MaltParser: A Data-Driven Parser-Generator for Dependency Parsing

Joakim Nivre **Johan Hall** **Jens Nilsson**

Växjö University
School of Mathematics and Systems Engineering
351 95 Växjö
{joakim.nivre, johan.hall, jens.nilsson}@msi.vxu.se

Abstract

We introduce MaltParser, a data-driven parser generator for dependency parsing. Given a treebank in dependency format, MaltParser can be used to induce a parser for the language of the treebank. MaltParser supports several parsing algorithms and learning algorithms, and allows user-defined feature models, consisting of arbitrary combinations of lexical features, part-of-speech features and dependency features. MaltParser is freely available for research and educational purposes and has been evaluated empirically on Swedish, English, Czech, Danish and Bulgarian.

1. Introduction

One of the alleged advantages of data-driven approaches to natural language processing is that development time can be much shorter than for systems that rely on hand-crafted resources in the form of lexicons or grammars. However, this is possible only if the data resources required to train or tune the data-driven system are already available. For instance, all the more successful data-driven approaches to syntactic parsing presuppose the existence of a treebank, a kind of annotated corpus that is relatively expensive to produce. And while the number of languages for which treebanks are available is growing steadily, the size of these treebanks is seldom larger than 100k tokens or thereabout. Hence, in order to capitalize on the potential advantages of data-driven parsing methods, we need metods that can give good accuracy without requiring huge amounts of syntactically annotated data.

In this paper, we present a system for data-driven dependency parsing that has been applied to several languages, consistently giving a dependency accuracy of 80–90%, while staying within a 5% increase in error rate compared to state-of-the-art parsers without any language-specific enhancements and with fairly modest data resources (on the order of 100k tokens or less). The empirical evaluation of the system, using data from Swedish, English, Czech, Danish and Bulgarian, has been described elsewhere (Nivre and Hall, 2005). In this paper, we will concentrate on the functionality provided in the system, in particular the availability of different parsing algorithms, learning algorithms, and feature models, which can be varied and optimized independently of each other. MaltParser 0.3 is freely available for research and educational purposes.[1]

The paper is structured as follows. Section 2. presents the underlying parsing methodology, known as *inductive dependency parsing*. Section 3. describes the parsing algorithms supported by the system and introduces the abstract data structures needed for the definition of features for machine learning. Section 4. explains how features can be defined by the user in order to create customized feature models, and section 5. briefly describes the learning algorithms available in the system. Section 6. deals with the two modes of the system, *learning* and *parsing*, as well as input and output formats. Section 7. is our conclusion.

2. Inductive Dependency Parsing

MaltParser can be characterized as a data-driven parser-generator. While a traditional parser-generator constructs a parser given a grammar, a data-driven parser-generator constructs a parser given a treebank. MaltParser is an implementation of *inductive dependency parsing* (Nivre, 2005), where the syntactic analysis of a sentence amounts to the derivation of a dependency structure, and where inductive machine learning is used to guide the parser at nondeterministic choice points. This parsing methodology is based on three essential components:

1. Deterministic parsing algorithms for building dependency graphs (Yamada and Matsumoto, 2003; Nivre, 2003)
2. History-based feature models for predicting the next parser action (Black et al., 1992; Magerman, 1995; Ratnaparkhi, 1997; Collins, 1999)
3. Discriminative machine learning to map histories to parser actions (Yamada and Matsumoto, 2003; Nivre et al., 2004)

Given the restrictions imposed by these components, MaltParser has been designed to give maximum flexibility in the way components can be varied independently of each other. Sections 3.–5. describe the functionality for each of the components in turn. Section 6. then describes how the system as a whole is used for learning and parsing.

3. Parsing Algorithms

Any deterministic parsing algorithm compatible with the MaltParser architecture has to operate with the following set of data structures, which also provide the interface to the feature model:

- A stack STACK of partially processed tokens, where STACK[i] is the i+1th token from the top of the stack, with the top being STACK[0].

[1] URL: http://www.msi.vxu.se/users/nivre/MaltParser.html

- A list INPUT of remaining input tokens, where INPUT[i] is the $i+1$th token in the list, with the first token being INPUT[0].

- A stack CONTEXT of unattached tokens occurring between the token on top of the stack and the next input token, with the top CONTEXT[0] being the token closest to STACK[0] (farthest from INPUT[0]).

- A function HEAD defining the partially built dependency structure, where HEAD[i] is the syntactic head of the token i (with HEAD[i] = 0 if i is not yet attached to a head).

- A function DEP labeling the partially built dependency structure, where DEP[i] is the dependency type linking the token i to its syntactic head (with DEP[i] = ROOT if i is not yet attached to a head).

- A function LC defining the leftmost child of a token in the partially built dependency structure (with LC[i] = 0 if i has not left children).

- A function RC defining the rightmost child of a token in the partially built dependency structure (with RC[i] = 0 if i has not right children).

- A function LS defining the next left sibling of a token in the partially built dependency structure (with LS[i] = 0 if i has no left siblings).

- A function RS defining the next right sibling of a token in the partially built dependency structure (with RS[i] = 0 if i has no right siblings).

An algorithm builds dependency structures incrementally by updating HEAD and DEP, but it can only add a dependency arc between the top of the stack (STACK[0]) and the next input token (INPUT[0]) in the current configuration. (The context stack CONTEXT is therefore only used by algorithms that allow non-projective dependency structures, since unattached tokens under a dependency arc are ruled out in projective dependency structures.) MaltParser 0.3 provides two basic parsing algorithms, each with two options:

- Nivre's algorithm (Nivre, 2003) is a linear-time algorithm limited to projective dependency structures. It can be run in *arc-eager* or *arc-standard* mode (Nivre, 2004).

- Covington's algorithm (Covington, 2001) is a quadratic-time algorithm for unrestricted dependency structures, which proceeds by trying to link each new token to each preceding token. It can be run in a *projective* mode, where the linking operation is restricted to projective dependency structure, or in a *non-projective* mode, allowing non-projective (but acyclic) dependency structures. In non-projective mode, the algorithm uses the CONTEXT stack to store unattached tokens occurring between STACK[0] and INPUT[0] (from right to left).

The empirical evaluation reported in Nivre and Hall (2005) is based on the arc-eager version of Nivre's algorithm, which has so far given the highest accuracy for all languages and data sets.

4. Feature Models

MaltParser uses history-based feature models for predicting the next action in the deterministic derivation of a dependency structure, which means that it uses features of the partially built dependency structure together with features of the (tagged) input string. More precisely, features are defined in terms of the word form (LEX), part-of-speech (POS) or dependency type (DEP) of a token defined relative to one of the data structures STACK, INPUT and CONTEXT, using the auxiliary functions HEAD, LC, RC, LS and RS.

A feature model is defined in an external *feature specification* with the following syntax:

```
<fspec>  ::= <feat>+
<feat>   ::= <lfeat>|<nlfeat>
<lfeat>  ::= LEX\t<dstruc>\t<off>\t<suff>\n
<nlfeat> ::= (POS|DEP)\t<dstruc>\t<off>\n
<dstruc> ::= (STACK|INPUT|CONTEXT)
<off>    ::= <nnint>\t<int>\t<nnint>
             \t<int>\t<int>
<suff>   ::= <nnint>
<int>    ::= (...|-2|-1|0|1|2|...)
<nnint>  ::= (0|1|2|...)
```

As syntactic sugar, any `<lfeat>` or `<nlfeat>` can be truncated if all remaining integer values are zero.

Each feature is specified on a single line, consisting of at least two tab-separated columns. The first column defines the feature type to be lexical (LEX), part-of-speech (POS), or dependency (DEP). The second column identifies one of the main data structures in the parser configuration, usually the stack (STACK) or the list of remaining input tokens (INPUT), as the "base address" of the feature. (The third alternative, CONTEXT, is relevant only together with Covington's algorithm in non-projective mode.) The actual address is then specified by a series of "offsets" with respect to the base address as follows:

- The third column defines a list offset i, which has to be non-negative and which identifies the $i+1$th token in the list/stack specified in the second column (i.e. STACK[i], INPUT[i] or CONTEXT[i]).

- The fourth column defines a linear offset, which can be positive (forward/right) or negative (backward/left) and which refers to (relative) token positions in the original input string.

- The fifth column defines an offset i in terms of the HEAD function, which has to be non-negative and which specifies i applications of the HEAD function to the token identified through preceding offsets.

- The sixth column defines an offset i in terms of the LC or RC function, which can be negative ($|i|$ applications of LC), positive (i applications of RC), or zero (no applications).

- The seventh column defines an offset i in terms of the LS or RS function,which can be negative ($|i|$ applications of LS), positive (i applications of RS), or zero (no applications).

Let us consider a few examples:

```
POS   STACK 0    0    0    0    0
POS   INPUT 1    0    0    0    0
POS   INPUT 0    -1   0    0    0
DEP   STACK 0    0    1    0    0
DEP   STACK 0    0    0    -1   0
```

The feature defined on the first line is simply the part-of-speech of the token on top of the stack (TOP). The second feature is the part-of-speech of the token immediately after the next input token in the input list (NEXT), while the third feature is the part-of-speech of the token immediately before NEXT in the original input string (which may not be present either in the INPUT list or the STACK anymore). The fourth feature is the dependency type of the head of TOP (zero steps down the stack, zero steps forward/backward in the input string, one step up to the head). The fifth and final feature is the dependency type of the leftmost dependent of TOP (zero steps down the stack, zero steps forward/backward in the input string, zero steps up through heads, one step down to the leftmost dependent). Using the syntactic sugar of truncating all remaining zeros, these five features can also be specified more succintly:

```
POS   STACK
POS   INPUT 1
POS   INPUT 0    -1
DEP   STACK 0    0    1
DEP   STACK 0    0    0    -1
```

The only difference between lexical and non-lexical features is that the specification of lexical features may contain an eighth column specifying a suffix length n. By convention, if n = 0, the entire word form is included; otherwise only the n last characters are included in the feature value. Thus, the following specification defines a feature the value of which is the four-character suffix of the word form of the next left sibling of the rightmost dependent of the head of the token immediately below TOP.

```
LEX   STACK 1    0    1    1    -1   4
```

Finally, it is worth noting that if any of the offsets is undefined in a given configuration, the feature is automatically assigned a null value.

Although the feature model must be optimized individually for each language and data set, there are certain features that have proven useful for all language investigated so far (Nivre and Hall, 2005). In particular:

- Part-of-speech features for TOP and NEXT, as well as a lookahead of 1–3 tokens.

- Dependency features for TOP, its leftmost and rightmost dependents, and the leftmost dependents of NEXT.

- Lexical features for at least TOP and NEXT, possibly also the head of TOP and one lookahead token.

Combining these features gives the following standard model (informally known as model 7):

```
POS   STACK
POS   INPUT
POS   INPUT 1
POS   INPUT 2
POS   INPUT 3
POS   STACK 1
DEP   STACK
DEP   STACK 0    0    0    -1
DEP   STACK 0    0    0    1
DEP   INPUT 0    0    0    -1
LEX   STACK
LEX   INPUT
LEX   INPUT 1
LEX   STACK 0    0    1
```

For very small datasets, it may be useful to exclude the last two lexical features, as well as one or more POS features at the periphery, in order to counter the sparse data problem. Alternatively, lexical features be defined as suffix features, where a suffix length of 6 characters often gives good results (Nivre and Hall, 2005).

5. Learning Algorithms

Inductive dependency parsing requires a learning algorithm to induce a mapping from parser histories, relative to a given feature model, to parser actions, relative to a given parsing algorithm. MaltParser 0.3 comes with two different learning algorithms, each with a wide variety of parameters:

- Memory-based learning and classification (Daelemans and Van den Bosch, 2005) stores all training instances at learning time and uses some variant of k-nearest neighbor classification to predict the next action at parsing time. MaltParser uses the software package TiMBL to implement this learning algorithm, and supports all the options provided by that package.

- Support vector machines rely on kernel functions to induce a maximum-margin hyperplane classifier at learning time, which can be used to predict the next action at parsing time. MaltParser uses the library LIBSVM (Chang and Lin, 2001) to implement this algorithm with all the options provided by this library.

All the published results for MaltParser so far are based on memory-based learning. However, given the competitive results achieved with support vector machines by Yamada and Matsumoto (2003), among others, it is likely that this will change in the future.

6. Learning and Parsing

MaltParser can be run in two modes:

- In *learning mode* the system takes as input a dependency treebank and induces a classifier for predicting parser actions, given specifications of a parsing algorithm, a feature model and a learning algorithm.

- In *parsing mode* the system takes as input a set of sentences and constructs a projective dependency graph for each sentence, using a classifier induced in learning mode (and the same parsing algorithm and feature model that were used during learning).

The input (during both learning and parsing) must be in the Malt-TAB format, which represents each token by one line, with tabs separating word form, part-of-speech tag, and (during learning) head and dependency type, and with blank lines representing sentence boundaries, as follows:

```
This    DT      2       SBJ
is      VBZ     0       ROOT
an      DT      4       DET
old     JJ      4       NMOD
story   NN      2       PRD
.       .       2       P

So      RB      2       PRD
is      VBZ     0       ROOT
this    DT      2       SBJ
.       .       2       P
```

The output can be produced in the same format or in two different XML formats (Malt-XML, TIGER-XML). The same sentences represented in Malt-XML would look as follows:

```
<sentence>
</sentence>
  <word form="This" postag="DT" head="2" deprel="SBJ"/>
  <word form="is" postag="VBZ" head="0" deprel="ROOT"/>
  <word form="an" postag="DT" head="4" deprel="DET"/>
  <word form="old" postag="JJ" head="4" deprel="NMOD"/>
  <word form="story" postag="NN" head="2" deprel="PRD"/>
  <word form="." postag="." head="2" deprel="P"/>
<sentence>
  <word form="So" postag="RB" head="2" deprel="PRD"/>
  <word form="is" postag="VBZ" head="0" deprel="ROOT"/>
  <word form="this" postag="DT" head="2" deprel="SBJ"/>
  <word form="." postag="." head="2" deprel="P"/>
</sentence>
```

7. Conclusion

In this paper, we have described the functionality of MaltParser, a data-driven parser-generator for dependency parsing, which can be used to create a parser for a new language given a dependency treebank representing that language. The system allows the user to choose between different parsing algorithms and learning algorithms and to define arbitrarily complex feature models in terms of lexical features, part-of-speech features and dependency type features.

8. References

E. Black, F. Jelinek, J. Lafferty, D. Magerman, R. Mercer, and S. Roukos. 1992. Towards history-based grammars: Using richer models for probabilistic parsing. In *Proceedings of the 5th DARPA Speech and Natural Language Workshop*, pages 31–37.

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

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania.

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

Walter Daelemans and Antal Van den Bosch. 2005. *Memory-Based Language Processing*. Cambridge University Press.

David M. Magerman. 1995. Statistical decision-tree models for parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 276–283.

Joakim Nivre and Johan Hall. 2005. MaltParser: A language-independent system for data-driven dependency parsing. In *Proceedings of the Fourth Workshop on Treebanks and Linguistic Theories (TLT)*.

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

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

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In Frank Keller, Stephen Clark, Matthew Crocker, and Mark Steedman, editors, *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together (ACL)*, pages 50–57.

Joakim Nivre. 2005. *Inductive Dependency Parsing of Natural Language Text*. Ph.D. thesis, Växjö University.

Adwait Ratnaparkhi. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1–10.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In Gertjan Van Noord, editor, *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 195–206.