A generic architecture for data-driven dependency parsing

Johan Hall and Joakim Nivre

Växjö University
School of Mathematics and Systems Engineering
{jha,nivre}@msi.vxu.se

Abstract

We present a software architecture for data-driven dependency parsing of unrestricted natural language text, which achieves a strict modularization of parsing algorithm, feature model and learning method such that these parameters can be varied independently. The design has been realized in MaltParser, which supports several parsing algorithms and learning methods, for which complex feature models can be defined in a special description language.

1 Introduction

One of the advantages of data-driven approaches to syntactic parsing is the relative ease with which they can be ported to new languages or domains, provided that the necessary linguistic data resources are available. Thus, the parser of Collins (1999), originally developed for English and trained on Wall Street Journal data, has been successfully applied to languages as different as Czech (Collins et al., 1999) and Chinese (Sun and Jurafsky, 2004). However, most available systems for data-driven syntactic parsing lack another kind of flexibility, namely the possibility to combine different parsing algorithms with different feature models and learning methods.

Data-driven dependency parsing has recently been explored as a robust and efficient method for syntactic parsing of unrestricted natural language text (Yamada and Matsumoto, 2003; Nivre et al., 2004). Dependency parsing means that the goal of the parsing process is to construct a dependency graph, of the kind depicted in Figure 1. The 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)
3. Discriminative machine learning to map histories to parser actions (Yamada and Matsumoto, 2003; Nivre et al., 2004)

Given the restriction imposed by these components, we present a software design for data-driven dependency parsing of unrestricted natural language text. The most important feature of the design is a clean separation of parsing algorithms, feature models, and learning methods, so that these components can be varied independently of each other. Moreover, the design makes it possible to use the same basic components both for inducing a model from treebank data in the learning phase and for using the model to parse new data in the parsing phase. This architecture has been realized in the MaltParser system.[1]

This paper is structured as follows. Section 2 gives the necessary background and introduces the framework of *inductive dependency parsing* (Nivre, 2005). Section 3 presents the generic architecture for data-driven dependency parsing, and section 4 describes its realization in the MaltParser system. Finally, conclusions are presented in section 5.

2 Inductive dependency parsing

Given a set R of dependency types, we define a *dependency graph* for a sentence $x = (w_1, \ldots, w_n)$ to be a labeled directed graph $G =$

[1] MaltParser is freely available for research and educational purposes and can be downloaded from http://www.vxu.se/msi/users/nivre/research/MaltParser.html.

S. Werner (ed.), *Proceedings of the 15th NODALIDA conference, Joensuu 2005*, Ling@JoY 1, 2006, pp. 47–56
ISBN 952-458-771-8, ISSN 1796-1114. http://ling.joensuu.fi/lingjoy/

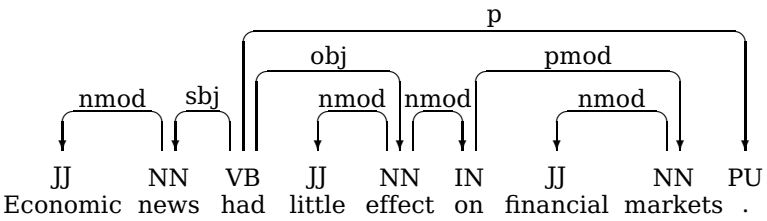

Figure 1: Dependency structure for English sentence

(V, E, L), where V is the set of input tokens $w_1, \ldots, w_n$, extended with a special root node w_0 and ordered by a linear precedence relation $<$; $E \subseteq V \times V$ is a set of directed arcs; and $L : E \rightarrow R$ is a function that labels arcs with dependency types. A dependency graph G is *well-formed* iff (i) the node w_0 is a root of G, (ii) G is connected, (iii) every node in G has an indegree of at most 1, and (iv) G is acyclic. Dependency parsing is the task of mapping sentences to well-formed dependency graphs.

Inductive approaches to natural language parsing can in general be defined in terms of three essential components:

1. A formal model M defining permissible representations for sentences (such as the model of dependency graphs defined above).
2. A parameterized stochastic model M_Θ, defining a score $S(x, y)$ for every sentence x and well-formed representation y.
3. An inductive learning algorithm L for estimating the parameters Θ from a representative sample $T_t = (x_1 : y_1, \ldots, x_n : y_n)$ of sentences with their correct representations (normally a treebank sample).

Inductive dependency parsing is compatible with a variety of different models, but we focus here on *history-based* models (Black et al., 1992; Magerman, 1995; Ratnaparkhi, 1997; Collins, 1999), which can be defined in three steps:

1. Define a one-to-one mapping between syntactic representations y and decision sequences $D = (d_1, \ldots, d_m)$ such that D uniquely determines y.
2. Define the score $S(x, y)$, for every sentence x and representation y, in terms of each decision d_i in the decision sequence $D = (d_1, \ldots, d_m)$, conditioned on the history $H = (d_1, \ldots, d_{i-1})$.
3. Define a function Φ that groups histories into equivalence classes, thereby reducing the number of parameters in Θ.

In a conditional history-based model, the score $S(x, y)$ defined by the model is the conditional probability $P(y \mid x)$ of the analysis y given the sentence x, which means that the input sentence is a conditioning variable for each decision in the decision sequence:

$$P(y \mid x) = P(d_1, \ldots, d_m \mid x) = \prod_{i=1}^{m} P(d_i \mid \Phi(d_1, \ldots, d_{i-1}, x))$$

The parameters of this model are the conditional probabilities $P(d \mid \Phi(H, x))$, for all possible decisions d and non-equivalent conditions $\Phi(H, x)$.

Given a conditional history-based model, the conditional probability $P(y_j \mid x)$ of analysis y_j given input x can be used to rank a set of alternative analyses $\{y_1, \ldots, y_k\}$ of the input sentence x, derived by a nondeterministic parser. If the model allows a complete search of the analysis space, we can in this way be sure to find the analysis y_j that maximizes the probability $P(y_j \mid x)$ according to the model:

$$\arg\max_j P(y_j \mid x) = \arg\max_j \prod_{i=1}^{m} P(d_i \mid \Phi(d_1, \ldots, d_{i-1}, x))$$

With a deterministic parsing strategy, we instead try to find the most probable analysis y_j without exploring more than one decision sequence, based on the following approximation:

$$\max_j P(y_j \mid x) \approx$$

$$\prod_{i=1}^{m} \max_i P(d_i \mid \Phi(d_1, \ldots, d_{i-1}, x))$$

A deterministic parsing strategy is in this context a *greedy algorithm*, making a locally optimal choice in the hope that this choice will lead to a globally optimal solution (Cormen et al., 1990). The main problem with the greedy strategy is that it may not lead to a globally optimal solution. The main advantage of the greedy strategy is that it improves parsing efficiency by avoiding an exhaustive search of the analysis space, but an additional advantage is that it reduces the effective number of parameters of the stochastic model, since only the mode of the distribution $P(d_i \mid \Phi(H, x))$ needs to be estimated for each distinct condition $\Phi(H, x)$. This also means that a larger class of learning methods can be used, including purely discriminative methods.

With a discriminative learning method we can reduce the learning problem to a pure classification problem, where an *input instance* is a parameterized history $\Phi(H, x)$, which is represented by a feature vector, and an *output class* is a decision d. Using a supervised learning method, our task is then to induce a classifier C given a set of training instances T, derived from a treebank sample, where O is an oracle function that predicts the correct decision given the gold standard treebank:

$$T = \{(\Phi(H, x), d) \mid O(H, x) = d, x \in T_t\}$$

In order to construct a specific instance of the inductive dependency parser, we therefore need to specify three things:

1. A deterministic *parsing algorithm* used to derive dependency graphs, which defines the set $\mathcal{D}$ of permissible decisions, as well as the oracle function O that determines the correct decision given a certain history H and input sentence x.
2. A parameterization function Φ used to define equivalence classes of histories and sentences in terms of a feature vector $\Phi_{(1,p)} = (\phi_1, \ldots, \phi_p)$, where each feature ϕ_i is a function that maps a token to its part-of-speech, lexical form or dependency type (in the partially built dependency graph). We call $\Phi_{(1,p)}$ a *feature model*.
3. A discriminative *learning algorithm* used to approximate the mode function $f(\Phi(H, x)) = \arg\max_d P(d \mid \Phi(H, x))$ given a set T of training instances.

We will now turn to the description of an architecture that allows the user to construct such an instance in an efficient and flexible manner, given a suitable sample of a dependency treebank.

3 A generic architecture

We propose an architecture with a strict modularization of parsing algorithms, feature models and learning methods, thereby giving maximum flexibility in the way these components can be varied independently of each other. The architecture can be seen as a data-driven parser-generator framework, which constructs a parser without rebuilding the framework given a treebank and a specified parsing algorithm, feature model, learning method. The idea is to give the user the flexibility to experiment with the components in a more convenient way, although there are still dependencies between components, in the sense that not all combinations will perform well with respect to accuracy and efficiency.

The design of the architecture deals also with the fact that the parser can be executed in two different phases: *the learning phase* and *the parsing phase*. In the learning phase, the system uses a treebank to learn a model; in the parsing phase, it takes a previously learnt model and uses this to parse new and unseen data. Although these two tasks have a different structure, they often involve similar or even identical subproblems. For instance, learning normally requires the construction of a gold standard parse derivation in order to estimate the parameters of a model, which is exactly the same kind of derivation that is constructed during parsing.

The architecture consists of three main components:

1. Parser, which derives the dependency graph for a sentence (during both learning and parsing) using a deterministic parsing algorithm.
2. Guide, which extracts feature vectors from the current state of the system (according to the specified model) and passes data between the Parser and the Learner.

3. Learner, which is responsible for inducing a model during learning time and for using this model to guide the Parser to make decisions at all nondeterministic choice points during the parsing phase.

In addition to the three main components the framework includes input/output components and an overall control structure. These components are responsible for reading a text $S = (x_1, \ldots, x_n)$ and invoking the Parser for each sentence $x_i \in S$. When the Parser has constructed the dependency graph G_i, the dependency graph should be outputted in a suitable format. The next three subsections describe the main components in more detail.

3.1 Parser

The Parser is responsible for the derivation of a well-formed dependency graph G_i for a sentence $x = w_1, \ldots, w_n$, given a set $R = \{r_0, r_1, \ldots, r_m\}$ of dependency types (r_0 is a special symbol for dependents of the root). To perform this derivation we need a parsing algorithm that can map a dependency graph into a sequence of decisions $D = (d_1, \ldots, d_m)$ during learning time by using the gold standard. Each decision will be passed on to the Guide as an example of a correct decision. During parsing time the parsing algorithm constructs a dependency graph by asking the Guide for each nondeterministic decision d_i. During both learning and parsing time, the *parser configuration* will be updated according to the decision. We can define a *parser configuration* for x as a sextuple $c = (\sigma, \tau, \upsilon, p, h, l)$, where:

1. σ is a stack of partially processed tokens.
2. τ is a list of remaining input tokens.
3. υ is a stack of unattached tokens occurring between the token on top of the stack σ and the next input token, called the context stack.
4. p is the part-of-speech function, where $p(w_i)$ is the part-of-speech of the token i.
5. h is the head function defining the partially built dependency structure, where $h(w_i)$ is the syntactic head of the token i (with $h(w_i) = 0$ if i is not yet attached to a head). h_g is the gold standard head function, where $h_g(w_i)$ is the syntactic head of the token i, taken from a treebank.
6. l is the dependency function labeling the partially built dependency structure, where $l(w_i)$ is the dependency type r_i linking the token i to its syntactic head (with $l(w_i) = r_0$ if i is not yet attached to a head). l_g is the gold standard dependency function, where $l_g(w_i)$ is the dependency type labeling the token i to its syntactic head, taken from a treebank.

Moreover, one of the main ideas of the Parser component is that it should be capable of incorporating several parsing algorithms, which allow the user of the system to choose a suitable algorithm for a specific purpose. Not all parsing algorithms are of course applicable for this architecture and therefore we need to define the restrictions that the parsing algorithm must fulfil:

1. It must be able to operate the configuration $c = (\sigma, \tau, \upsilon, p, h, l)$ of the form defined above for each sentence x.
2. It must only assign a dependency relation between the token on top of the stack σ and the next input token in the list τ.
3. It must be deterministic, in the sense that it always derives a single analysis for each sentence x. However, the algorithm can be nondeterministic in its individual choice points such as adding a dependency arc between two tokens or shifting a new token onto its stack. This implies that the algorithm must be able to break down the parsing problem into a sequence of decisions D. However, not all decisions have to be nondeterministic, for example when the stack σ is empty we can only shift the next token on to the stack.
4. It must be capable to make a gold standard parse given a training corpus $T_t = \{x_1, \ldots, x_n\}$ in the learning phase. In this way, the algorithm derives the correct decision in all nondeterministic choice points using an oracle function. This function defines the mapping between the dependency graph G_i and the decision sequences $D = (d_1, \ldots, d_m)$ such that D uniquely determines the dependency graph G_i.
5. It must define a set $\mathcal{D}_c$ of permissible decisions for each configuration c and check

that it does not perform an illegal decision. For example, if the stack σ is empty then it is not possible to pop the stack. To comply with the requirement of robustness, it must therefore provide a default decision according to the current state of the system if an illegal decision is proposed by the guide.

3.2 Guide

The Guide is responsible for constructing a set of training instances T for the Learner during the learning phase, and for passing the Learner's predictions to the Parser during the parsing phase.

At learning time, the Guide constructs one training instance $\Phi((H,x),d)$ for each decision d_i passed from the Parser, where $\Phi(H,x)$ is the current vector of feature values (given the parameterization function Φ and the current state of the system), and passes this on to the Learner. At parsing time, the Guide constructs a feature vector $\Phi(H,x)$ for each request from the Parser, sends it to the Learner and passes on the predicted decision d from the Learner to the Parser. In this way, the feature model is completely separated from the Parser, and the Learner only has to learn a mapping from feature vectors to decisions, without knowing either how the features are extracted or how the decisions are to be used. Moreover, the feature extraction is performed in exactly the same way during both learning and parsing.

The feature extraction uses the parameterization function Φ, which is defined in terms of a feature vector $\Phi_{(1,p)}$, where each feature ϕ_i is a function, defined in terms of three simpler functions: an *address function* a_{ϕ_i}, which identifies a specific token in a given parser configuration, an *attribute function* f_{ϕ_i}, which picks out a specific attribute of the token, and a *mapping function* m_{ϕ_i}, which defines equivalence classes of attribute values.

1. For every i, $i \geq 0$, $\sigma[i]$, $\tau[i]$ and $\upsilon[i]$ are address functions identifying the $i+1$th token from the top of the stack σ, the start of the input list τ, and the top of the stack υ, respectively. (Hence, $\sigma[0]$ is the top of the stack, $\tau[0]$ is the next input token and $\upsilon[0]$ is the top of the context stack.)
2. If α is an address function, then $h(\alpha)$, $lc(\alpha)$, $rc(\alpha)$, $ls(\alpha)$ and $rs(\alpha)$ are address functions, identifying the head (h), the leftmost child (lc), the rightmost child (rc), the next left sibling (ls) and the next right sibling (rs), respectively, of the token identified by α (according to the partially built dependency graph G_i).
3. If α is an address function, then $p(\alpha)$, $w(\alpha)$ and $l(\alpha)$ are feature functions, identifying the part-of-speech (p), word form (w) and dependency type (l) of the token identified by α (where the dependency type, if any, is given by the partially built dependency graph G_i). We call p, w and l attribute functions.

Given a feature function β, we can also define a mapping function that maps each value of β to a new value. For example, a mapping function can be used to restrict the value of a lexical feature ($w(\alpha)$) to the last n characters of the word form.

Given this parameterization function Φ, the Guide will extract a parser state $s_i = (v_1, \ldots, v_p)$ and passes this to the learner, where the v_j is extracted value for the corresponding feature ϕ_j. During learning it also provide the decision d_i , or request the d_i during parsing.

3.3 Learner

The Learner, finally, is responsible for inducing a classifier g from the set of training instances T by using a learning algorithm L. The *learned function* g is an approximation of the true oracle O. The set of possible decisions is discrete and finite, and therefore we can view this as *classification* problem. The classifier g is used to predict parser decisions given a parser state during the parsing phase. In practice, the Learner will normally be an interface to a standard machine learning package.

3.4 The learning and parsing phase

To conclude this section we will summarize the architecture by illustrating the data flow during the learning and parsing phase.

Figure 2 shows the architecture during the learning phase. The Parser takes as input a list of input tokens τ and the gold standard functions (h_g, l_g). For each nondeterministic decision the Parser use the oracle function O to derive the correct decision d_{i+1} and updates the configuration according to d_{i+1}, which results in the next configuration c_{i+1}. The Guide takes

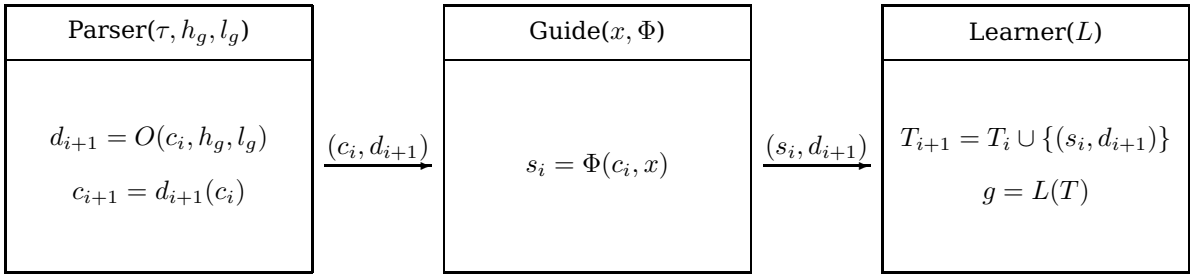

Figure 2: The data flow in the architecture during the learning phase.

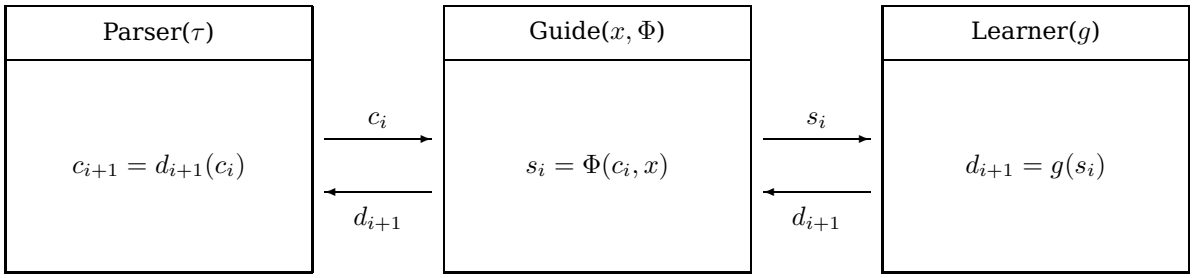

Figure 3: The data flow in the architecture during the parsing phase.

```
Genom           pp                      3   ADV
skattereformen  nn.utr.sin.def.nom      1   PR
införs          vb.prs.sfo              0   ROOT
individuell     jj.pos.utr.sin.ind.nom  5   ATT
beskattning     nn.utr.sin.ind.nom      3   SUB
(               pad                     5   IP
särbeskattning  nn.utr.sin.ind.nom      5   APP
)               pad                     5   IP
av              pp                      5   ATT
arbetsinkomster nn.utr.plu.ind.nom      9   PR
.               mad                     3   IP
```

Figure 4: An example sentence taken from the Swedish treebank Talbanken (Einarsson, 1976) in the Malt-TAB format. The first column is the word from, followed by the part-of-speech, the syntactic head and the dependency relation to the head (in that order).

as input a feature model Φ, and a sentence x_i. For each nondeterministic decision, the Guide uses the feature model Φ to extract the parser state s_i. Finally, the Learner collects the instance (s_i, d_{i+1}) passed from the Guide in the training set T. When all training instances are collected the learning algorithm L is applied to induce a classifier g.

The architecture during the parsing phase is shown in figure 3. The Parser takes as input a list of input tokens τ. For each nondeterministic decision the Parser requests a prediction of the next decision from the Guide and updates the configuration according to d_{i+1}, which results in the next configuration c_{i+1}. The Guide acts as a middle layer between the Parser and the Learner by extracting the parser state $s_i = \Phi(c_i, x)$ in the same way as in the learning phase and passing d_{i+1} from Learner to the Parser. The Learner uses the induced classifier g to predict the decision d_{i+1}.

4 Implementation

The architecture described in section 3 has been realized in the MaltParser system, which can be applied to a labeled dependency treebank in order to induce a labeled dependency parser for the language represented by the treebank.

The system takes as input a file in the Malt-TAB format, there each token is represented on one line, with attribute values being separated by tabs and each sentence separated by a blank line. Figure 4 shows an example of the Malt-TAB format. During the learning phase the parser requires all four columns, but during parsing it only needs the first two columns, i.e. the word form and the part-of-speech. The output of the MaltParser system is a sequence of labeled dependency graphs in the Malt-TAB format or the Malt-XML format. The latter is an XML version of the Malt-TAB format.

In the sections below, we describe the main components in the architecture from the point of view of how they are implemented in the MaltParser system.

4.1 Parser

The user of MaltParser can choose between several deterministic parsing algorithms, including the algorithms described by Nivre (2003; 2004) and the incremental parsing algorithms described by Covington (2001).

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

Covington's algorithm 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 structures, or in a *non-projective* mode, allowing non-projective (but acyclic) dependency structures.

4.2 Guide

The MaltParser system comes with a formal specification language for feature functions, which enables the user to define arbitrarily complex feature models in terms of address functions, attribute functions and mapping functions (cf. section 3.2). Each feature model is defined in a *feature specification* file, which allows users to define feature models without rebuilding the system. The feature specification uses the syntax described in Figure 5. Each feature is specified on a single line, consisting of at least two tab-separated columns. Below follows a description of each column:

1. Defines the feature type to be part-of-speech (`POS`), dependency (`DEP`) or word form (`LEX`), corresponding to the attribute functions p, l and w in section 3.

2. Identifies one of the main data structures in the parser configuration: `STACK` (corresponding to σ), `INPUT` (corresponding to τ or `CONTEXT` (corresponds to υ). The third alternative, `CONTEXT`, is relevant only together with Covington's algorithm in non-projective mode.

3. Defines a list offset i which can only be positive and which identifies the $i+1$th token in the list/stack specified in the second column (i.e. $\sigma[i]$, $\tau[i]$ or $\upsilon[i]$).

4. Defines a linear offset i, which can be positive (forward/right) or negative (backward/left) and which refers to (relative) token positions in the original input string.

5. Defines an offset i in terms of the function h (head), which has to be non-negative and

```
<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|...)
```

Figure 5: The syntax for defining a feature model in an external feature specification.

which specifies i applications of the h function to the token identified through preceding offsets.

6. Defines an offset i in terms of the functions lc (the leftmost child) or rc (the rightmost child), which can be negative ($|i|$ applications of lc), positive (i applications of rc), or zero (no applications).

7. Defines an offset i in terms of the functions ls (the next left sibling) or rs (the next right sibling), which can be negative ($|i|$ applications of ls), positive (i applications of rs), or zero (no applications).

8. If the first column specifies the attribute function w (LEX), the eighth column, defines a mapping function which specifies a suffix of length n of the word form w. By convention, if n = 0, the entire word form is included; otherwise only the n last characters are included in the feature value.

As syntactic sugar, any feature definition can be truncated if all remaining integer values are zero. Let us consider an example:

```
POS INPUT
DEP STACK 0 0 1
LEX INPUT 1
```

The first feature is the part-of-speech of the token located first in the list τ of remaining input tokens, i.e. $p(\tau[0])$. The feature defined on the second line is the dependency type of the head of the token located at the top of the stack σ (zero steps down the stack, zero steps forward/backward in the input string, one step up to the head), i.e. $l(h(\sigma[0]))$. Finally, the third feature is the word form of the token immediately after the next input token, i.e. $w(\tau[1])$.

When the Guide extracts the features according to the specification, it stores the feature values in a dedicated data structure. The Learner later iterates through this data structure in a linear fashion and outputs each feature in the format by a specific machine learning package.

4.3 Learner

In the current implementation this component is actually a set of interfaces to machine learning packages. The interfaces prepare the set of training instances, provided by the Guide, for the specific package and invoke the appropriate functions to learn a model or predict a decision. MaltParser comes with two different learning algorithms: Memory-based learning and Support Vector Machines (SVMs), each with a wide variety of parameters.

Memory-based learning and classification is based on the assumption that a cognitive learning task to a high degree depends on direct experience and memory, rather than extraction of an abstract representation. Solving new problems is achieved by reusing solutions from similar previously solved problems (Daelemans et al., 1999). During learning time, all training instances are stored and at parsing time a variant of k-nearest neighbor classification is used to predict the next action. MaltParser uses the software package TiMBL (Tilburg Memory-Based Learner) (Daelemans and Van den Bosch, 2005) to implement this learning algorithm, and supports all the options provided by that package.

Support Vector Machines (SVMs) was formulated in the late seventies by Vapnik (1979), but the main development as a machine learning approach has taken place in the last decade. SVMs have been used for many pattern recognition problems. In the field of natural language

processing, SVMs have been used for example for text categorization (Joachims, 1998) and syntactic dependency parsing (Kudo and Matsumoto, 2000). SVMs 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 (Wu et al., 2004) to implement this algorithm with all the options provided by this library.

5 Conclusion

We have presented a generic architecture for data-driven dependency parsing that provides a strict modularization of parsing algorithm, feature model and learning method. The main advantage of this design is that these three dimensions can be varied independently of each other, but the design also enables the reuse of components between the learning and the parsing phase. The design has been implemented in the MaltParser system, which is freely available for research and educational purposes, and which supports several parsing algorithms and learning methods, as well as a specification language for feature models that lets the user specify complex combinations of different types of features.

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.

Michael Collins, Jan Hajič, Lance Ramshaw, and Christoph Tillmann. 1999. A statistical parser for Czech. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 505–512.

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

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. MIT Press.

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.

Walter Daelemans, Antal van den Bosch, and Jakub Zavrel. 1999. Forgetting exceptions is harmful in language learning. *Machine Learning*, 34(1-3):11–41.

Jan Einarsson. 1976. *Talbankens skriftspråkskonkordans*. Lund University, Department of Scandinavian Languages.

T. Joachims. 1998. Text categorization with support vector machines. In *Proceedings of the 10th European Conference on Machine Learning (ECML'98)*, pages 137–142.

Taku Kudo and Yuji Matsumoto. 2000. Japanese Dependency Structure Analysis Based on Support Vector Machines. In *Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 18–25.

D. 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, 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 in 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.

Honglin Sun and Daniel Jurafsky. 2004. Shallow semantic parsing of Chinese. In *Proceedings of the Human Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, pages 249–256.

V. Vapnik. 1979. Estimation of dependences based on empirical data. Technical report, Nauka, Moscow.

T.-F. Wu, C.-J. Lin, and R. C. Weng. 2004. Probability estimates for multi-class classification by pairwise coupling. *Journal of Machine Learning Research*, 5:975–1005.

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.