projective pre- and post-processing (Nivre and Nilsson, 2005). Add the option `marking_strategy` (`-pp`) with the value `head`, `path` or `head+path` to try this out.

**Note:** If pseudo-projective parsing turns out to work best, there are a few additional options besides `marking_strategy` that can be tuned, but the effect is usually marginal.[1] See the MaltParser documentation for more information.

## 4 Learning Algorithm

The learning problem in transition-based parsing, as implemented in MaltParser, is to induce a classifier for predicting the next transition given a feature representation of the current parser configuration, using as training data the transitions needed to derive the dependency structures in the training set relative to a given oracle. MaltParser 1.4.1 has two built-in machine learning packages for this task: LIBSVM and LIBLINEAR (see Table 2).

Table 2: Learning algorithms in MaltParser 1.4.1.

| **learner (–l)** | Learner Types | **References** |
|---|---|---|
| libsvm | Support vector machines (with kernels) | Chang and Lin (2001) |
| liblinear | Various linear classifiers (including SVMs) | Fan et al. (2008) |

### 4.1 Choosing a Learning Algorithm

Both LIBSVM and LIBLINEAR can give state-of-the-art accuracy and the choice between them is primarily dictated by practical constraints having to do with optimization effort and efficiency constraints. In particular:

[1] The exception is the option `covered_roots`, which can sometimes be quite important and which is discussed in Section 6 below.

- LIBSVM makes feature optimization a little easier thanks to the kernel that implicitly add conjoined features, whereas LIBLINEAR requires conjoined features to be added explicitly to the feature model specification (see Section 5).
- LIBSVM is usually more memory efficient than LIBLINEAR because it does not store weight vectors explicitly.
- LIBLINEAR is much faster than LIBSVM, both for training and parsing, and parsing speed depends only on the number of features, whereas for LIBSVM it is proportional to the number of training instances in the worst case.

We therefore recommend using LIBSVM for quick feature optimization or when memory consumption is an issue, but LIBLINEAR in all other circumstances.

**Note:** Both LIBSVM and LIBLINEAR include a large number of options that can be tuned, including different learner types as well as hyperparameters for each learner type, but MaltParser's default settings for each package – a quadratic kernel for LIBSVM and a multi-class SVM for LIBLINEAR – usually gives competitive performance. See the MaltParser documentation for more information.

### 4.2 Splitting the Training Data

Instead of training a single classifier on the entire training set, MaltParser provides functionality for dividing the training set into equivalence classes with respect to some salient feature (such as the part of speech of the next input token) and training separate classifiers on each subset. At parsing time, the same feature is used to decide which classifier to consult. It is important to note that the effect of using this functionality is completely different depending on which machine learning package is used:

- With LIBSVM it is primarily a technique for speeding up both training and parsing with large training sets, for which training times can otherwise be prohibitive. On the negative side, it may lead to a small drop in accuracy (partly because tagging errors may lead to the wrong classifier being used at parsing time).
- With LIBLINEAR it is primarily a technique for improving parsing accuracy by increasing the separability of the training data, essentially emulating a one-level decision tree with linear classifiers at the leaf nodes. On the negative side, it will lead to a small drop in efficiency and may increase memory usage. Moreover, the positive effect on accuracy is usually small if the feature model has been properly optimized (see Section 5).

In order to use the data splitting functionality, three options must be specified:

- `data_split_column (-d)` – the input feature used for the split, usually `POSTAG` in the CoNLL format.

- `data_split_structure (-s)` – the word token used for the split, defined relative to the parser configuration, usually `Input[0]` for `nivreeager`, `nivreeager`, `planar` and `2planar`; `Stack[0]` for `stackproj`, `stackeager` and `stacklazy`; and `Right[0]` for `covproj` and `covnonproj`.
- `data_split_threshold (-T)` – the minimum frequency for a feature value to define its own equivalence class, usually set to `1000`.

**Note:** Splitting the training data is in practically necessary when using LIBSVM on training sets that contain on the order of 1 million words (like the training sets of the Prague Dependency Treebank and the Penn Treebank), where training can otherwise take weeks or months, but optional with LIBLINEAR and with small training sets.

# 5 Feature Model

Most of the effort when optimizing MaltParser usually goes into feature selection, that is, in tuning the feature representation of a parser configuration that constitutes the input to the classifier. A feature model in MaltParser is defined by a feature specification file in XML format, and the option `features (-F)` is used to specify which file should be used. The default specification files for each combination of parsing algorithm and machine learning package, which are used if no other file is specified, can be found in the `appdata/features` directory of the MaltParser distribution and constitute good starting points for further optimization. In the rest of this section, we will give step-by-step instructions for how to optimize the feature model. We will start by assuming that the model is to be used together with LIBSVM, with a kernel that creates conjoined features implicitly. We will then go on to describe how such a model can be adapted for use with LIBLINEAR. We will concentrate on the conceptual content of feature models and refer the reader to the MaltParser documentation for the formal syntax of feature specifications.

## 5.1 Optimizing a Feature Model for LIBSVM

A default feature model essentially consists of three groups of features, all of which are centered around the two *target tokens*, i.e., the tokens that are being considered for a dependency in the current configuration:[2]

- Part-of-speech tags (column POSTAG) in a wide window around the target tokens.
- Word forms (column FORM) in a narrower window around the target tokens.
- Dependency labels (column DEPREL) on outgoing arcs of the target tokens.

[2]The target tokens are referred to by `Stack[0]` (top of the stack) and `Input[0]` (next input token) for the parsing algorithms `nivreeager`, `nivrestandard`, `planar` and `2planar`; by `Stack[1]` and `Stack[0]` (the two top tokens of the stack) for `stackproj`, `stackeager` and `stacklazy`; and by `Left[0]` and `Right[0]` for `covproj` and `covnonproj`.

The optimal window size for part-of-speech tags and word forms may differ depending on training set size and therefore needs to be tuned using backward feature selection (shrinking the window) and forward feature selection (expanding the window). The usefulness of dependency label features can also be checked using backward selection, but this is usually less important. Finally, if the training data contains additional token attributes such as lemmas (column LEMMA), coarse part-of-speech tags (column CPOSTAG) and morphosyntactic features (column FEATS), it is usually worth adding features over these columns for the two target tokens and possibly one lookahead token.

**Note:** When defining features over the list-valued FEATS column, it is possible to use the `Split` operator to define one binary feature over each atom that may occur in the list (as opposed to having a feature for each distinct list). See the MaltParser documentation for more information.

### 5.2 Optimizing a Feature Model for LIBLINEAR

Using LIBLINEAR with a feature model optimized for LIBSVM will usually result in suboptimal performance, because all features are treated as independent of each other. In order to compensate for the lack of a kernel, it is necessary to add conjoined features using the operators `Merge` (for pairs) and `Merge3` (for triples). Usually, a relatively small number of such features will suffice to reach the same level as the corresponding LIBSVM model:

- Part-of-speech tag features can be conjoined into (overlapping) trigrams covering the same window as the atomic features (e.g., `Merge3(InputColumn(POSTAG, Stack[2]), InputColumn(POSTAG, Stack[1]), InputColumn(POSTAG, Stack[0]))`).
- Word form features can be conjoined with the corresponding part-of-speech tag feature (e.g., `Merge(InputColumn(FORM, Input[0]), InputColumn(POSTAG, Input[0]))`).
- Dependency label features can be conjoined with the part-of-speech tag feature of the parent (e.g., `Merge3(InputColumn(POSTAG, Stack[0]), OutputColumn(DEPREL, ldep(Stack[0])), OutputColumn(DEPREL, rdep(Stack[0])))`.

Although the addition of conjoined features will generally improve parsing accuracy, it will also increase memory requirements. This holds especially for conjunctions involving high-dimensional features such as word form features, which can be very expensive in this respect.

**Note:** MaltParser's feature specification language supports a wide range of different feature types over and above the ones discussed here, such as distance features and suffix features. See the MaltParser documentation for more information.

## 6 Tips and Tricks

In this final section, we discuss a few special phenomena that may cause problems unless MaltParser's options are adjusted appropriately.

- **Dangling punctuation:** If the annotation scheme used in the training data does not attach punctuation as dependents of words, and if this is represented in the CoNLL format by setting `HEAD=0` for punctuation tokens, this may give rise to a lot of spurious non-projective dependencies (dependency arcs spanning a punctuation token). To avoid this, use the option `covered_root (-pcr)` with value `left/right/head`, which makes MaltParser preprocess the data and attach the punctuation tokens to the left/right/head end of the shortest arc spanning the token.
- **Root labels:** After parsing is completed, MaltParser assigns the default label `ROOT` to all tokens having `HEAD=0` that have not been assigned a label by the parser. If the annotation scheme used in the training data has another standard root label, the default root label can be changed using the option `root_label (-grl)`.

## References


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

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

Fan, R.-E., 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.

Gómez-Rodríguez, C. and J. Nivre (2010). A transition-based parser for 2-planar dependency structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pp. 1492–1501.

Hall, J., J. Nilsson, J. Nivre, G. Eryiğit, B. Megyesi, M. Nilsson, and M. Saers (2007). Single malt or blended? A study in multilingual parser optimization. In *Proceedings of the CoNLL Shared Task of EMNLP-CoNLL 2007*.

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

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

Nivre, J. (2006). Constraints on non-projective dependency graphs. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pp. 73–80.

Nivre, J. (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)*, pp. 396–403.

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

Nivre, J. (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)*, pp. 351–359.

Nivre, J., J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret (2007). The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task of EMNLP-CoNLL 2007*, pp. 915–932.

Nivre, J., J. Hall, J. Nilsson, A. Chanev, G. Eryiğit, S. Kübler, S. Marinov, and E. Marsi (2007). Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering 13*, 95–135.

Nivre, J., J. Hall, J. Nilsson, G. Eryiğit, and S. Marinov (2006). Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pp. 221–225.

Nivre, J., M. Kuhlmann, and J. Hall (2009). An improved oracle for dependency parsing with online reordering. 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)*, pp. 73–76.

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