Induction, Logic, and Natural Language Processing

Luc Dehaspe, Hendrik Blockeel, and Luc De Raedt

March 16, 1995

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
email : Luc.Dehaspe,Hendrik.Blockeel,Luc.DeRaedt@cs.kuleuven.ac.be
fax : ++ 32 16 32 75 39; telephone : ++ 32 16 32 75 50
WWW: http://www.cs.kuleuven.ac.be/~ml/MLRG.html

Abstract

While computational logic has become widely used for representing and reasoning with linguistic knowledge, the cross-fertilization between logic programming and machine learning has given rise to a new discipline known as inductive logic programming. Inspired by, and building on the achievements of logic programming within both natural language research and machine learning, we point out opportunities for induction of linguistic knowledge within logic (programming).

Keywords: inductive logic programming, natural language processing, logic programming, machine learning.

1 Introduction

There is a growing interest amongst both linguistic engineers and machine learning researchers for applying symbolic learning algorithms in natural language R&D[1]. Linguists confronted with the high cost of development of essential resources are drawn towards machine learning in search for generic technologies to exploit corpora for system training purposes. Vice versa machine learning researchers are attracted by the audacious idea of learning from text, and, on a more modest scale, by the variety of language processing problems and data available for experiments.

The applicability of a particular machine learning approach to a specific linguistic task obviously relies on the compatibility of the knowledge representation formalisms used

[1] Cf. for instance the joint ELSnet/MLnet workshop on Machine Learning of Natural Language and Speech held in Amsterdam on December 2-3, 1994; and SIGNLL, the ACL Special Interest Group on Natural Language Learning (WWW: http://www.cs.rulimburg.nl/~antal/signll/signll-home.html).

in both domains. In that respect a most favourable opportunity occurs when a single formalism is successfully employed within both natural language processing and machine learning. This is the case with computational logic, of which the expressive power, firm theoretical foundations, and declarative nature continue to appeal to natural language and machine learning researchers alike[2]. While computational logic has become widely used for representing and reasoning with linguistic knowledge, the cross-fertilization between logic programming and machine learning has given rise to a new discipline known as inductive logic programming [8, 10, 12].

Inspired by, and building on the achievements of logic programming within natural language processing and machine learning, we explore the intersection of the three disciplines. On a quick tour around inductive logic programming we first point out opportunities for the application of the different modes and settings to natural language research. One possible type of applications is then further illustrated in the second part of the paper, where we report on a small-scale experiment with learning Dutch morphology rules.

2 Inductive logic programming

Being a subfield of logic programming, inductive logic programming concerns the development of methods and tools for the automation of reasoning, i.e. the inference of new explicit knowledge on the basis of what is already known, within logic. As it focuses on inductive rules of inference, inductive logic programming is at the same time embedded in the inductive learning domain, which in turn is part of the broad field of machine learning.

The generic inductive logic programming task is to search a predefined subspace of clausal logic for a set of logical formulae that in some respect *explain* the data available in a clausal knowledge base. This knowledge base is traditionally subdivided into background knowledge and examples that represent positive and negative evidence of some concept to be learnt.

In a natural language R&D context background knowledge will typically be some more or less stable body of general linguistic rules, a grammar or a lexicon. A corpus can function as a source for positive examples, i.e. evidence of correct sentences, parse trees, semantic analyses, translations, etc... The negative evidence might consist of erroneous output produced by a natural language processing system under construction. The linguistic knowledge representation formalism defines the search space. The goal of the search is then a set of formulae, well-formed according to the linguistic formalism, that *explain* linguistic background knowledge and evidence.

In three subsections we will clarify the notion of explanation (*what* to search *for*), the organization of the search (*how* to search), and the definition of the search space (*what* to search).

[2]Cf. for instance a recent "ten years on" anniversary issue of the Journal of Logic Programming featuring two consecutive overview articles on the combinations *logic programming-machine learning* [12] and *logic programming-natural language processing* [3].

2.1 A model theory: what to search for

In this paragraph we present two alternative notions of *explanation* or semantics for in inductive logic programming, i.e. the *normal setting* and the *nonmonotonic setting*, cf. [12]. As a running example, we use the derivation and maintenance of a simple lexicon and phrase structure grammar encoded in the Definite Clause Grammar (DCG) formalism.

DCG is a notational extension of Prolog that makes it possible to write clauses, and grammar rules in particular, in a more compact way. The transformation from DCG to Prolog clauses roughly consists of adding two extra arguments to the head and all the goals in a systematic way. The first extra argument is an open ended list of which the last element unifies with the second extra argument. Some examples will make this clear[3]:

DCG rule	corresponding clause
$s \longrightarrow det$	$s(S0, S1) \leftarrow det(S0, S1)$
$s \longrightarrow np(X), verb(X, Y)$	$s(S0, S2) \leftarrow np(X, S0, S1) \wedge verb(X, Y, S1, S2)$
$noun \longrightarrow [john]$	$noun([john \mid S], S) \leftarrow$

2.1.1 Normal setting: concept learning

The normal semantics is the default setting of inductive logic programming used in systems such as MIS [17], FOIL [16], GOLEM [13], and PROGOL [18]. Characteristic is that both instances (positive evidence) and non-instances (negative evidence) of a concept are required. The aim is then to find a rule set that discriminates between these two classes.

Definition 1 *(normal semantics) Given background theory B, positive evidence E^+, negative evidence E^-, and formal language L, the aim is to find a hypothesis $H \subset L$ such that: $B \cup H \cup E^- \not\models \Box$ (consistency) and $B \cup H \models E^+$ (completeness). For instance Figure 1.*

In the example in Figure 1, grammar rules H_1 are induced given B_1 [4] and some evidence of correct (E_1^+) and incorrect (E_1^-) sentences. Notice that the rules in H_1 together with the grammar and lexicon in B_1 logically imply all the examples of E_1^+ (completeness) and none of E_1^- (consistency).

There are two special cases of the normal setting that seem to be particularly relevant to natural language processing: theory revision and abduction (understood as the induction of ground facts).

The theory revision problem can be cast as an instantiation of the normal setting where hypothesis H is initially non-empty.

Definition 2 *(theory revision) Given background theory B, negative evidence E^-, positive evidence E^+, formal language L, and a non-empty hypothesis H, the aim is to find a theory $H_{mod} \subset L$ obtained by applying transformations on H such that: $B \cup H_{mod} \cup E^- \not\models \Box$ (consistency) and $B \cup H_{mod} \models E^+$ (completeness). For instance Figure 2.*

$$B_1 = \boxed{\begin{array}{l} det \longrightarrow [the] \\ noun(sing3(+)) \longrightarrow [boy] \\ noun(sing3(+)) \longrightarrow [mary] \\ noun(sing3(-)) \longrightarrow [you] \\ verb(sing3(+), trans) \longrightarrow [likes] \\ verb(sing3(-), trans) \longrightarrow [like] \\ verb(sing3(+), intrans) \longrightarrow [sleeps] \\ verb(sing3(-), intrans) \longrightarrow [sleep] \\ \\ np(F) \longrightarrow noun(F) \end{array}}$$

$$E_1^- = \boxed{\begin{array}{l} \leftarrow s([likes, you, mary]) \\ \leftarrow s([the, boy, mary, sleeps])) \end{array}}$$

$$E_1^+ = \boxed{\begin{array}{l} s([the, boy, likes, you], []) \leftarrow \\ s([the, boy, sleeps], []) \leftarrow \end{array}}$$

$$H_1 = \boxed{\begin{array}{l} s \longrightarrow det, np(\_), verb(\_, \_) \\ s \longrightarrow det, np(\_), verb(\_, \_), np(\_) \end{array}}$$

Figure 1: Example of normal semantics

The lexicon $B2$ is the same as in B_1, and H_2 contains the grammar rules of B_1 and H_1. In H_{2mod}, the rules for np are extended so that they cover the new positive evidence in E_2^+. Notice that this extension together with negative evidence E_2^- triggers a modification of the rules for s. This is illustrative of the fact that in revision changes made to the definition of one predicate cascade through the total theory, potentially affecting other predicate definitions.

With theory revision the knowledge base can be updated incrementally. Thus on arrival of new negative evidence H_{2mod} can be further adapted, as is shown in Figure 2. In H_{3mod}, the $sing3$ features of subject noun and verb unify, and the features $intrans$ and $trans$ control the absence or presence of a direct object. We should mention here that in our illustrations all negative and positive examples are always generated and input by the user. Alternatively, the revision system might generate autonomously the most informative examples and prompt the user for their classification. This is done in interactive revision systems such as CLINT [4].

A final instantiation of the normal setting we would like to consider is the abductive case where facts are induced rather than ground rules. As can be seen in Figure 3, inducing facts in a natural language processing context corresponds to extending the lexicon on the basis of new language material and a reliable grammar.

[3] Notice the use of $\longrightarrow$ in DCG rules and $\leftarrow$ in logical formulae.

[4] The feature name $sing3$ abbreviates *third person singular*, and $(in)trans$ abbreviates $(in)transitive$.

$$B_2 = \boxed{\begin{array}{l} det \longrightarrow [the] \\ noun(sing3(+)) \longrightarrow [boy] \\ noun(sing3(+)) \longrightarrow [mary] \\ noun(sing3(-)) \longrightarrow [you] \\ verb(sing3(+), trans) \longrightarrow [likes] \\ verb(sing3(-), trans) \longrightarrow [like] \\ verb(sing3(+), intrans) \longrightarrow [sleeps] \\ verb(sing3(-), intrans) \longrightarrow [sleep] \end{array}}$$

$$H_2 = \boxed{\begin{array}{l} s \longrightarrow det, np(\_), verb(\_, \_) \\ s \longrightarrow det, np(\_), verb(\_, \_), np(\_) \\ np(F) \longrightarrow noun(F) \end{array}}$$

$$E_2^- = \boxed{\begin{array}{l} \leftarrow np([boy, the], []) \\ \leftarrow s([the, the, boy, sleeps]) \end{array}}$$

$$E_2^+ = \boxed{\begin{array}{l} np([the, boy], []) \leftarrow \\ np([mary], []) \leftarrow \end{array}}$$

$$H_{2mod} = \boxed{\begin{array}{l} s \longrightarrow np(\_), verb(\_, \_) \\ s \longrightarrow np(\_), verb(\_, \_), np(\_) \\ np(F) \longrightarrow noun(F) \\ np(F) \longrightarrow det, noun(F) \end{array}}$$

$$B_3 = B_2$$
$$H_3 = H_{2mod}$$

$$E_3^- = \boxed{\begin{array}{l} \leftarrow s([you, likes, mary]) \\ \leftarrow s([mary, like, the, man])) \\ \leftarrow s([you, sleeps])) \\ \leftarrow s([mary, sleep])) \\ \leftarrow s([mary, sleeps, you])) \\ \leftarrow s([mary, likes])) \end{array}}$$

$$E_3^+ = \boxed{\begin{array}{l} s([mary, likes, you], []) \leftarrow \\ s([you, sleep], []) \leftarrow \end{array}}$$

$$H_{3mod} = \boxed{\begin{array}{l} np(F) \longrightarrow noun(F) \\ np(F) \longrightarrow det, noun(F) \\ s \longrightarrow noun(F1), verb(F1, intrans) \\ s \longrightarrow noun(F1), verb(F1, trans), noun(\_) \end{array}}$$

Figure 2: Example of theory revision

$$B_4 = \boxed{\begin{array}{l} det \longrightarrow [the] \\ noun(sing3(+)) \longrightarrow [boy] \\ noun(sing3(+)) \longrightarrow [mary] \\ noun(sing3(-)) \longrightarrow [you] \\ verb(sing3(+), trans) \longrightarrow [likes] \\ verb(sing3(-), trans) \longrightarrow [like] \\ verb(sing3(+), intrans) \longrightarrow [sleeps] \\ verb(sing3(-), intrans) \longrightarrow [sleep] \\ \\ np(F) \longrightarrow noun(F) \\ np(F) \longrightarrow det, noun(F) \\ s \longrightarrow noun(F1), verb(F1, intrans) \\ s \longrightarrow noun(F1), verb(F1, trans), noun(\_) \end{array}}$$

$$E_4^+ = \boxed{s([the, girl, wants, the, boy], []) \leftarrow}$$

$$H_4 = \boxed{\begin{array}{l} noun(sing3(+)) \longrightarrow [girl] \\ verb(sing3(+), trans) \longrightarrow [wants] \end{array}}$$

Figure 3: Example of abduction

2.1.2 Nonmonotonic setting: knowledge discovery

The less common nonmonotonic[5] setting is used (f.i.) in the system CLAUDIEN [5, 6]. The aim here is not to discriminate between different classes, but to discover properties that are valid with respect to the knowledge base as a whole.

Definition 3 *(nonmonotonic explanation) Given knowledge base KB and formal language L, target hypothesis H is a maximal subset*[6] *of L such that all clauses $h \in H$ are true in the minimal model of $B \cup E$. For instance Figure 4.*

In a natural language processing context discovered properties might offer additional insight into the linguistic knowledge base, and can as such be added as a posteriori specifications. One might also add rules H_5 to the knowledge base itself, as they put additional constraints on correct sentences:

- $H1_5$: *if you have two consecutive words in a sentence, and the second one is a verb, then the first one is always a noun that agrees with that verb.*
- $H2_5$: *if you have a sentence with a transitive verb, then this sentence always contains two nouns*

[5]The name *nonmonotonic* for this setting was introduced by Helft [7], and relates to the fact that the closed world assumption is used. Thus, the addition of new facts to the knowledge base might falsify previously inferred rules.

[6]Sometimes, see [5, 7], one also requires minimality, which means that the hypothesis should not contain redundant clauses.

$$KB_5 = \boxed{\begin{array}{l}
det \longrightarrow [the] \\
noun(sing3(+)) \longrightarrow [boy] \\
noun(sing3(+)) \longrightarrow [girl] \\
noun(sing3(+)) \longrightarrow [mary] \\
noun(sing3(-)) \longrightarrow [you] \\
verb(sing3(+), trans) \longrightarrow [likes] \\
verb(sing3(-), trans) \longrightarrow [like] \\
verb(sing3(+), trans) \longrightarrow [wants] \\
verb(sing3(+), intrans) \longrightarrow [sleeps] \\
verb(sing3(-), intrans) \longrightarrow [sleep] \\
\\
np(F) \longrightarrow noun(F) \\
np(F) \longrightarrow det, noun(F) \\
s \longrightarrow noun(F1), verb(F1, intrans) \\
s \longrightarrow noun(F1), verb(F1, trans), noun(_) \\
\\
nth_word(N, S, W) \leftarrow \ldots \%W \text{ is the } N^{th} \text{ word in list } S \\
member(E, L) \leftarrow \ldots \%E \text{ occurs in list } L \\
contains(L, N, C) \leftarrow \ldots \%\text{list } L \text{ contains } N \text{ words of category } C
\end{array}}$$

$H_5 = H1_5 \wedge H2_5 \wedge H3_5 \wedge \ldots$

$$H1_5 = \boxed{\begin{array}{ll}
noun(F1, Word1) \leftarrow & s(Sentence, [])\wedge \\
& nth_word(N, Sentence, Word1)\wedge \\
& nth_word((N+1), Sentence, Word2)\wedge \\
& verb(F1, _, Word2)
\end{array}}$$

$$H2_5 = \boxed{\begin{array}{ll}
contains(Sentence, 2, noun) \leftarrow & s(Sentence, [])\wedge \\
& member(Sentence, Word)\wedge \\
& verb(_, trans, Word)
\end{array}}$$

$$H3_5 = \boxed{\begin{array}{ll}
contains(Sentence, 1, noun)\vee & \\
contains(Sentence, 2, noun) \leftarrow & s(Sentence, [])\wedge \\
& member(Sentence, Word)\wedge \\
& verb(_, _, Word)
\end{array}}$$

Figure 4: Example of knowledge discovery

- $H3_5$: *if you have a sentence with a verb, then this sentence contains either one or two nouns*

2.2 A proof theory: how to search

A general strategy to search for a hypothesis that meets the normal or nonmonotonic acceptance criteria defined above would be to generate and test candidate solutions randomly. A standard technique in artificial intelligence to overcome practical problems associated with this naive method is to structure the space of possible solutions. Search algorithms can then exploit the structure to prune away unpromising parts, and explore the remaining areas in a systematic way. To navigate effectively, inductive logic programming algorithms typically rely on the graph structure imposed by the dual notions of generalization and specialization [9, 17]. Depending on whether the learner starts from the most specific or most general clause it will use a generalization or specialization operator for making the smallest possible next move through the graph. Such an operator can for instance be based on θ-subsumption as introduced by Plotkin:

Definition 4 *(θ-subsumption [14, 15]) A clause* c1 *θ-subsumes a clause* c2 *if and only if there exists a substitution θ such that* $c1\theta \subseteq c2$. c1 *is a generalization of* c2 *(and* c2 *is a specialization of* c1*) under θ-subsumption.*

For example, given the relation between DCG's and clauses as explained above,

$$s \longrightarrow noun(A), verb(B, C)$$

θ-subsumes

$$s \longrightarrow noun(X), verb(X, trans), noun(Y)$$

with $\theta = \{A = X, B = X, C = trans\}$. Figure 5 shows a larger part of the DCG search space as it is structured by θ-subsumption. The descendants of a node represent its *specializations*.

Given a structured search space, most inductive logic programming systems can be seen as instantiations of the following generic procedure taken from Muggleton and De Raedt [12].

QH := `Initialize`
repeat
    `Delete` H from QH
    `Choose` the inference rules $r_1, ..., r_k \in$ `R` to be applied to H
    Apply the rules $r_1, ..., r_k$ to H to yield $H_1, H_2, ..., H_n$
    Add $H_1, ..., H_n$ to QH
    `Prune` QH
until `Stop-criterion`(QH) satisfied

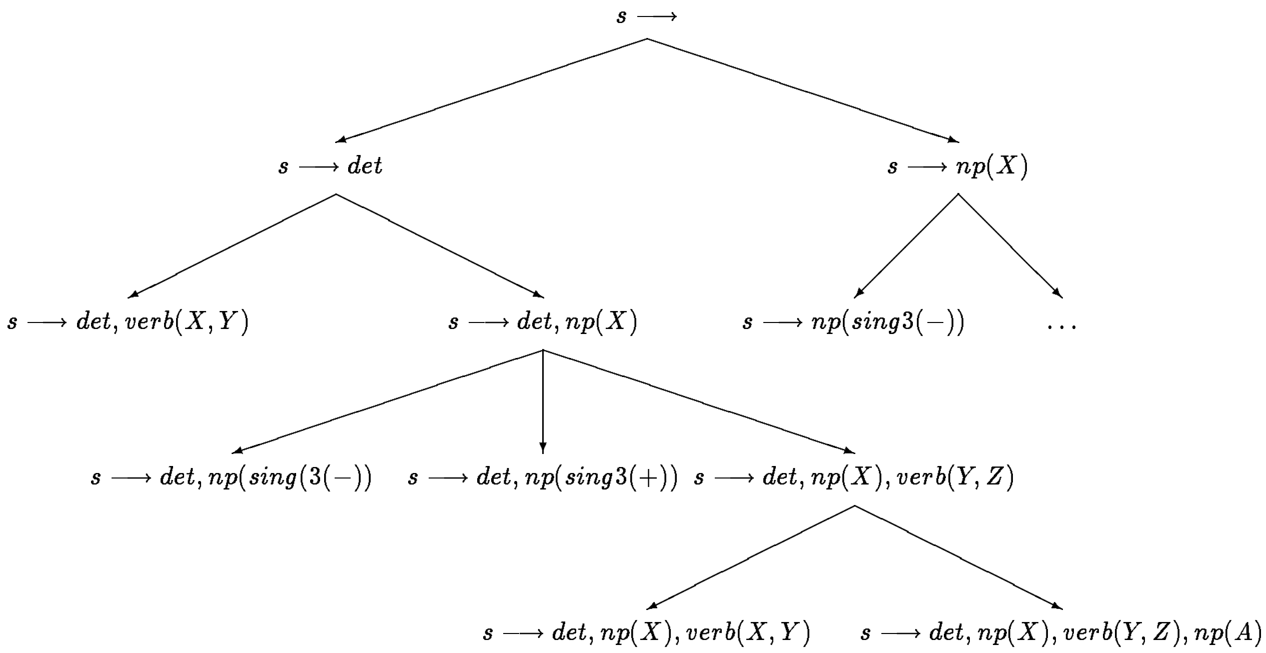

Figure 5: A DCG search space structured under θ-subsumption

The algorithm above iteratively processes a queue QH of hypotheses H, with each H a conjunction of first-order clauses. Iteration continues until QH satisfies some stopping-criterion. At each step, a hypothesis H is selected from QH, the result of inferencing on H is put back, and unpromising items are pruned away. Inference rules typically generate either specializations or generalizations of H. The hypotheses in QH can thus be seen as nodes in a search tree, the strategy for searching this tree being largely determined by the instantiation of the generic parameter `Delete`.

2.3 Declarative bias: what to search

To make inductive logic programming useful for real world practical purposes, the search space should not only be structured, but also constrained. A priori knowledge about restrictions on the form (syntactic bias) or the meaning (semantic bias) of clauses allowed in a solution hypothesis should be added as declarative bias to the learning system. Several frameworks exist for specifying declarative bias (see [11] for an overview). We briefly introduce an extension of the formalism of [1] as it is used in the nonmonotonic system CLAUDIEN.

In CLAUDIEN, a language L for well-formed hypotheses is specified as a set of clause-models of the form:

$$template \leftarrow template$$

where the *template* to the left (right) defines a language for the heads (bodies) of clauses in L. For instance, again taking into account the straightforward transformation from clauses to DCG-formulae, with clausemodel

$$s \longrightarrow +1\{det, np(X)\}$$

the search space is constrained to the following three clauses:

$$L = \boxed{\begin{array}{l} s \longrightarrow det \\ s \longrightarrow noun(X) \\ s \longrightarrow det, noun(X) \end{array}}$$

The form and meaning of *template* are defined as follows:

- syntax of *template*

$$template \Rightarrow \boxed{\begin{array}{l} literal \\ < template, \ldots, template > \\ [template, \ldots, template] \\ \{template, \ldots, template\} \\ +1\{template, \ldots, template\} \end{array}}$$

- semantics of *template*

template	added to clause
literal	*literal*
$< t_1, \ldots, t_n >$	t_1 and ... and t_n
$[t_1, \ldots, t_n]$	t_1 xor ... xor t_n
$\{t_1, \ldots, t_n\}$	subset of $t_1, \ldots, t_n$
$+1\{t_1, \ldots, t_n\}$	subset $\neq \emptyset$ of $t_1, \ldots, t_n$

An example of the use of *template* in clausemodels is included in the discussion of a small-scale experiment with CLAUDIEN in the following section.

3 A small-scale discovery experiment

In the experiment described here, CLAUDIEN is used for the analysis of diminutive forms in Dutch[7]. Most Dutch words have a diminutive form, which is constructed by adding one of the suffixes '-je', '-tje', '-etje', '-pje' or '-kje' to it; usually, only one suffix is correct for a given word. The aim of this experiment is to let CLAUDIEN learn the relationship between the sounds occurring in a word, and the suffix it gets.

[7]A more elaborate discussion of the application of machine learning techniques to Dutch diminutive formation can be found in [2].

3.1 Data and background knowledge

The data used for the experiment is a set of 3897 diminutives. For every diminutive, the last three syllables of the stem are given. Each syllable is represented by a 4-tuple consisting of a stress feature, onset cluster, vocal and coda cluster. A phonological representation is used for phonemes. Furthermore, the appropriate suffix is given, as well as the written representation of the diminutive and the logarithm of its frequency in the corpus. All this information is included in one fact for each diminutive.

Some examples of these data are:

$dim(+, r, ie, =, -, j, aa, =, -, b, ee, lt, j, \text{`}Mariabeeldje\text{'}, 0) \leftarrow$

$dim(=, =, =, =, -, =, aa, =, +, b, ee, l, t, abeeltje, 0) \leftarrow$

The '='-symbol signifies the absence of the corresponding value. The suffix is represented by its first letter. To this set of data, some background knowledge is added. For all vocals, the features *back* (+,−) and *round* (+,−) are given, as well as their height (*low*, *middle* or *high*). Furthermore, vocals are classified into long vowels, short vowels, diphthongs, schwa, and loan vowels.

For consonants, the following features and classifications are represented:

- classification into obstruents, half-vocals, liquids and nasals
- classification into dental, labial and velar
- the *tense* feature
- classification into plosives and fricatives

Since consonant clusters are used, derived predicates are introduced that indicate the properties of the last consonant in a cluster. For instance, a predicate *endobstr* is defined, which is true for some cluster if the last consonant of the cluster is an obstruent.

The background knowledge that is added to the data, reflects the way the user thinks about diminutive formation; i.e. features that may be important are included, while features that are considered irrelevant are left out. There are no predicates to represent the properties of the first consonant in a cluster, because this is deemed irrelevant. The background thus offers a lot of flexibility: in combination with the language model, it allows the system to focus on relevant features.

3.2 The declarative bias

For this experiment, two clausemodels have been used; the language defined by the two of them is the union of the languages defined by each one of them. The exact specification of these models is given in the appendix; only an informal description will be given here.

The first model indicates that clauses that predict some specific suffix are interesting. These clauses are always of the form "if the word has this sound pattern, then the suffix is …". This models defines a concept learning task: the words will be classified into several classes, each class corresponding to one suffix.

In this model, only the coda cluster of the last syllable is considered important; there will be no clauses where properties of other consonant clusters are used as conditions. All vocals of the last three syllables, however, can be taken into account, as well as their stress features.

The second model is used to look for the relationship between the properties of the last consonant of the last coda cluster, and the properties of the first consonant of the suffix. The clauses produced by this model are all of the form "if a diminutive with these features is used, the immediately preceding consonant must have the following features: ...". As opposed to the previous model, this model does not define a concept learning task, but a knowledge discovery task. The clauses specified by it will relate properties of suffixes (not the suffixes themselves) to properties of the immediately preceding consonant, if there is one.

As an extra restriction on rules, a maximal body complexity of three literals was imposed. This implies that, since the *dim* literal is always present (it contains the data that will be used), only two extra literals can be added.

3.3 Results

With the above models, about 300 clauses have been found. They are easily recognisable as having been produced by the first, respectively the second model.

3.3.1 Predictive rules

The first model gives rise to the discovery of rules such as

$$\begin{array}{l} Dimin = t \leftarrow \\ \quad dim(S3, O3, K3, C3, S2, O2, K2, C2, S1, O1, K1, C1, Dimin, D, F), \\ \quad endhalfv(C1) \\ Dimin = j \leftarrow \\ \quad dim(S3, O3, K3, C3, S2, O2, K2, C2, S1, O1, K1, C1, Dimin, D, F), \\ \quad C1 = k \\ Dimin = t \leftarrow \\ \quad dim(S3, O3, K3, C3, S2, O2, K2, C2, S1, O1, K1, C1, Dimin, D, F), \\ \quad long(K1), endliq(C1) \end{array}$$

The rules show that all words ending on a halfvocal get the suffix '-tje' (thereby combining words on 'j' and words on 'w' in one rule), that all words ending on 'k' get suffix '-je', and that of the words ending on a liquid, those where the last vocal is long get '-tje'.

CLAUDIEN finds many rules of this kind. If all rules predicting a '-tje' suffix are put together, this ruleset will adequately define the class of words getting '-tje'. However, the ruleset will not be minimal. As CLAUDIEN is, in principle, a knowledge discovery program, *every* valid clause will be returned. Typical concept learning programs will look for some subset of all valid clauses, such that this subset completely defines the concept;

once such a set is found, they will stop. To find such a minimal set with CLAUDIEN, one has to apply some filter to the ruleset.

The predictive power of a ruleset can be computed by extracting clauses from a subset of the available data (the training set), and then applying these clauses to predict suffixes for the remaining data (the test set). A test where 90 % of the diminutives has been used as a training set, and the remaining 10 % as a test set, has shown that in 2 % of the cases a wrong suffix is predicted (possibly together with the correct suffix), while 5 % of the test cases was not covered by any rule.

3.3.2 Other rules

The second model gives rise to a.o. the following clause:

$$
\begin{aligned}
&\mathit{endobstr}(C1) \leftarrow \\
&\quad \mathit{dim}(S3, O3, K3, C3, S2, O2, K2, C2, S1, O1, K1, C1, \mathit{Dimin}, D, F), \\
&\quad \mathit{halfvocal}(\mathit{Dimin})
\end{aligned}
$$

This rule states that the '-je' suffix can only follow an obstruent. Other rules have been derived which show that suffixes starting with a labial or velar consonant ('-pje' respectively '-kje') can only follow the nasal consonant of the same class (i.e. 'm' and 'ng').

4 Conclusions

At least at first sight, general inductive logic programming methods seem to be directly applicable to support relevant natural language research tasks situated in the field of computational logic. The normal setting might contribute to the construction of grammars and lexicons, and to their adaptation in the presence of newly arriving evidence. When applied in the nonmonotonic setting, inductive logic programming might lead to a better understanding of given natural language processing systems and to the discovery of new theories and models.

Acknowledgements

This work is part of the Esprit Basic Research project no. 6020 on Inductive Logic Programming. Luc Dehaspe is paid by the Esprit Basic Research Action ILP (project 6020), and co-financed by the Flemish Government through contract nr.93/014. Hendrik Blockeel is financed by the Flemish Institute for the advancement of scientific-technological research in industry (IWT). Luc De Raedt is supported by the Belgian National Fund for Scientific Research.

The authors would like to thank Walter Daelemans for making the data on Dutch diminutives available, and for interesting discussions on the application of machine learning techniques to linguistics.

Appendix: declarative bias for the Dutch diminutive

In CLAUDIEN, the bias is specified using clausemodels, as described in Section 2.3. In addition to the constructs described there, some shorthands can be used: most constructs to group predicates are also allowed at the argument level. For instance,

$$endtense(C1, [+, -])$$

can be used as a shorthand for

$$[endtense(C1, +), endtense(C1, -)].$$

Also, (second order) predicate variables can be used; in a clause, such a variable will be substituted by one of its possible instantiations.

The first model

This model produces predictive clauses, where the head contains the correct suffix and the body contains possibly relevant features. It is specified as follows:

$$
\begin{array}{l}
= (Dimin, [t, e, j, k, p]) \leftarrow \\
\quad dim(S3, O3, K3, C3, S2, O2, K2, C2, S1, O1, K1, C1, Dimin, D, F), \\
\quad \{= (+1\{S3, S2, S1\}, [=, +, -]), \\
\quad\; Type(+1\{K3, K2, K1\}), \\
\quad\; Height(+1\{K3, K2, K1\}), \\
\quad\; back(+1\{K3, K2, K1\}, [+, -]), \\
\quad\; round(+1\{K3, K2, K1\}, [+, -]), \\
\quad\; [= (C1, [=, g, gd, ng, ngk, nk, ngs, ngz, sj, b, bd, d, dz, f, ft, gk, gkd, gz, gzd, j, k, \\
\qquad\quad ks, kst, kt, l, lG, lb, ld, ldz, lf, lfs, lg, lk, lks, lm, lp, ls, lt, lx, lz, m, md, mf, \\
\qquad\quad mp, mt, mts, n, nS, nd, ndz, ns, nst, nt, nts, nz, p, ps, r, rd, rdz, rf, rg, \\
\qquad\quad rk, rkt, rm, rn, rp, rs, rst, rt, rv, rx, rz, s, st, t, ts, v, w, x, xd, xs, xt, z, zd]), \\
\quad\;\; +1\{[endobstr(C1), endnas(C1), endhalfv(C1), endliq(C1)], \\
\qquad endtense(C1, [+, -]), \\
\qquad [enddental(C1), endlabial(C1), endvelar(C1)], \\
\qquad [endfricative(C1), endplosive(C1)]\}]\}
\end{array}
$$

where $Type \in \{long,\ short,\ diphthong,\ schwa,\ loan\}$ and $Height \in \{high,\ middle,\ low\}$.

Note that the clusters $C2, C3, O1, O2, O3$ are never used; only the $C1$ cluster is checked for certain properties. Of the vocals $K1, K2, K3$ and stress features $S1, S2, S3$, all are used.

The second model

This model relates properties of the suffix to properties of the immediately preceding consonant, and is specified as follows:

$$
\begin{array}{l}
\{[endobstr(C1), endnas(C1), endhalfv(C1), endliq(C1)], \\
\quad endtense(C1, [+, -]), \\
\quad [enddental(C1), endlabial(C1), endvelar(C1)], \\
\quad [endfricative(C1), endplosive(C1)]\} \\
\leftarrow \\
dim(S3, O3, K3, C3, S2, O2, K2, C2, S1, O1, K1, C1, Dimin, D, F), \\
\{[obstruent(Dimin), nasal(Dimin), halfvocal(Dimin), liquid(Dimin)], \\
\quad tense(Dimin, [+, -]), \\
\quad [dental(Dimin), labial(Dimin), velar(Dimin)], \\
\quad [fricative(Dimin), plosive(Dimin)]\}
\end{array}
$$

As every suffix is represented by its first consonant, the predicates for consonant features are directly applicable to the suffix symbols, without any conversion.

References

[1] H. Adé, L. De Raedt, and M. Bruynooghe. Declarative Bias for Specific-To-General ILP Systems. *Machine Learning*, 1994. to appear.

[2] W. Daelemans and P. Berck. Linguistics as data mining: the case of dutch diminutive formation. *unpublished paper*, 1995.

[3] V. Dahl. Natural language processing and logic programming. *Journal of Logic Programming*, 19,20:681–714, 1994.

[4] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.

[5] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.

[6] L. Dehaspe, W. Van Laer, and L. De Raedt. Applications of a logical discovery engine. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 291–304. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[7] N. Helft. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–156. Morgan Kaufmann, 1989.

[8] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[9] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[10] S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, 1992.

[11] S. Muggleton. Predicate invention and utility. *Journal for Experimental and Theoretical Artificial Intelligence*, 1994. To appear.

[12] S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[13] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[14] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[15] G. Plotkin. *Automatic Methods of Inductive Inference.* PhD thesis, Edinburgh University, 1971.

[16] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[17] E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, 1983.

[18] A. Srinivasan, S.H. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: Ilp experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 217–232. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.