Natural Language Engineering **22** (2): 187–213. © Cambridge University Press 2014
doi:10.1017/S1351324914000035

MaltOptimizer: Fast and effective parser optimization

MIGUEL BALLESTEROS[1] and JOAKIM NIVRE[2]

[1]*Natural Language Processing Group, Pompeu Fabra University, Tànger 122-140, 08018 Barcelona, Spain*
e-mail: miguel.ballesteros@upf.edu
[2]*Department of Linguistics and Philology, Uppsala University, Box 635, 75126 Uppsala, Sweden*
e-mail: joakim.nivre@lingfil.uu.se

(*Received 2 September 2013; revised 27 January 2014; accepted 27 January 2014; first published online 24 February 2014*)

Abstract

Statistical parsers often require careful parameter tuning and feature selection. This is a nontrivial task for application developers who are not interested in parsing for its own sake, and it can be time-consuming even for experienced researchers. In this paper we present MaltOptimizer, a tool developed to automatically explore parameters and features for MaltParser, a transition-based dependency parsing system that can be used to train parser's given treebank data. MaltParser provides a wide range of parameters for optimization, including nine different parsing algorithms, an expressive feature specification language that can be used to define arbitrarily rich feature models, and two machine learning libraries, each with their own parameters. MaltOptimizer is an interactive system that performs parser optimization in three stages. First, it performs an analysis of the training set in order to select a suitable starting point for optimization. Second, it selects the best parsing algorithm and tunes the parameters of this algorithm. Finally, it performs feature selection and tunes machine learning parameters. Experiments on a wide range of data sets show that MaltOptimizer quickly produces models that consistently outperform default settings and often approach the accuracy achieved through careful manual optimization.

1 Introduction

Training statistical parsers to achieve high parsing accuracy often requires specific parameter tuning and in-depth feature selection, especially when porting parsers to new languages or domains. These tasks are not easy for people that lack specific expertise and experience from parsing research, which is often the case for application developers that need a parser for a new language. A full exploration of the parameter and feature space offered by a parsing system may be a daunting task even for experienced researchers. However, it has been shown that different parsing models may yield very different accuracy levels when parsing the same pieces of text. For instance, Hall *et al.* (2007) report differences of more than

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

3% absolute in labeled attachment score (henceforth, LAS) between a baseline version of MaltParser and manually optimized models for some languages during the CoNLL 2007-shared task on dependency parsing. Consequently, parser optimization is often a necessary step to exploit the full potential of a given parsing system.

In this paper, we present MaltOptimizer, an interactive tool for automatic or semi-automatic optimization of parsers developed using the MaltParser system for transition-based dependency parsing. MaltParser (Nivre, Hall and Nilsson 2006b; Nivre *et al.* 2007b) is a widely used transition-based parser-generator that offers a wide range of parameters to tune parsing models. It incorporates the implementation of nine different transition-based parsing algorithms, each of which has its own parameters. Each parsing algorithm defines a set of possible features that the parser may use for selecting the transition that is most likely to occur given the parser state, and it is possible to define arbitrarily complex feature models using an expressive feature specification language. Any combination of parsing algorithm and feature model can furthermore be combined with a number of different machine learning algorithms available in LIBSVM (Chang and Lin 2011) and LIBLINEAR (Fan *et al.* 2008).

MaltOptimizer automates the parser optimization process in three phases and optionally allows the user to influence the process after each phase. In phase 1, MaltOptimizer analyzes the training data set with respect to size, availability of features, and frequency of non-projective dependency trees, among other things, in order to find a good starting point for the optimization. In phase 2, the system explores different parsing algorithms and tunes the parameters of the best performing algorithm. Phase 3, finally, consists of extensive feature selection experiments followed by parameter tuning for the machine learning algorithm. As will be discussed below, MaltOptimizer does not provide any theoretical guarantees that the best possible model will be found, but it is guaranteed to find the model with best development set (or cross-validation) accuracy out of a range of different models tested throughout the process. Experiments show that the model found consistently improves over default settings and often comes quite close to the results obtained with manually optimized models.

The rest of the paper is structured as follows. Section 2 discusses the problem of optimizing statistical parsers in general, and Section 3 defines the specific optimization problem for MaltParser by explaining the transition-based parsing method and the available options and parameters. Section 4 gives a detailed description of the three-phase optimization process implemented in MaltOptimizer. Section 5 reports experiments and their results on data sets from the CoNLL 2006 and 2007-shared tasks on dependency parsing. Section 6 discusses related work, and Section 7 concludes. The material presented in this paper substantially extends that of previous publications (Ballesteros and Nivre 2012a, 2012b) by including a theoretical discussion of the parser optimization problem, a much more exact and detailed description of the optimization process, more extensive experiments, and a deeper analysis of the experimental results. MaltOptimizer is available for download at `http://nil.fdi.ucm.es/maltoptimizer`.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

2 Parser optimization

Any statistical parser has a set of parameters that need to be estimated from a training data set, such as feature weights or rule probabilities. In addition, many parsers have a set of hyper-parameters, which can be manipulated to improve the result of machine learning but are not themselves learned from data. Hyper-parameters include both aspects that define the basic parameter space, such as manually constructed feature templates, and that influence the parameter estimation process, such as the choice of regularizer. And in order to get the best possible performance when training a parser on a new training data set, we need to select hyper-parameters such that the learned model gives the best possible accuracy when applied to new sentences from the same population as the training set. This is what we mean by *parser optimization*.

It is important to note that the parser optimization problem, as defined here, is not a well-defined optimization problem in the mathematical sense. The reason is that the 'objective function' we want to minimize is the true generalization error, that is, the probability that the model will produce an incorrect parse for a sentence drawn from the population (Mitchell 1997)). This is different from, say, the problem of minimizing the sample error on the training set, which is a problem that can (sometimes) be solved exactly using an appropriate machine learning algorithm. The standard way of tackling this problem is therefore to rely on statistical inference and estimate the generalization error using a held-out development test set (or cross-validation) while systematically varying hyper-parameters. This is also the methodology underlying MaltOptimizer.

If the hyper-parameter space is finite (and not too large), we can exhaustively search this space to find the settings that minimize our estimate of the true generalization error. This is the case, for example, for the downloadable version of MSTParser (McDonald 2006), where the user can choose to use either a first- or a second-order model and either projective or non-projective decoding, giving a total space of four possible settings.[1] However, for a system like MaltParser, which among other things allows the user to define arbitrarily complex combinations of feature templates, the space is in principle unbounded and for all practical purposes too large to be searched exhaustively. We therefore have to resort to approximate methods such as grid search for numerical hyper-parameters and greedy forward and backward feature selection, both of which are used in MaltOptimizer. Empirical results show that such methods can be quite effective for parser optimization, but it is important to remember that there are no theoretical guarantees that we will find the setting of hyper-parameters that minimizes error on held-out data (because of search errors), let alone minimizes the true generalization error (because of estimation errors). Still, by basing our optimization procedure on proven experience (Hall *et al.* 2007; Nivre and Hall 2010), we hope to provide a useful tool for system developers.

[1] Here we restrict our attention to hyper-parameters that can be varied without modifying the source code of the system.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Initial configuration $C_i = \langle [\,], [w_1 \ldots w_n], \emptyset, \rangle$

Terminal configuration $C_f \in \{C \mid C = \langle \Sigma, [\,], A \rangle\}$

Transitions:

SHIFT	$\langle \Sigma, i\|B, A\rangle \Rightarrow \langle \Sigma\|i, B, A\rangle$
REDUCE	$\langle \Sigma\|i, B, A\rangle \Rightarrow \langle \Sigma, B, A\rangle$ if $\exists j, r : j \xrightarrow{r} i \in A$
LEFT-ARC(r)	$\langle \Sigma\|i, j\|B, A\rangle \Rightarrow \langle \Sigma, j\|B, A \cup \{j \xrightarrow{r} i\}\rangle$ if $\neg\exists k, r' : (k, r', i) \in A$
RIGHT-ARC(r)	$\langle \Sigma\|i, j\|B, A\rangle \Rightarrow \langle \Sigma\|i\|j, B, A \cup \{i \xrightarrow{r} j\}\rangle$

Fig. 1. Nivre's arc-eager transition system.

The experimental results presented in Section 5 also indicate that MaltOptimizer gives good performance in practice.

3 MaltParser

MaltParser (Nivre 2006b, 2007b) can be described as a data-driven parser-generator, which takes as input a training set consisting of sentences annotated with dependency trees and yields as output a parser for the language of the training set. The parsers generated are said to be *transition-based* because they are based on a search for the best path through a transition system, or state machine. The approach is similar to shift–reduce parsing, but it allows the inclusion of a wider range of transition systems (Nivre 2008). MaltParser is capable of training models that predict the optimal transition to the next parser configuration (or state) using features over previous parsing decisions and the words from the input sentence. At parsing time, the parser starts in an initial configuration and greedily applies the best transition until it reaches a terminal configuration (where there are no more arcs to add to the dependency tree and no more tokens to process). It is worth noting that transition-based parsers are generally quite efficient, often with a running time that is linear in sentence length (Nivre 2008). However, as shown by McDonald and Nivre (2007), the greedy parsing strategy can also lead to error propagation when bad decisions are made early in the parsing process.

Summing up, these are the basic properties of a transition-based parser generated by MaltParser:

- Sentences are processed from left to right in a greedy deterministic fashion.
- At each step, a classifier selects the transition to the next parser configuration.
- The classifier uses a rich set of features, including linguistic properties of the input tokens (parts of speech, lemmas, morphosyntactic features, *etc.*).

By way of example, Figure 1 shows one of the transition systems available in MaltParser, known as Nivre's arc-eager system (Nivre 2003, 2008). In this system, a parser configuration consists of a stack Σ of partially processed tokens, a buffer B of remaining input tokens, and a set A of dependency arcs defining the (partially built) dependency tree. The initial configuration has an empty stack and arc set, and all

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Transition	Stack	Buffer	Added Arc
	[ ]	[She, sent, him, a, letter, .]	
SHIFT	[She]	[sent, him, a, letter, .]	
L-A(SUBJ)	[ ]	[sent, him, a, letter, .]	sent $\xrightarrow{\text{SUBJ}}$ She
SHIFT	[sent]	[him, a, letter, .]	
R-A(IOBJ)	[sent, him]	[a, letter, .]	sent $\xrightarrow{\text{IOBJ}}$ him
REDUCE	[sent]	[a, letter, .]	
SHIFT	[sent, a]	[letter, .]	
L-A(DET)	[sent]	[letter, .]	letter $\xrightarrow{\text{DET}}$ a
R-A(DOBJ)	[sent, letter]	[.]	sent $\xrightarrow{\text{DOBJ}}$ letter
REDUCE	[sent]	[.]	
R-A(PUNC)	[sent, .]	[ ]	sent $\xrightarrow{\text{PUNC}}$.

Fig. 2. Transition sequence for *She sent him a letter*.

the tokens of the sentence in the buffer; a configuration is terminal if it has an empty buffer (in which case the arc set defines the output parse). There are four possible transitions: SHIFT pushes the next input token i onto the stack; REDUCE pops the stack; LEFT-ARC adds an arc from the next input token j to the token i on top of the stack (with some dependency label r) and then pops the stack; RIGHT-ARC adds an arc from the token i on top of the stack to the next input token j (with some dependency label r) and then pushes j onto the stack. Figure 2 shows the sequence of transitions needed to parse an English sentence. The features that can be used to predict each transition will be described in detail in Section 3.2.

As already mentioned, MaltParser has a large number of hyper-parameters that can be tuned for optimal performance. They can be divided into three groups, which will be described in the following sections:

(1) Parsing algorithm (Section 3.1)
(2) Feature model (Section 3.2)
(3) Learning algorithm (Section 3.3)

3.1 Parsing algorithm

In MaltParser, the selection of a parsing algorithm is essentially the choice of a transition system and a set of constraints that are inherent to the transition system. The latest version of MaltParser includes the implementation of nine different transition-based parsing algorithms. These all are based on similar left-to-right parsing strategies, but they differ in how configurations are defined, how new arcs are created, and how non-projective dependencies are handled (if at all). The parsing algorithms available in MaltParser can be grouped into four families:

(1) **Nivre's algorithms** include the *arc-eager* algorithm from Nivre (2003) and the *arc-standard* variant introduced in Nivre (2004). They use exactly the same type of configurations and both are restricted to projective dependency trees, but while the arc-eager algorithm is incremental and adds every arc at the earliest

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

possible moment, the arc-standard algorithm uses a strict bottom-up strategy and therefore has to postpone the attachment of some right-dependents. The arc-eager algorithm, which is illustrated in Figures 1 and 2, is the default parsing algorithm in MaltParser.

(2) **Covington's algorithms** include the projective and non-projective versions of the algorithm adapted by Nivre (2007a, 2008) for MaltParser and previously described by Covington (2001). Covington's algorithms behave similarly to Nivre's arc-eager with respect to parsing order, but use a different type of configuration with open lists instead of a stack and a buffer.

(3) **Stack algorithms** include the projective and non-projective versions of the algorithm described by Nivre (2009) and Nivre, Kuhlmann and Hall (2009). These parsing algorithms are similar to Nivre's arc-standard parsing algorithm concerning parsing order, but they use a different type of configuration where arcs are always added between tokens on the stack. The non-projective versions use a special SWAP transition that reorders words on the stack to permit non-projective dependencies (Nivre 2009).

(4) **Multiplanar parsers** include the Planar and 2-Planar parsing algorithms described by Gómez-Rodríguez and Nivre (2010). These algorithms are based on the concept of *planarity* in graphs and its extension to *multiplanar* dependency structures (Yli-Jyrä 2003). The Planar parser can only parse dependency trees with no crossing arcs, which is a superset of projective dependency trees. The 2-Planar parser allows the construction of trees that can be divided into two different planar graphs, thereby allowing crossing arcs and non-projective structures.

Any algorithm that cannot handle non-projective structures, such as the ones included in Nivre's family, can be used to parse non-projective trees by combining it with pseudo-projective pre- and post-processing (Nivre and Nilsson 2005). At training time, all trees are projectivized and (partial) information about the transformation is encoded in augmented arc labels. At parsing time, the augmented arc labels are used to guide a heuristic search for the inverse transformation, resulting in non-projective output trees. Pseudo-projective parsing offers competitive results that are often on par with parsing algorithms that directly handle non-projective structures, such as Covington's non-projective algorithm.

3.2 Feature model

The transition-based parsing strategy allows the inclusion of features extracted from the entire parser configuration (as well as the preceding transition sequence). Crucially, this means that features can be defined not only over the input sentence but also over the partially built dependency structure, which can be expensive in parsers that apply dynamic programming. MaltParser incorporates a specification language for feature templates, which can be used to define models of arbitrary complexity. The most common feature types are based on extracting the values

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

of linguistic attributes defined in the CoNLL data format,[2] which is the default input/output format in MaltParser:

(1) FORM: Word form.
(2) LEMMA: Lemma or stemmed version of the word.
(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.

Basic feature templates are defined by combining such an attribute with a token address that is relative to the parser configuration. For example, in the arc-eager system defined in Figure 1, we can define a feature template for the POSTAG of the token on top of the stack and another feature template for the POSTAG of the next input token. In general, most feature templates are defined relative to what we will call the *focus tokens*, namely the two tokens between which it is possible to add a dependency arc in the next transition. In the arc-eager system, the *left* focus token is the token on top of the stack, and the *right* focus token is the next input token, but in other systems these are referenced differently (see Section 4.3.1). Finally, the specification language provides a number of operations on basic feature templates, such as the splitting of list-valued features into atomic (boolean) features (cf. FEATS above) and conjunction of features.

When running MaltParser with default settings, a default feature model is invoked with four groups of feature templates.[3]

(1) A wide window of POSTAG features containing the focus tokens plus one token to the left and three tokens to the right.
(2) A narrower window of FORM features containing the focus tokens plus one token to the right.
(3) A small set of DEPREL features over dependents and heads of the focus tokens.
(4) A small set of feature conjunctions, in particular POSTAG n-grams, covering the same window as the basic POSTAG features and pairs of POSTAG and FORM features over the focus tokens.

As we shall see later on, the optimization of feature models is based on both tuning the size of these feature groups and exploring additional features that may be available in the input sentence such as CPOSTAG, LEMMA, and FEATS features.

3.3 Learning algorithm

MaltParser makes use of two libraries for learning to predict transitions: LIBSVM (Chang and Lin 2011) and LIBLINEAR (Fan *et al.* 2008). The LIBSVM package

[2] See `http://nextens.uvt.nl/depparse-wiki/DataFormat`.
[3] For more information about default feature models, see the MaltParser documentation at `http://maltparser.org`.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

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 hyper-parameters, which can be optimized.

In general, it is possible to achieve roughly the same level of accuracy with both libraries, and in some cases LIBLINEAR has even been shown to outperform LIBSVM (Prudhvi Kosaraju and Kukkadapu 2010; Gómez-Rodríguez and Fernández-González 2012). Moreover, LIBLINEAR is always faster, both during training and parsing, and it is the default library of MaltParser since version 1.7.2. Given that speed is important when running a large number of tuning experiments, MaltOptimizer is currently restricted to the use of LIBLINEAR for machine learning, and more specifically to the LIBLINEAR implementation of the multi-class SVM of Crammer and Singer (2006). It is possible that future versions of MaltOptimizer will support additional learning algorithms.

This means that machine learning optimization in MaltOptimizer is restricted to the cost parameter C, which is the single hyper-parameter of the linear support vector machine (Fan *et al.* 2008). This parameter enforces the margin between the different classes and can be tuned to allow some instances to be misclassified. A high C value might lead to overfitting, providing a model with lower training error but higher test error due to a narrower margin. For MaltParser optimization, Cassel (2009) recommends only considering C values between 0 and 1.

4 MaltOptimizer

The main idea behind MaltOptimizer is to provide a tool that automatically optimizes the parsing algorithms, feature models, and parameters described in Section 3. The tool is primarily aimed at non-experts who need to train a parser without deep knowledge of transition-based dependency parsing in general and MaltParser in particular. For more expert users, it can also be a way of saving time and work, since it replaces the specialized scripts normally used to automate parser optimization.

MaltOptimizer implements an optimization procedure for MaltParser models based on acquired previous experience (Hall *et al.* 2007; Nivre and Hall 2010). The input to the system is a training set containing sentences annotated with dependency trees in the CoNLL data format, and the optimization is divided into three different phases, with optional input from the user after each phase:

(1) Data analysis and initial optimization
(2) Parsing algorithm selection
(3) Feature selection and LIBLINEAR optimization

The general approach to optimization is to train MaltParser models with different hyper-parameter settings and evaluate them on held out data, either using a single development test set or cross-validation in the case of small data sets. This gives

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

us a statistical estimate of the true generalization error, which can be used to rank competing models and select the approximately best one. We vary one hyper-parameter at a time and always keep the best model so far in a greedy fashion. As discussed in Section 2, this method does not guarantee that we find an optimal model, but it is a well-established practice in the field, and experimental results indicate that it works reasonably well in most cases.

The evaluation metric used to evaluate models is by default the LAS, that is, the percentage of words that are assigned the correct syntactic head and the correct dependency label computed over all tokens, including punctuation. However, MaltOptimizer can be configured to exclude punctuation and/or use different evaluation metrics such as unlabeled attachment score (UAS) or labeled/unlabeled complete match (L/UCM). In order to select a new model, MaltOptimizer generally requires an increase in LAS by at least 0.05 points, except in the case of backward feature selection, where the threshold is relaxed to 0.00 in order to favor simpler models over more complex ones when this does not hurt accuracy.

Before we describe each of the three optimization phases in turn in the following sections, it is worth noting again that MaltOptimizer restricts the machine learning component of MaltParser to the multi-class SVM of Crammer and Singer (2006) as implemented in LIBLINEAR (Fan *et al.* 2008). This decision was made in the interest of efficiency and to allow the exploration of a wider range of features.

4.1 Phase 1: data analysis and initial optimization

In phase 1, MaltOptimizer performs an analysis of the training set used as input in order to set up the rest of the optimization process. It also suggests the best validation strategy according to the size of the data set and optimizes some initial parameters.

4.1.1 Data analysis

MaltOptimizer first performs a validation of the training set used as input to ensure that it is correctly formatted and that is possible to perform all the optimization steps. To this end, it runs the official validation script from the CoNLL-X-shared task (*validateFormat.py*)[4], which tests whether the data set contains the ten standard columns of the CoNLL-X format, that there are no cycles in the sentences which would violate the dependency tree constraint and some minor (but important) checks of the character encoding. The script also gives warnings when something is inconsistent but not fatal for training a parser. MaltOptimizer stops the process if the data set does not pass the validation script and informs the user about how to fix it by showing the output of the script. MaltOptimizer also informs the user if there are warnings that should be checked, even if it is still possible to run the rest of the process.

[4] The script can be downloaded from `http://ilk.uvt.nl/conll/software.html`.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

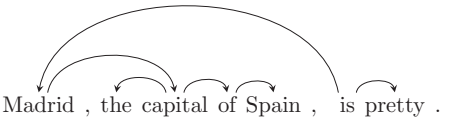


Fig. 3. An English sentence with two covered roots (a case of 'dangling punctuation').

Before analyzing the data set, MaltOptimizer calculates the amount of dynamic memory needed to run the whole optimization process. In order to do so, it trains a model with Covington's non-projective parsing algorithm (which requires most memory) while varying the Java heap size until it finds an optimal setting. In case the computer used to run the optimization does not have a sufficient amount of memory, MaltOptimizer iteratively reduces the size of the data set (by removing sentences) until it can be run and informs the user about it.

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

(1) *Training set size*: Number of words/sentences.
(2) *Non-projectivity*: Percentage of non-projective trees.
(3) *Covered roots*: Number of root nodes covered by a dependency arc.
(4) *Multiple root label*s: Number of dependency labels used for root nodes.
(5) Identity of CPOSTAG and POSTAG values (boolean).
(6) Empty LEMMA and FEATS columns (boolean).

The notion of covered roots (property 3) is illustrated in Figure 3, which shows a dependency tree with two covered roots: (i) The first comma is covered by the arc between *Madrid* and *capital*. (ii) The second comma is covered by the arc between *is* and *Madrid*. Cases like these, where punctuation tokens are not attached to words and therefore may end up being covered roots, are sometimes referred to as 'dangling punctuation'. The issue of multiple root labels (property 4) is relevant because, although most data sets use a dummy label like ROOT for all root nodes, there are data sets that use different labels to make functional distinctions, such as the data sets for Czech, Portuguese, Arabic, or Slovene treebanks from the CoNLL 2006 and 2007-shared tasks. For part-of-speech tags (property 5), The CoNLL data format provides for both coarse-grained (CPOSTAG) and fine-grained (POSTAG) tags. Since both fields are required, some data sets have the same tags in both fields. Finally, regarding lemmas and morphosyntactic features (property 6), these may be missing in some data sets, in which case one or both of these fields may be completely filled with null values in the form of an underscore (_).

The information from the data analysis is used in all the following optimization steps. More specifically, property 1 is used to suggest the best validation strategy (simple train-devtest split or five-fold cross-validation); properties 2, 3, and 4 are relevant for the choice of parsing algorithm in phase 2 as well as for some parameters of pre- and post-processing in phase 1 (see below); properties 5 and 6 are important for the feature selection experiments in phase 3.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

4.1.2 Validation strategy

Based on the size of the data set, MaltOptimizer recommends the user to choose from two validation methods that are available during phases 2 and 3:

(1) Simple training-devtest split (80% training, 20% devtest). This validation method is recommended for large data sets with more than 90,000 tokens. In these cases, cross-validation would be time consuming and a single devtest set is large enough to give reliable estimates.
(2) Five-fold cross validation (20% in each fold). This method is recommended for smaller data sets, where a single evaluation estimate might be unreliable and the extra time needed to run cross-validation is likely to be tolerable.

In both cases, the system performs stratified sampling to ensure a homogeneous distribution of data in all subsets. MaltOptimizer by default uses the training-devtest split in phases 2 and 3, but the user can override this regardless of the data set size.

4.1.3 Initial optimization

At the beginning of the optimization process, MaltOptimizer makes some initial tests with the default parsing algorithm, which, as stated above, is Nivre's arc-eager algorithm with default settings. These tests would be the same for any parsing algorithm, and this is why they are tested on the default algorithm in phase 1. The initial tests are the following:

- If there are covered roots, MaltOptimizer tests MaltParser's built-in techniques for attaching these to a nearby word (with a specific label) before training and unattaching them after parsing (to prevent having to deal with spurious non-projective dependencies). This is done using the flag `-pcr` with one of the values *none* (default), *left*, *right*, and *head* for attaching the covered root to no node, the left node, the right node, or the head node of the shortest covering arc. The option with the best accuracy is retained in the following optimization phases.
- If there are multiple root labels, MaltOptimizer tests which one should be used as the default root label, which is used for any token that is left unattached at the end of parsing. This test is performed by modifying the `-grl` option provided in MaltParser, and the best value is kept for further optimization.

At the end of phase 1, MaltOptimizer creates an option file and a log file which is used as the starting point for optimization. The user is given the opportunity to edit the option file before the optimization continues by modifying the default root label or the optimal way of handling covered roots. The user may also choose to stop the process and continue with manual optimization. Figure 4 shows the output log file (top) and the output option file after phase 1 for the Slovene training set from the CoNLL 2006-shared task. In addition to the results of data analysis, the log file shows the best LAS achieved so far as well as the corresponding score with default settings. This information is updated in later phases, whereas the rest of the log file remains the same.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

```
phase1_logfile.txt
------------------------------
Training set path: slovene_sdt_train.conll
Size (tokens): 28750
Size (sentences): 1534
Non-projective: 22.16
Covered roots: 27
LAS: 66.15
Default: 65.63
NumRootLabels: 11
JavaHeap: -Xmx2048m
CposEqPos:false
LemmaBlank: false
FeatsBlank: false

phase1_optfile.txt
------------------------------
1. root_label (-grl): Pred
2. covered_root (-pcr): right
```

Fig. 4. Phase 1 log file and option file. The log file (`phase1_logfile.txt`) shows the main characteristics of the data set plus the results of the default model and the best model so far. The option file (`phase1_optfile.txt`) shows the options that are different from default settings. The user may edit the option file before continuing.

4.2 Phase 2: parsing algorithm selection

In the second phase, MaltOptimizer explores the parsing algorithms implemented in MaltParser with default feature models and parameters (see Section 3) based on the results of the data analysis. Once the algorithm is selected, MaltOptimizer also tunes the specific options of each parsing algorithm (if any).

4.2.1 Parsing algorithm selection

Taking into account the percentage of non-projective trees in the training set, MaltOptimizer explores the parsing algorithms implemented in MaltParser as follows:

- If there are no non-projective arcs/trees in the training set, only projective algorithms are explored:
  - — The arc-eager and arc-standard versions of Nivre's algorithm.
  - — Covington's projective parsing algorithm.
  - — The projective Stack algorithm.
- If more than 15% of the trees in the training set are non-projective, MaltOptimizer instead tests the following algorithms:
  - — Covington's non-projective algorithm.
  - — The non-projective Stack algorithms.
  - — Projective algorithms in combination with pseudo-projective parsing.
- Otherwise, MaltOptimizer examines both groups of algorithms and finally suggests the one that provides the best results of all.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

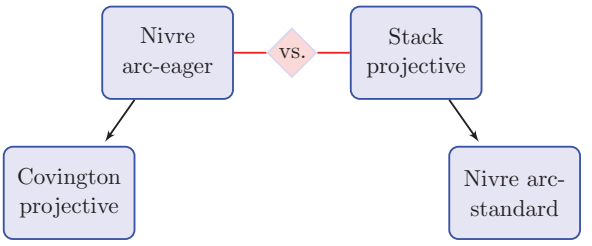


Fig. 5. (Colour online) Decision tree for the best projective algorithm. Each box in the decision tree represents the test of an algorithm. An undirected link labeled 'vs.' signifies that both algorithms are evaluated against each other. A directed link means that the lower algorithm is evaluated only if the higher one is the best so far.

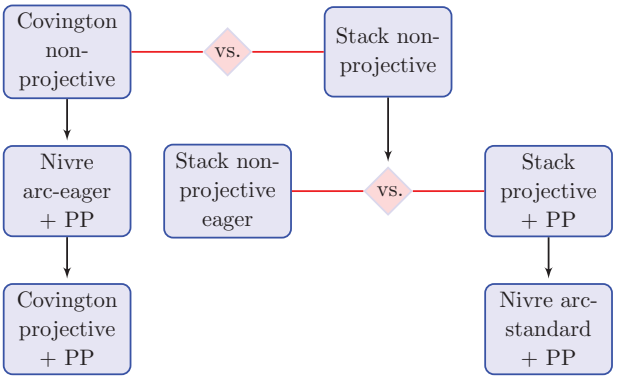


Fig. 6. (Colour online) Decision tree for the best non-projective algorithm (+PP = combined with pseudoprojective parsing). Each box in the decision tree represents the test of an algorithm. An undirected link labeled 'vs.' signifies that both algorithms are evaluated against each other. A directed link means that the lower algorithm (or algorithm pair) is evaluated only if the higher one is the best so far.

Note that we also performed optimization experiments with MaltOptimizer and the planar parsers for feature selection (Ballesteros, Gómez-Rodríguez and Nivre 2012). However the planar parsers are not included in the standard version of MaltOptimizer because they were never selected as best parsers. In general, MaltOptimizer can be easily extended by including new parsing algorithms.

In order to reduce the number of tests needed, we use two different decision trees based on previous experience (Nivre and Hall 2010). The first one, shown in Figure 5, tests projective algorithms in such a way that the maximum number of tests is three, and the procedure avoids unnecessary tests such as testing Nivre's arc-standard algorithm when Nivre's arc-eager algorithm outperforms the projective Stack algorithm because the arc-standard algorithm uses the same parsing order as the Stack algorithm. The second decision tree, shown in Figure 6, is for non-projective algorithms and results in a maximum of five tests using similar considerations. However, in this case we have a larger number of algorithms to consider.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

4.2.2 Parsing algorithm parameter optimization

Once MaltOptimizer decides which parsing algorithm is the best with default settings after traversing the decision trees, it tunes the parameters of the best performing algorithm and creates a new option file for the best configuration according to the evaluation measure selected. The tests that MaltOptimizer performs for each parsing algorithm are the following:

- **Nivre's algorithms**: MaltOptimizer tests the following two boolean options:
  - `allow_root`: When set to true (default), the parser starts parsing with an artificial root node on the stack, and root tokens are explicitly attached to the artificial root in a RIGHT-ARC transition, which allows different (non-default) labels to be chosen for root tokens. When set to false, the parser starts with an empty stack, as illustrated in Figure 2.
  - `allow_reduce`: When set to false (default), the parser strictly enforces the REDUCE precondition (that the token popped must be attached to a head). When set to true, REDUCE is always permitted.
- **Covington's algorithms**: MaltOptimizer tests the following two boolean options:
  - `allow_root`: Same as for Nivre's algorithms (see above).
  - `allow_shift`: When set to false (default), no SHIFT is allowed, which means that the parser must consider a link between every new word and every preceding word. When set to true, the parser can choose SHIFT at any point and thereby disregard a subset of the preceding words.
- **Pseudo-projective options**: If MaltOptimizer finds that the best parsing algorithm is a projective algorithm that should be run with pseudo-projective parsing (+PP in Figure 6), it tests the various values of the `-pp` option defining the encodings used to record the projectivization transform in augmented arc labels (Nivre and Nilsson 2005):
  - `baseline`: Only projectivize training data, no post-processing.
  - `head`: Projectivize with head encoding.
  - `path`: Projectivize with path encoding.
  - `head+path`: Projectivize with head+path encoding.

At the end of phase 2, the user is again given the opportunity to edit the option file (or stop the process) before optimization continues. For instance, the user may change the parsing algorithm or some option selected by MaltOptimizer as optimal. Figure 7 shows the log file and the option file after phase 2 for the Slovene treebank of the CoNLL 2006-shared task.

4.3 Phase 3: feature selection and LIBLINEAR optimization

In the third phase, MaltOptimizer tries to optimize the feature model for the parsing algorithm selected as best and given the parameters chosen so far. Finally, it tunes the single hyper-parameter of the LIBLINEAR classifier.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

```
phase2_logfile.txt
------------------------------
Training set path: slovene_sdt_train.conll
Size (tokens): 28750
Size (sentences): 1534
Non-projective: 22.16
Covered roots: 27
LAS: 68.59
Default: 65.63
NumRootLabels: 11
JavaHeap: -Xmx2048m
CposEqPos: false
LemmaBlank: false
FeatsBlank: false

phase2_optfile.txt
------------------------------
1. root_label (-grl): Pred
2. covered_root (-pcr): right
3. parsing_algorithm (-a): stackproj
4. marking_strategy (-pp): head+path
```

Fig. 7. Phase 2 log file and option file. The log file (`phase2_logfile.txt`) shows the main characteristics of the data set plus the results of the default model and the best model so far. The option file (`phase2_optfile.txt`) shows the options that are different from default settings. (The algorithm `stackproj` is the projective Stack algorithm and the -pp option refers to the pseudo-projective encoding.) The user may edit the option file before continuing.

Let $F = \{F_1, \ldots, F_n\}$ be the given set of features and let $M(X)$ be the evaluation metric for feature set X.

```
1   X ← F
2   while |X| > 0
3       B ← 0
4       Y ← ∅
5       for each X_i ∈ X
6           if M(X \ {X_i}) > B then
7               B ← M(X \ {X_i})
8               Y ← X \ {X_i}
9       if M(X) > B then
10          return X
11      else
12          X ← Y
13  return X
```

Fig. 8. Algorithm for backward feature selection.

4.3.1 Feature selection

The two basic techniques for (greedy) feature selection are *backward* and *forward* selections. Backward selection starts from a maximal set of features and systematically omits features until no further improvement is observed, as shown in Figure 8.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Let $F = \{F_1, \ldots, F_n\}$ be the pool of potential features
and let $M(X)$ be the evaluation metric for feature set X.

```
1   X ← ∅
2   while X ≠ F
3       B ← 0
4       Y ← ∅
5       for each X_i ∈ F \ X
6           if M(X ∪ {X_i}) > B then
7               B ← M(X ∪ {X_i})
8               Y ← X ∪ {X_i}
9       if M(X) > B then
10          return X
11      else
12          X ← Y
13  return X
```

Fig. 9. Algorithm for forward feature selection.

Forward selection instead starts from the empty feature set and systematically adds features as long as this improves performance, as outlined in Figure 9. Both backward and forward algorithms can be quite expensive even for moderately sized feature sets. Moreover, in the case of MaltParser, the pool of potential features is in principle infinite, making unconstrained backward and forward selection practically infeasible. Our strategy for dealing with this problem is to start from the default feature model for the parsing algorithm selected in phase 2 and try to optimize the different feature groups using a combination of backward and forward selection that exploits the intrinsic structure of the model, such as the constraint that POSTAG features should always form a contiguous window that includes the two focus tokens. This strategy is based on proven experience from previous studies of feature optimization (Hall *et al.* 2007; Nivre and Hall 2010). The full feature selection procedure consists of the following six steps:

(1) Backward selection of POSTAG (n-gram) features, proceeding from the periphery toward the focus tokens, followed by forward selection in the opposite direction only if backward selection did not result in omission of any feature.
(2) Backward selection of FORM features, proceeding from the periphery towards the focus tokens, followed by forward selection in the opposite direction only if backward selection did not result in omission of any feature.
(3) Backward selection of DEPREL features, followed by forward selection of POSTAG features for the same dependency tree nodes.
(4) Forward selection of string context features (predecessors and successors of focus tokens) using the POSTAG and FORM attributes.
(5) Forward selection of CPOSTAG, FEATS, and LEMMA features (if available), starting with the focus tokens and extending the window leftwards and rightwards.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Table 1. *Token references used in feature specifications for different families of parsing algorithms. L/RFocus = left/right focus token; L/RContext = left/right context for left/right focus token; CContext = central context between focus tokens (available only for non-projective algorithms)*

Algorithm	LContext	LFocus	CContext	RFocus	RContext
Nivre	Stack[i] ($i > 0$)	Stack[0]	–	Input[0]	Input[i] ($i > 0$)
Covington	Left[i] ($i > 0$)	Left[0]	LeftContext[i] RightContext[i] ($i \geq 0$)	Right[0]	Right[i] ($i > 0$)
Stack	Stack[i] ($i > 1$)	Stack[1]	Input[i] ($i \geq 0$)	Stack[0]	Lookahead[i] ($i \geq 0$)

(6) Forward selection of conjunctions of POSTAG and FORM features, starting with the focus tokens and extending the window leftwards and rightwards.

Whenever a feature group forms a contiguous window, including the focus tokens, backward/forward selection is interrupted as soon as the omission/addition of a feature leads to a drop in accuracy. This linguistically motivated heuristic speeds up the process considerably and has been shown to give results on par with more exhaustive search (Ballesteros 2013b).

The exact implementation of the six steps differs depending on which parsing algorithm has been selected, because the algorithms use different configuration types and therefore require different token references. Table 1 shows how to refer to the left and right focus tokens and their surrounding context for different families of parsing algorithms. Figure 10 shows the feature specification file resulting from the feature selection experiments for the Slovene data set from the CoNLL 2006-shared task (where the non-projective Stack algorithm was selected in phase 2).

4.3.2 LIBLINEAR *optimization*

At the end of phase 3, MaltOptimizer tunes the C parameter of the multi-class SVM in LIBLINEAR using a simple grid search. It starts with $C = 0.01$ and proceeds by testing values of 0.1, 0.2, 0.3, *etc.* up to 1.0, selecting the best C value that it can find.

After all the feature selection and machine learning experiments are completed, MaltOptimizer creates a new log file, a new option file, and a new feature specification file. After that the user can continue with manual optimization using the MaltOptimizer outcome as a starting point or modifying the option file if appropriate. Figure 11 shows the log and option files after phase 3 for the Slovene training set from the CoNLL 2006-shared task. The option file references the feature specification file shown in Figure 10.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

```
<featuremodel name="stackprojective">
   <feature>InputColumn(POSTAG, Stack[0])</feature>
   <feature>InputColumn(POSTAG, Stack[1])</feature>
   <feature>InputColumn(POSTAG, Stack[2])</feature>
   <feature>InputColumn(POSTAG, Lookahead[0])</feature>
   <feature>InputColumn(POSTAG, Lookahead[1])</feature>
   <feature>InputColumn(POSTAG, Lookahead[2])</feature>
   <feature>Merge(InputColumn(POSTAG, Stack[1]),
                  InputColumn(POSTAG, Stack[0]))
   </feature>
   <feature>Merge3(InputColumn(POSTAG, Stack[2]),
                   InputColumn(POSTAG, Stack[1]),
                   InputColumn(POSTAG, Stack[0]))
   </feature>
   <feature>Merge3(InputColumn(POSTAG, Stack[1]),
                   InputColumn(POSTAG, Stack[0]),
                   InputColumn(POSTAG, Lookahead[0]))
   </feature>
   <feature>Merge3(InputColumn(POSTAG, Stack[0]),
                   InputColumn(POSTAG, Lookahead[0]),
                   InputColumn(POSTAG, Lookahead[1]))
   </feature>
   <feature>Merge3(InputColumn(POSTAG, Lookahead[0]),
                   InputColumn(POSTAG, Lookahead[1]),
                   InputColumn(POSTAG, Lookahead[2]))
   </feature>
   <feature>OutputColumn(DEPREL, ldep(Stack[0]))</feature>
   <feature>OutputColumn(DEPREL, rdep(Stack[0]))</feature>
   <feature>OutputColumn(DEPREL, ldep(Stack[1]))</feature>
   <feature>OutputColumn(DEPREL, rdep(Stack[1]))</feature>
   <feature>Merge3(InputColumn(POSTAG, Stack[0]),
                   OutputColumn(DEPREL, ldep(Stack[0])),
                   OutputColumn(DEPREL, rdep(Stack[0])))
   </feature>
   <feature>Merge3(InputColumn(POSTAG, Stack[1]),
                   OutputColumn(DEPREL, ldep(Stack[1])),
                   OutputColumn(DEPREL, rdep(Stack[1])))
   </feature>
   <feature>InputColumn(FORM, Stack[0])</feature>
   <feature>Merge(InputColumn(POSTAG, Stack[1]),
                  InputColumn(FORM, Stack[1]))
   </feature>
   <feature>InputColumn(FORM, Stack[1])</feature>
   <feature>Merge(InputColumn(POSTAG, Stack[1]),
                  InputColumn(FORM, Stack[2]))
   </feature>
   <feature>InputColumn(FORM, Lookahead[0])</feature>
   <feature>Merge(InputColumn(POSTAG, Stack[1]),
                  InputColumn(FORM, Lookahead[0]))
   </feature>
   <feature>Split(InputColumn(FEATS, Stack[0]),\|)</feature>
   <feature>Split(InputColumn(FEATS, Lookahead[0]),\|)</feature>
   <feature>Split(InputColumn(FEATS, Lookahead[1]),\|)</feature>
   <feature>Split(InputColumn(FEATS, Stack[1]),\|)</feature>
</featuremodel>
```

Fig. 10. Feature specification for Slovene (the projective Stack algorithm with pseudo-projective parsing) after the optimization process.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

```
phase3_logfile.txt
------------------------------
Training set path: slovene_sdt_train.conll
Size (tokens): 28750
Size (sentences): 1534
Non-projective: 22.16
Covered roots: 27
LAS: 71.14
Default: 65.63
NumRootLabels: 11
JavaHeap: -Xmx2048m
CposEqPos: false
LemmaBlank: false
FeatsBlank: false

phase3_optfile.txt
------------------------------
1. root_label (-grl): Pred
2. covered_root (-pcr): right
3. parsing_algorithm (-a): stackproj
4. marking_strategy (-pp): head+path
8. feature_model (-F): addMergPOSTAGS0FORMStack1.xml
```

Fig. 11. Phase 3 log file and option file. The log file (`phase3_logfile.txt`) shows the main characteristics of the data set plus the results of the default model and the best model so far. The option file (`phase3_optfile.txt`) shows the options that are different from default settings. (The numbering jumps from 4 to 8 because slots 5–7 are reserved for parsing algorithm options that are not used for the selected algorithm.) The user may edit the option file and continue with manual optimization.

5 Experiments with MaltOptimizer

In this section, we present a set of experiments performed on all the data sets from the CoNLL 2006 and 2007-shared tasks on dependency parsing (Buchholz and Marsi 2006; Nivre *et al.* 2007a). We run all the optimization phases on the training sets, using a simple training-devtest split for large training sets and five-fold cross-validation for all training sets smaller than 90,000 tokens, in line with the suggestions provided by MaltOptimizer (see Section 4.1.2).[5]

Table 2 shows the LAS, first with default settings and then after each of the three optimization phases, as well as the difference between the final configuration and the default. The last column shows the parsing algorithm selected by MaltOptimizer for each of the data sets. Table 3 compares the accuracy achieved on the final test set (Test-MO) with MaltParser in default settings (Default), and with the best score obtained by careful manual optimization of MaltParser (Test-MP) during the

[5] The results are obtained with the most recent versions of MaltOptimizer (1.0.3) and MaltParser (1.7.2) and do not coincide exactly with the ones published in Ballesteros and Nivre (2012a). Both MaltParser and MaltOptimizer have evolved since the first version of MaltOptimizer, which means that there are minor variations in experimental results although the overall patterns are the same. In case someone wants to reproduce the old results, all previous versions of MaltParser and MaltOptimizer are available on the websites http://nil.fdi.ucm.es/maltoptimizer/index.html and http://maltparser.org.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Table 2. *Labeled attachment score (LAS) per phase compared with default settings for all training sets from the CoNLL 2006 and 2007-shared tasks. Languages marked with * have a training set smaller than 90,000 tokens and have been optimized using five-fold cross-validation; the remaining languages have been optimized using a simple training-devtest split. The final column shows the parsing algorithm selected in phase 2 (P = projective, NP = non-projective, PP = pseudo-projective, E = arc-eager, S = arc-standard)*

CoNLL 2006-shared task						
Language	Def	1	2	3	Diff	Algorithm
Arabic*	62.99	63.02	63.85	66.11	3.12	Stack NP
Bulgarian	83.21	83.21	84.00	86.33	3.12	Stack NP
Chinese	84.14	84.14	85.11	86.27	2.13	Stack P
Czech	68.85	69.26	72.39	78.54	9.69	Stack NP
Danish	80.76	80.86	81.75	84.92	4.16	Stack P+PP
Dutch	72.69	72.69	78.06	82.48	9.79	Nivre E+PP
German	81.80	81.80	83.56	86.20	4.40	Covington NP
Japanese	89.76	89.76	90.92	90.92	1.16	Covington P
Portuguese	83.21	83.67	84.88	86.80	3.56	Stack P+PP
Slovene*	65.63	66.15	68.59	71.14	5.51	Stack P+PP
Spanish*	76.59	76.59	76.64	78.60	2.01	Nivre E
Swedish	83.20	83.20	83.50	84.20	1.00	Nivre E+PP
Turkish*	57.39	57.39	58.06	67.15	9.76	Covington NP
CoNLL 2007-shared task						
Language	Def	1	2	3	Diff	Algorithm
Arabic	66.95	67.55	67.73	71.11	4.16	Nivre S+PP
Basque*	66.96	67.80	68.47	75.52	8.56	Covington NP
Catalan	82.86	82.86	83.13	84.90	2.04	Nivre E
Chinese	84.04	84.04	85.03	86.21	2.17	Stack P
Czech	67.39	69.41	72.48	77.85	10.45	Stack P+PP
English	83.85	83.85	85.34	86.61	2.76	Nivre S+PP
Greek*	69.68	70.79	72.34	75.27	5.59	Nivre S + PP
Hungarian	65.99	65.99	68.21	76.53	10.54	Stack P+PP
Italian*	79.38	79.38	79.58	81.17	1.79	Nivre E+PP
Turkish*	67.33	68.44	69.81	75.78	8.45	Covington NP
Average	74.98	75.30	76.67	80.03	5.04	

original shared tasks (Nivre *et al.* 2006a; Hall *et al.* 2007). In addition, since all the Test-MP results were obtained with Nivre's arc-eager parsing algorithm, we also present the results of MaltOptimizer after fixing the parsing algorithm to Nivre arc-eager (Test-MO[N]). The results exclude punctuation symbols for CoNLL 2006 corpora and include punctuation symbols for CoNLL 2007 corpora in line with the scoring policies of the original shared tasks.

The first conclusion to extract is that MaltOptimizer improves parsing accuracy for all languages without exception not only on the development test sets (or with cross-validation), as shown in Table 2, but more importantly on the final test sets,

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Table 3. *Labeled attachment score (LAS) on final test sets for MaltParser with default settings (default), the best model found by MaltOptimizer with Nivre's arc-eager parsing algorithm (Test-MO[N]), the best model found by MaltOptimizer overall (Test-MO), and the best MaltParser model in the CoNLL 2006 and 2007-shared tasks (Test-MP). Languages marked with * have a training set smaller than 90,000 tokens and have been optimized using five-fold cross-validation*

CoNLL 2006-shared task

Language	Default	Test-MO[N]	Test-MO	Test-MP
Arabic*	63.91	65.65	66.43	**66.71**
Bulgarian	84.30	86.18	86.18	**87.41**
Chinese	84.55	85.67	85.81	**86.92**
Czech	71.80	77.28	**81.14**	78.42
Danish	80.92	83.61	84.43	**84.77**
Dutch	68.89	76.91	76.91	**78.59**
German	83.53	84.98	85.40	**85.82**
Japanese	89.69	89.69	90.39	**91.65**
Portuguese	83.05	84.99	86.18	**87.60**
Slovene*	67.63	70.24	**73.16**	70.30
Spanish*	78.26	79.26	79.44	**81.29**
Swedish	83.47	83.85	83.85	**84.58**
Turkish*	56.28	64.39	65.03	**65.68**

CoNLL 2007-shared task

Language	Default	Test-MO[N]	Test-MO	Test-MP
Arabic	68.89	73.09	74.45	**74.75**
Basque*	63.38	72.93	72.97	**74.99**
Catalan	83.61	85.29	85.59	**87.74**
Chinese	78.86	80.66	82.58	**83.51**
Czech	66.72	72.93	**77.48**	77.22
English	83.69	85.07	85.17	**85.81**
Greek*	70.79	72.42	**74.31**	74.21
Hungarian	66.12	75.15	77.17	**78.09**
Italian*	79.98	81.93	81.93	**82.48**
Turkish*	67.52	77.95	78.57	**79.24**
Average	75.04	78.70	79.76	**80.34**

as shown in Table 3. This indicates that the optimization strategy implemented in the tool is a reliable way of tuning MaltParser for new data sets even in the absence of theoretical optimization guarantees. However, the amount of improvement varies considerably from about 1–2 percentage points for languages such as Catalan, Chinese, Japanese, or Swedish to 7–10 percentage points for Basque (CoNLL 2007), Czech (CoNLL 2006 and 2007), Dutch (CoNLL 2006), Hungarian (CoNLL 2007), and Turkish (CoNLL 2006 and 2007). Other languages, such as Greek (CoNLL 2007) or Slovene (CoNLL 2006), obtain intermediate improvements.

One important factor for explaining the different levels of improvement is the amount of linguistic annotation available for different languages. For most

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

languages, the greatest improvement comes from feature selection in phase 3, and this is true in particular for languages with rich morphosyntactic annotation, where the addition of morphological features often improve accuracy substantially. However, there are also quite remarkable improvements from phase 2, in this case for languages with a substantial amount of non-projective dependencies, such as Czech, Dutch, and Slovene, where the selection of parsing algorithm gives a high impact on the results. In Table 2, we see that the languages for which a non-projective algorithm is selected are the ones that have the greatest improvement during phase 2. A good example is Czech (CoNLL 2006 and 2007). It is also worth noting that the LIBLINEAR optimization at the end of phase 3 never resulted in any improvements over the MaltParser default value of 0.1 for the cost parameter C. Nevertheless, we think it is important to include this test for completeness, since it could give different results for new data sets.

Looking more closely at the final test results in Table 3, we may conclude that MaltOptimizer models perform competitively compared with the manually optimized MaltParser models. We see an average difference of only 0.58, a maximum negative difference of 2.15 (for Catalan CoNLL 2007), and a maximum positive difference of 2.86 (for Slovene CoNLL 2006). We even observe that, in four cases out of twenty-three, MaltOptimizer actually improves on the manually optimized models and in a few cases with a quite substantial margin, for instance Czech (CoNLL 2006) and Slovene (CoNLL 2006). In fact, in the 2006-shared task, MaltOptimizer would have finished third, outperformed only by MSTParser (McDonald, Lerman and Pereira 2006) and the manually optimized version of MaltParser (Nivre *et al.* 2006a). This is arguably a strong result for a system that performs a completely automatic optimization. Even more so if we compare the time and knowledge needed to get results in the same accuracy range with manual optimization.

Considering finally the results of MaltOptimizer with a fixed parsing algorithm (Nivre arc-eager), we see that MaltOptimizer approaches the results based on manual optimization and substantially outperforms the results based on default settings. However, it is also clear that the freedom to choose different parsing algorithms is a powerful feature of the more recent versions of MaltParser, which is highlighted by the difference in results between Test-MO[N] and Test-MO for languages such as Czech (CoNLL 2006 and 2007) or Slovene (CoNLL 2006).

The time needed to run the optimization depends primarily on the size of the training set and the validation method chosen. With a simple training-devtest split, it ranges from about half an hour for the smallest data sets to about 30 hours for the largest ones. Running five-fold cross-validation simply takes five times longer, which is why we recommend it primarily for smaller data sets. It is worth noting that the time needed depends crucially on the computer used to run the optimization and the parsing algorithm selected in phase 2. The estimates given in this paper were obtained with an Intel Xeon server with several cores, 2.8 GHz, and a quite substantial amount of dynamic memory. The memory requirements vary from 16 GB for the biggest data set (Czech CoNLL 2006) to 2 GB for the smallest (Slovene CoNLL 2006).

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

6 Related works

Several researchers have tried to find better feature models for transition-based parsers manually, such as Nivre *et al.* (2006a), Hall *et al.* (2007), Hall (2008), Zhang and Nivre (2011), and Agirre *et al.* (2011). However, besides MaltOptimizer, automatic feature selection has only been explored in a study by Nilsson and Nugues (2010). They start from a much more reduced feature model than the MaltParser default models, and incrementally add features using the 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, Zavrel and Daelemans (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. Moreover, a tool developed specifically for the optimization of learning algorithm parameters in the context of natural language processing is Paramsearch (van den Bosch 2004).

Feature selection is important in machine learning and has been the object of several studies (Guyon and Elisseeff 2003; McCallum 2003). Greedy methods like the ones employed by MaltOptimizer are well represented in the literature. For instance, Korycinski *et al.* (2003) apply an adaptive greedy feature selection technique to tackle hyperspectral data analysis in optical engineering. Doraisamy *et al.* (2008) present a comparative study on feature selection techniques applied to automatic genre classification, including best first, greedy stepwise, and genetic adaptive search, and conclude that all these methods can significantly improve the performance of general machine learning models. Finally, Das and Kempe (2011) demonstrate that greedy algorithms perform well even when features are highly correlated, which is definitely the case in transition-based dependency parsing.

At the time of writing, several groups have tried MaltOptimizer in their own research. For instance, Tsarfaty *et al.* (2012) use MaltOptimizer to optimize Hebrew parsing models in the context of introducing new and improved evaluation metrics. Seraji *et al.* (2012) apply MaltOptimizer to improve parsing accuracy when training MaltParser on a Persian treebank. Other researchers have used MaltOptimizer to optimize parsing models for Ancient Greek in order to find out which features are actually useful (Mambrini and Passarotti 2012). In a similar way, Agirre, Atutxa and Sarasola (2012) and Ballesteros, Mille and Burga (2013) have used MaltOptimizer to explore morphological feature sets in the case of Basque and Spanish respectively.

Finally, it is worth mentioning that MaltOptimizer participated in the Shared Task of Parsing Morphologically Rich Languages (Seddah *et al.* 2013), providing interesting and competitive results in comparison with other systems (Ballesteros 2013a). Moreover, MaltOptimizer was also used by other systems participating in the shared task (Cirik and Sensoy 2013; Goenaga, Gojenola and Ezeiza 2013).

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

7 Conclusions and future work

MaltOptimizer is an optimization tool for MaltParser that can support developers in adapting the system to new languages, new domains, or new annotation schemes. In Section 6 we have already cited work where MaltOptimizer has been used successfully to optimize MaltParser on different data sets. This shows that it is possible to get substantial improvements over the default settings, allowing non-experts in dependency parsing to achieve high (if not optimal) accuracy, which is usually not possible when using MaltParser in its default settings.

In addition to application developers, MaltOptimizer should also be useful in parsing research when using MaltParser as a point of comparison for other systems and/or for exploring different feature sets in a quick and convenient way. As already mentioned, it is possible to influence the optimization process at various points, which should make the system potentially useful to speed up the optimization process and hopefully also improve the final results.

We believe that it is possible to extend the process under MaltOptimizer to different transition-based parsers such as Zhang and Nivre (2011), Bohnet and Nivre (2012), or Choi and McCallum (2013). All these parsers use rich feature representations and are inherently based on the algorithms implemented in MaltParser. However, it is also true that these parsers use different learning algorithms and search strategies and may therefore benefit from different features. Therefore, it is likely that we would have to develop a new optimization strategy for each parser.

Other 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 fully greedy approach. Another idea is to allow the user to define customized input/output formats containing other feature types than the ones defined in the CoNLL format. For example, instead of lumping all morphosyntactic features together in the list-valued FEATS field, one could define a different field for each individual feature (case, number, gender, tense, aspect, *etc.*), which would allow a much more fine-grained exploration of these features. Finally, it would be interesting to see whether machine learning could be used to infer potential features based on the results of the data analysis, rather than applying the same feature selection steps for all data sets.

References

Agirre, E., Atutxa, A., and Sarasola, K. 12/2012. Contribution of complex lexical information to solve syntactic ambiguity in Basque. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, Mumbai, India, pp. 97–113.

Agirre, E., Bengoetxea, K., Gojenola, K., and Nivre, J. 2011. Improving dependency parsing with semantic classes. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL)*, Stroudsburg, PA, pp. 699–703.

Ballesteros, M. 2013a. Effective morphological feature selection with MaltOptimizer at the SPMRL 2013 shared task. In *Proceedings of the Fourth Workshop on Statistical Parsing of Morphologically Rich Languages (SPMRL)*, Stroudsburg, PA, pp. 63–70.

Ballesteros, M. 2013b. Exploring automatic feature selection for transition-based dependency parsing. *Procesamiento del Lenguaje Natural* **51**: 119–26.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Ballesteros, M., and Nivre, J. 2012a. MaltOptimizer: a system for MaltParser optimization. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC)*, Paris, pp. 2757–63.

Ballesteros, M., Gómez-Rodríguez, C., and Nivre, J. 2012. Optimizing planar and 2-planar parsers with MaltOptimizer. *Procesamiento del Lenguaje Natural* **49**: 171–8.

Ballesteros, M., Mille, S., and Burga, A. 2013. Exploring morphosyntactic annotation over a Spanish corpus for dependency parsing. In *Proceedings of the Second International Conference on Dependency Linguistics (DEPLING)*, Prague, pp. 13–22.

Ballesteros, M., and Nivre, J. 2012b. 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)*, Stroudsburg, PA, pp. 58–62.

Bohnet, B., and Nivre, J. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, Stroudsburg, PA, pp. 1455–65.

Buchholz, S., and Marsi, E. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, Stroudsburg, PA, pp. 149–64.

Cassel, S. 2009. *MaltParser and LIBLINEAR: Transition-Based Dependency Parsing with Linear Classification for Feature Model Optimization.* M.Phil. thesis, Uppsala University, Uppsala, Sweden.

Chang, C.-C., and Lin, C.-J. 2011. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* **2**: 1–27. Software available at `http://www.csie.ntu.edu.tw/ cjlin/libsvm`.

Choi, J. D., and McCallum, A. 2013. Transition-based dependency parsing with selectional branching. *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL)*, Stroudsburg, PA, pp. 1052–62.

Cirik, V., and Sensoy, H. 2013. The AI-KU system at the SPMRL 2013 shared task: unsupervised features for dependency parsing. In *Proceedings of the Fourth Workshop on Statistical Parsing of Morphologically Rich Languages (SPMRL)*, Stroudsburg, PA, pp. 79–85.

Covington, M. A. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, New York, NY, pp. 95–102.

Crammer, K., Dekel, O., Keshet, J., Shalev-Shwartz, S., and Singer, Y. 2006. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, **7**, 551–85. MIT Press.

Daelemans, W., Hoste, V., Meulder, F. De, and Naudts, B. 2003. Combined optimization of feature selection and algorithm parameters in machine learning of language. *Proceedings of the 14th European Conference on Machine Learning (ECML)*, Berlin, pp. 84–95.

Das, A., and Kempe, D. 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)*, Madison, WI, pp. 1057–64.

Doraisamy, S., Golzari, S., Norowi, N. M., Sulaiman, Md N., and Udzir, N. I. 2008. A study on feature selection and classification techniques for automatic genre classification of traditional malay music. *Proceedings of the Ninth International Conference on Music Information Retrieval (ISMIR)*, Philadelphia, PA, pp. 331–6.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. 2008. Liblinear: a library for large linear classification. *Journal of Machine Learning Research* **9**: 1871–4.

Goenaga, I., Gojenola, K., and Ezeiza, N. 2013. Exploiting the contribution of morphological information to parsing: the BASQUE TEAM system in the SPMRL 2013 shared task. In *Proceedings of the Fourth Workshop on Statistical Parsing of Morphologically Rich Languages (SPMRL)*, Stroudsburg, PA, pp. 71–7.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Gómez-Rodríguez, C., and Fernández-González, D. 2012. Dependency parsing with undirected graphs. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, Stroudsburg, PA, pp. 66–76.

Gómez-Rodríguez, C., and Nivre, J. 2010. A transition-based parser for 2-planar dependency structures. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, Stroudsburg, PA, pp. 1492–501.

Guyon, I., and Elisseeff, A. 2003. An introduction to variable and feature selection. *Journal of Machine Learning Research* **3**: 1157–82. MIT Press.

Hall, J 2008. *Transition-Based Natural Language Parsing with Dependency and Constituency Representations*. PhD thesis, Växjö University, Sweden.

Hall, J., Nilsson, J., Nivre, J., Eryiğit, G., Megyesi, B., Nilsson, M., and Saers, M. 2007. Single malt or blended? A study in multilingual parser optimization. In *Proceedings of the CoNLL Shared Task at the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CONLL)*, Stroudsburg, PA, pp. 933–9.

Kool, A., Zavrel, J., and Daelemans, W. 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)*, Tilburg, pp. 93–100.

Korycinski, D., Crawford, M., Barnes, J. W., and Ghosh, J. 2003. Adaptive feature selection for hyperspectral data analysis. In *Proceedings of the SPIE Conference on Image and Signal Processing for Remote Sensing IX*, Bellingham, WA, pp. 213–25.

Mambrini, F., and Passarotti, M. C. 2012. Will a parser overtake Achilles? First experiments on parsing the ancient Greek dependency treebank. In *Proceedings of the Eleventh International Workshop on Treebanks and Linguistic Theories (TLT11)*, Sofia, pp. 133–44.

McCallum, A 2003. Efficiently inducing features of conditional random fields. In *Proceedings of the Conference on Uncertainty in AI*, Burlington, MA, pp. 403–10.

McDonald, R 2006. *Discriminative Learning and Spanning Tree Algorithms for Dependency Parsing*. PhD thesis, University of Pennsylvania, Philadelphia, PA.

McDonald, R., Lerman, K., and Pereira, F. 2006. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, Stroudsburg, PA, pp. 216–20.

McDonald, R., and Nivre, J. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, Stroudsburg, PA, pp. 122–31.

Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill, New York, NY.

Nilsson, P., and Nugues, P. 2010. Automatic discovery of feature sets for dependency parsing. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING)*, Beijing, China, pp. 824–32.

Nivre, J. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pp. 149–60.

Nivre, J. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together at the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*, Stroudsburg, PA, pp. 50–7.

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)*, Stroudsburg, PA, pp. 396–403.

Nivre, J. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics* **34**(4): 513–53. MIT Press.

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*

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press

Conference on Natural Language Processing of the AFNLP (ACL-IJCNLP), Stroudsburg, PA, pp. 351–9.

Nivre, J., and Hall, J. 2010. *A Quick Guide to MaltParser Optimization*. Tech. rept. 1. MaltParser (maltparser.org).

Nivre, J., Hall, J., Kübler, S., McDonald, R., Nilsson, J., Riedel, S., and Yuret, D. 2007a. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task at the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CONLL)*, Stroudsburg, PA, pp. 915–32.

Nivre, J., Hall, J., and Nilsson, J. 2006b. Maltparser: a data-driven parser-generator for dependency parsing. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, Paris, pp. 2216–9.

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

Nivre, J., Hall, J., Nilsson, J., Eryiğit, G., and Marinov, S. 2006a. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, Stroudsburg, PA, pp. 221–5.

Nivre, J., Kuhlmann, M., and Hall, J. 2009. An improved oracle for dependency parsing with online reordering. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT)*, Stroudsburg, PA, pp. 73–6.

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

Prudhvi Kosaraju, Kesidi, S. R., Ainavolu, V. B. R. and Kukkadapu, P. 2010. Experiments on Indian language dependency parsing. In *Proceedings of the ICON-2010 Tools Contest on Indian Language Dependency Parsing*, Kharagpur, India, pp. 40–45.

Seddah, D., Tsarfaty, R., Kübler, S., Candito, M., Choi, J. D., Farkas, R., Foster, J., Goenaga, I., Galletebeitia, K. G., Goldberg, Y., Green, S., Habash, N., Kuhlmann, M., Maier, W., Marton, Y., Nivre, J., Przepiórkowski, A., Roth, R., Seeker, W., Versley, Y., Vincze, V., Woliński, M., and Wróblewska, A. 2013. Overview of the SPMRL 2013 shared task: a cross-framework evaluation of parsing morphologically rich languages. In *Proceedings of the Fourth Workshop on Statistical Parsing of Morphologically Rich Languages (SPMRL)*, Stroudsburg, PA, pp. 146–82.

Seraji, M., Megyesi, B., and Nivre, J. 2012. Dependency parsers for persian. In *Proceedings of 10th Workshop on Asian Language Resources, at 24th International Conference on Computational Linguistics (COLING)*, Mumbai, India, pp. 35–43.

Tsarfaty, R., Nivre, J., and Andersson, E. 2012. Cross-framework evaluation for statistical parsing. In *Proceedings of the European Chapter of the Association for Computational Linguistics (EACL)*, Stroudsburg, PA, pp. 44–54.

van den Bosch, A. 2004. Wrapped progressive sampling search for optimizing learning algorithm parameters. In *Proceedings of the 16th Belgian-Dutch Conference on Artificial Intelligence*, Groningen, The Netherlands, pp. 219–26.

Yli-Jyrä, A. M. 2003. Multiplanarity – a model for dependency structures in treebanks. In *Proceedings of the Second Workshop on Treebanks and Linguistic Theories (TLT)*, Mathematical Modelling in Physics, Engineering and Cognitive Sciences, vol. 9, pp. 189–200. Växjö, Sweden: Växjö University Press.

Zhang, Y., and Nivre, J. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, Stroudsburg, PA, pp. 188–93.

https://doi.org/10.1017/S1351324914000035 Published online by Cambridge University Press