An Introduction to Inductive Logic Programming and Learning Language in Logic

Sašo Džeroski[1], James Cussens[2], and Suresh Manandhar[2]

[1] Jožef Stefan Institute,
Jamova 39, 1000 Ljubljana, Slovenia
saso.dzeroski@ijs.si

[2] Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK
jc@cs.york.ac.uk, suresh@cs.york.ac.uk

Abstract. This chapter introduces Inductive Logic Programming (ILP) and Learning Language in Logic (LLL). No previous knowledge of logic programming, ILP or LLL is assumed. Elementary topics are covered and more advanced topics are discussed. For example, in the ILP section we discuss subsumption, inverse resolution, least general generalisation, relative least general generalisation, inverse entailment, saturation, refinement and abduction. We conclude with an overview of this volume and pointers to future work.

1 Introduction

Learning Language in Logic (LLL) is an emerging research area lying at the intersection of computational logic, machine learning and natural language processing (Fig 1). To see what is to be gained by focussing effort at this intersection begin by considering the role of logic in natural language processing (NLP). The flexibility and expressivity of logic-based representations have led to much effort being invested in developing logic-based resources for NLP. However, manually developed logic language descriptions have proved to be brittle, expensive to build and maintain, and still do not achieve high coverage on unrestricted text. As a result, rule-based natural language processing is currently viewed as too costly and fragile for commercial applications. Natural language learning based on statistical approaches, such as the use of n-gram models for part-of-speech tagging or lexicalised stochastic grammars for robust parsing, partly alleviate these problems. However, they tend to result in linguistically impoverished descriptions that are difficult to edit, extend and interpret.

Work in LLL attempts to overcome both the limitations of existing statistical approaches and the brittleness of hand-crafted rule-based methods. LLL is, roughly, the application of Inductive Logic Programming (See Section 3) to NLP. ILP works within a logical framework but uses data and logically encoded background knowledge to learn or revise logical representations. As Muggleton notes in the foreword to this volume: “From the NLP point of view the promise of ILP is that it will be able to steer a mid-course between the two alternatives of

J. Cussens and S. Džeroski (Eds.): LLL’99, LNAI 1925, pp. 3–35, 2000.
© Springer-Verlag Berlin Heidelberg 2000

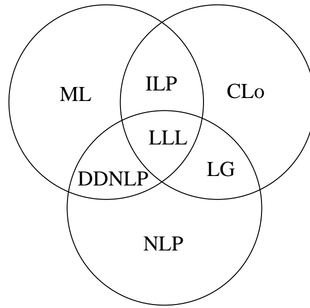

Fig. 1. Situating LLL (CLo = computational logic, ML = machine learning, DDNLP = data-driven NLP, LLL = learning language in logic, LG = logic grammars, NLP = natural language processing, ILP = inductive logic programming.)

large scale, but shallow levels of analysis, and small scale, but deep and precise analyses, and produce a better balance between breadth of coverage and depth of analysis." From the ILP point of view, NLP is an ideal application area. The existence within NLP problems of hierarchically defined, structured data with large amounts of relevant, often logically defined, background knowledge provides a perfect testbed for stretching ILP technology in a way that would also be beneficial in other application areas.

The aim of this introductory chapter is to provide the background required to make ILP and LLL accessible to the widest possible audience. The sections on logic programming and ILP (Sections 2–7) give a concise account of the main ideas in the hope that, for example, computational linguists unfamiliar with either or both of these topics can appreciate what LLL has to offer. Thompson (this volume) has a complementary goal: to give an overview, accessible to non-linguists, of the main problems of NLP and how they are being addressed. In both cases, no previous knowledge is assumed so that quite elementary topics are covered.

Section 8 summarises the contributions contained in this volume. We conclude with Section 9 where we analyse which NLP problems are most suited to a LLL approach and suggest future directions for LLL.

2 Introduction to Logic Programming

In this section, the basic concepts of logic programming are introduced. These include the language (syntax) of logic programs, as well as basic notions from model and proof theory. The syntax defines what are legal sentences/statements in the language of logic programs. Model theory is concerned with assigning meaning (truth values) to such statements. Proof theory focuses on (deductive) reasoning with such statements. For a thorough treatment of logic programming we refer to the standard textbook of Lloyd (1987). The overview below is mostly based on the comprehensive and easily readable text by Hogger (1990).

2.1 The Language

A first-order alphabet consists of variables, predicate symbols and function symbols (which include constants). A variable is a term, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X), h)$ is a term when f, g and h are function symbols and X is a variable—strings starting with lower-case letters denote predicate and function symbols, while strings starting with upper-case letters denote variables. A constant is a function symbol of arity 0 (i.e. followed by a bracketed 0-tuple of terms, which is usually left implicit). A predicate symbol immediately followed by a bracketed n-tuple of terms is called an atomic formula or atom. For example, if $mother$ and $father$ are predicate symbols then $mother(maja, filip)$ and $father(X, Y)$ are atoms.

A well-formed formula is either an atomic formula or takes one of the following forms: (F), $\overline{F}$, $F \vee G$, $F \wedge G$, $F \leftarrow G$, $F \leftrightarrow G$, $\forall X : F$ and $\exists X : F$, where F and G are well-formed formulae and X is a variable. $\overline{F}$ denotes the negation of F, $\vee$ denotes logical disjunction (or), and $\wedge$ logical conjunction (and). $F \leftarrow G$ stands for implication (F if G, $F \vee \overline{G}$) and $F \leftrightarrow G$ stands for equivalence (F if and only if G). $\forall$ and $\exists$ are the universal (for all X F holds) and existential quantifier (there exists an X such that F holds). In the formulae $\forall X : F$ and $\exists X : F$, all occurrences of X are said to be bound. A sentence or a closed formula is a well-formed formula in which every occurrence of every variable symbol is bound. For example, $\forall Y \exists X father(X, Y)$ is a sentence, while $father(X, andy)$ is not.

Clausal form is a normal form for first-order sentences. A clause is a disjunction of literals—a positive literal is an atom, a negative literal the negation of an atom—preceded by a prefix of universal quantifiers, one for each variable appearing in the disjunction. In other words, a clause is a formula of the form $\forall X_1 \forall X_2 ... \forall X_s (L_1 \vee L_2 \vee ... L_m)$, where each L_i is a literal and $X_1, X_2, ..., X_s$ are all the variables occurring in $L_1 \vee L_2 \vee ... L_m$. Usually the prefix of variables is left implicit so that $\forall X_1 \forall X_2 ... \forall X_s (L_1 \vee L_2 \vee ... L_m)$ is written as $L_1 \vee L_2 \vee ... L_m$.

A clause can also be represented as a finite set (possibly empty) of literals. The set $\{A_1, A_2, ..., A_h, \overline{B_1}, \overline{B_2}, ..., \overline{B_b}\}$, where A_i and B_i are atoms, stands for the clause $(A_1 \vee ... \vee A_h \vee \overline{B_1} \vee ... \vee \overline{B_b})$, which is equivalently represented as $A_1 \vee ... \vee A_h \leftarrow B_1 \wedge ... \wedge B_b$. Most commonly, this same clause is written as $A_1, ..., A_h \leftarrow B_1, ..., B_b$, where $A_1, ..., A_h$ is called the head and $B_1, ..., B_b$ the body of the clause. A finite set of clauses is called a clausal theory and represents the conjunction of its clauses.

A clause is a Horn clause if it contains at most one positive literal; it is a definite clause if it contains exactly one positive literal. A set of definite clauses is called a definite logic program. A fact is a definite clause with an empty body, e.g., $parent(mother(X), X) \leftarrow$, also written simply as $parent(mother(X), X)$. A goal (also called a query) is a Horn clause with no positive literals, such as $\leftarrow parent(mother(X), X)$.

A program clause is a clause of the form $A \leftarrow L_1, ..., L_m$ where A is an atom, and each of $L_1, ..., L_m$ is a positive or negative literal. A negative literal in the

body of a program clause is written in the form *not* B, where B is an atom. A normal program (or logic program) is a set of program clauses. A predicate definition is a set of program clauses with the same predicate symbol (and arity) in their heads.

Let us now illustrate the above definitions with some examples. The clause

$$daughter(X, Y) \leftarrow female(X), mother(Y, X).$$

is a definite program clause, while the clause

$$daughter(X, Y) \leftarrow not\ male(X), father(Y, X).$$

is a normal program clause. Together, the two clauses constitute a predicate definition of the predicate $daughter/2$. This predicate definition is also a normal logic program. The first clause is an abbreviated representation of the formula

$$\forall X \forall Y : daughter(X, Y) \vee \overline{female(X)} \vee \overline{mother(Y, X)}$$

and can also be written in set notation as

$$\{daughter(X, Y), \overline{female(X)}, \overline{mother(Y, X)}\}$$

The set of variables in a term, atom or clause F, is denoted by $vars(F)$. A substitution $\theta = \{V_1/t_1, ..., V_n/t_n\}$ is an assignment of terms t_i to variables V_i. Applying a substitution θ to a term, atom, or clause F yields the instantiated term, atom, or clause $F\theta$ where all occurrences of the variables V_i are simultaneously replaced by the term t_i. A term, atom or clause F is called ground when there is no variable occurring in F, i.e., $vars(F) = \emptyset$. The fact $daughter(maty, ann)$ is thus ground.

A clause or clausal theory is called function free if it contains only variables as terms, i.e., contains no function symbols (this also means no constants). The clause $daughter(X, Y) \leftarrow female(X), mother(Y, X)$ is function free and the clause $even(s(s(X)) \leftarrow even(X)$ is not. A Datalog clause (program) is a definite clause (program) that contains no function symbols of non-zero arity. This means that only variables and constants can be used as predicate arguments. The size of a term, atom, clause, or a clausal theory T is the number of symbols that appear in T, i.e., the number of all occurrences in T of predicate symbols, function symbols and variables.

2.2 Model Theory

Model theory is concerned with attributing meaning (truth value) to sentences in a first-order language. Informally, the sentence is mapped to some statement about a chosen domain through a process known as interpretation. An interpretation is determined by the set of ground facts (ground atomic formulae) to which it assigns the value true. Sentences involving variables and quantifiers are interpreted by using the truth values of the ground atomic formulae and a fixed

set of rules for interpreting logical operations and quantifiers, such as "$\overline{F}$ is true if and only if F is false".

An interpretation which gives the value true to a sentence is said to satisfy the sentence; such an interpretation is called a model for the sentence. An interpretation which does not satisfy a sentence is called a counter-model for that sentence. By extension, we also have the notion of a model (counter-model) for a set of sentences (e.g., for a clausal theory): an interpretation is a model for the set if and only if it is a model for each of the set's members. A sentence (set of sentences) is satisfiable if it has at least one model; otherwise it is unsatisfiable.

A sentence F logically implies a sentence G if and only if every model for F is also a model for G. We denote this by $F \models G$. Alternatively, we say that G is a logical (or semantic) consequence of F. By extension, we have the notion of logical implication between sets of sentences.

A Herbrand interpretation over a first-order alphabet is a set of ground facts constructed with the predicate symbols in the alphabet and the ground terms from the corresponding Herbrand domain of function symbols; this is the set of ground atoms deemed to be true by the interpretation. A Herbrand interpretation I is a model for a clause c if and only if for all substitutions θ such that $c\theta$ is ground $body(c)\theta \subset I$ implies $head(c)\theta \cap I \neq \emptyset$. In that case, we say c is true in I. A Herbrand interpretation I is a model for a clausal theory T if and only if it is a model for all clauses in T. We say that I is a Herbrand model of c, respectively T.

Roughly speaking, the truth of a clause c in a (finite) interpretation I can be determined by running the goal (query) $body(c), not\ head(c)$ on a database containing I, using a theorem prover such as PROLOG. If the query succeeds, the clause is false in I; if it fails, the clause is true. Analogously, one can determine the truth of a clause c in the minimal (least) Herbrand model of a theory T by running the goal $body(c), not\ head(c)$ on a database containing T.

To illustrate the above notions, consider the Herbrand interpretation

$$i = \{parent(saso, filip), parent(maja, filip), son(filip, saso), son(filip, maja)\}$$

The clause $c = parent(X, Y) \leftarrow son(Y, X)$ is true in i, i.e., i is a model of c. On the other hand, i is not a model of the clause $parent(X, X) \leftarrow$ (which means that everybody is their own parent).

2.3 Proof Theory

Proof theory focuses on (deductive) reasoning with logic programs. Whereas model theory considers the assignment of meaning to sentences, proof theory considers the generation of sentences (conclusions) from other sentences (premises). More specifically, proof theory considers the derivability of sentences in the context of some set of inference rules, i.e., rules for sentence derivation. An inference rule has the following schematic form: "from a set of sentences of that kind, derive a sentence of this kind". Formally, an inference system consists of an initial set S of sentences (axioms) and a set R of inference rules.

Using the inference rules, we can derive new sentences from S and/or other derived sentences. The fact that sentence s can be derived from S is denoted $S \vdash s$. A proof is a sequence $s_1, s_2, ..., s_n$, such that each s_i is either in S or derivable using R from S and $s_1, ..., s_{i-1}$. Such a proof is also called a derivation or deduction. Note that the above notions are of entirely syntactic nature. They are directly relevant to the computational aspects of automated deductive inference.

The set of inference rules R defines the derivability relation $\vdash$. A set of inference rules is sound if the corresponding derivability relation is a subset of the logical implication relation, i.e., for all S and s, if $S \vdash s$ then $S \models s$. It is complete if the other direction of the implication holds, i.e., for all S and s, if $S \models s$ then $S \vdash s$. The properties of soundness and completeness establish a relation between the notions of syntactic ($\vdash$) and semantic ($\models$) entailment in logic programming and first-order logic. When the set of inference rules is both sound and complete, the two notions coincide.

Resolution comprises a single inference rule applicable to clausal-form logic. From any two clauses having an appropriate form, resolution derives a new clause as their consequence. For example, the clauses

$$daughter(X, Y) \leftarrow female(X), parent(Y, X)$$
$$female(sonja) \leftarrow$$

resolve into the clause

$$daughter(sonja, Y) \leftarrow parent(Y, sonja)$$

Resolution is sound: every resolvent is implied by its parents. It is also refutation complete: the empty clause is derivable by resolution from any set S of Horn clauses if S is unsatisfiable. Resolution is discussed further in Section 3.

2.4 Logic Programming for NLP

Even when restricted to clausal form, the language of first-order logic is sufficiently flexible to cleanly represent a wide range of linguistic information. For example, grammar rules are easy to represent as clauses. Consider the grammar rule

$$\text{VP} \rightarrow \text{VP MOD}$$

where VP stands for verb phrase and MOD stands for modifier. If a grammar can establish that the word *ran* constitutes a verb phrase and that the word *quickly* constitutes a modifier, then this rule allows us to infer that *ran quickly* is a verb phrase.

We can represent this rule by the definite clause

$$vp(Start, End) \leftarrow vp(Start, Middle), mod(Middle, End)$$

where the variables stand for vertices in between words. For example if, from the sentence $_0$ *She* $_1$ *ran* $_2$ *quickly* $_3$, we derive $vp(1, 2), mod(2, 3)$, then the given

clause allows us to infer $vp(1,3)$. $vp(1,2)$, $mod(2,3)$ and $vp(1,3)$ are examples of *edges* as used in chart parsing. See Thompson (this volume) for a description of chart parsing.

A clause such as this, where a predicate appears in both the head and the body is called a recursive clause.

Grammars represented as a set of definite clauses are, unsurprisingly, known as definite clause grammars (DCGs). The logic programming language PROLOG has a special DCG syntax so that the PROLOG clause

```
vp(Start,End) :- vp(Start,Middle), mod(Middle,End).
```

(where the `:-` represents a $\leftarrow$) may be more compactly written as

```
vp --> vp, mod.
```

First-order terms are important in representing structured information. The most common sort of term used in logic programming is the list. Lists are recursive terms: a list is either the empty list or of the form $.(Head, Tail)$, where $Tail$ is a list. In PROLOG, $.(Head, Tail)$ can be more intuitively represented as `[Head|Tail]`. Sentences are often represented as lists, so *She ran quickly* would be represented in PROLOG as `['She'|[ran|[quickly|[]]]]`. In fact, as a convenience, PROLOG allows one to write this as `['She',ran,quickly]`. Note that the constant *She* has been quoted since otherwise its initial capital letter would mean that it would be interpreted as a variable.

Structured terms allow great representational flexibility. For example, in the grammar framework used by Cussens and Pulman (this volume), it is more convenient to represent the grammar rule

$$\mathrm{VP} \rightarrow \mathrm{VP\ MOD}$$

as the following non-ground fact, using terms in place of literals

$$cmp_synrule(vp(Start, End), [vp(Start, Middle), mod(Middle, End)]).$$

First-order terms are sufficiently flexible to represent semantic, as well as syntactic, information. Boström (this volume) uses ILP to translate between quasi-logical forms (QLFs)—terms intended to represent the meaning of sentences. QLFs are often very complex; here is one representing the meaning of the sentence *List the prices*

```
[imp, form(_,verb(no,no,no,imp,y),A,
B^[B,[list_Enumerate,A,
term(_,ref(pro,you,_,l([])),_,
C^[personal,C],_,_),
term(_,q(_,bare,plur),_,
D^[fare_Price,D],_,_)]],_)]
```

The underscores (_) are a PROLOG convention used to represent ‘anonymous variables’, each underscore is implicitly replaced by a distinct variable.

3 Introduction to ILP

While logic programming (and in particular its proof theory) is concerned with deductive inference, inductive logic programming is concerned with inductive inference. It generalizes from individual examples/observations in the presence of background knowledge, finding regularities/hypotheses. The most commonly addressed task in ILP is the task of learning logical definitions of relations (Quinlan, 1990), where tuples that belong or do not belong to the target relation are given as examples. ILP then induces a logic program (predicate definition) defining the target relation in terms of other relations that are given as background knowledge.

We assume a set of examples is given, i.e., tuples that belong to the target relation p (positive examples) and tuples that do not belong to p (negative examples). Also given are background relations (or background predicates) q_i that constitute the background knowledge and can be used in the learned definition of p. Finally, a hypothesis language, specifying syntactic restrictions on the definition of p is also given (either explicitly or implicitly). The task is to find a definition of the target relation p that is consistent and complete. Informally, it has to explain all the positive and none of the negative examples.

More formally, we have a set of examples $E = P \cup N$, where P contains positive and N negative examples, and background knowledge B. The task is to find a hypothesis H such that $\forall e \in P : B \wedge H \models e$ (H is complete) and $\forall e \in N : B \wedge H \not\models e$ (H is consistent). This setting was introduced by Muggleton (1991). In an alternative setting proposed by De Raedt and Džeroski (1994), the requirement that $B \wedge H \models e$ is replaced by the requirement that H be true in the minimal Herbrand model of $B \wedge e$: this setting is called learning from interpretations.

In the most general formulation, each e, as well as B and H can be a clausal theory. In practice, each e is most often a ground example and H and B are definite logic programs. Recall that $\models$ denotes logical implication (semantic entailment). Semantic entailment ($\models$) is in practice replaced with syntactic entailment ($\vdash$) / provability, where the resolution inference rule (as implemented in PROLOG) is most often used to prove examples from a hypothesis and the background knowledge.

As an illustration, consider the task of defining the relation $daughter(X, Y)$, which states that person X is a daughter of person Y, in terms of the background knowledge relations $female$ and $parent$. These relations, as well as two more background relations $mother$ and $father$ are given in Table 1. There are two positive and two negative examples of the *target* relation $daughter$.

In the hypothesis language of definite program clauses it is possible to formulate the following definition of the target relation,

$$daughter(X, Y) \leftarrow female(X), parent(Y, X).$$

which is consistent and complete with respect to the background knowledge and the training examples.

Table 1. A simple ILP problem: learning the *daughter* relation.

Training examples		*Background knowledge*	
$daughter(mary, ann).$	$\oplus$	$mother(ann, mary).$	$female(ann).$
$daughter(eve, tom).$	$\oplus$	$mother(ann, tom).$	$female(mary).$
$daughter(tom, ann).$	$\ominus$	$father(tom, eve).$	$female(eve).$
$daughter(eve, ann).$	$\ominus$	$father(tom, ian).$	
		$parent(X, Y) \leftarrow mother(X, Y)$	
		$parent(X, Y) \leftarrow father(X, Y)$	

In general, depending on the background knowledge, the hypothesis language and the complexity of the target concept, the target predicate definition may consist of a set of clauses, such as

$$daughter(X, Y) \leftarrow female(X), mother(Y, X).$$
$$daughter(X, Y) \leftarrow female(X), father(Y, X).$$

The hypothesis language is typically a subset of the language of program clauses. Since the complexity of learning grows with the expressiveness of the hypothesis language, restrictions have to be imposed on hypothesised clauses. Typical restrictions include a bound on the number of literals in a clause and restrictions on variables that appear in the body of the clause but not in its head (so-called new variables).

ILP systems typically adopt the covering approach of rule induction systems. In a main loop, they construct a clause explaining some of the positive examples, add this clause to the hypothesis, remove the positive examples explained and repeat this until all positive examples are explained (the hypothesis is complete). In an inner loop, individual clauses are constructed by (heuristically) searching the space of possible clauses, structured by a specialization or generalization operator. Typically, search starts with a very general rule (clause with no conditions in the body), then proceeds to add literals (conditions) to this clause until it only covers (explains) positive examples (the clause is consistent). This search can be bound from below by using so-called bottom clauses, constructed by least general generalization or inverse resolution/entailment. We discuss these issues in detail in Sections 5–7.

When dealing with incomplete or noisy data, which is most often the case, the criteria of consistency and completeness are relaxed. Statistical criteria are typically used instead. These are based on the number of positive and negative examples explained by the definition and the individual constituent clauses.

4 ILP for LLL

In many cases, ‘standard’ single predicate learning from positive and negative examples as outlined in Section 3 can be used fairly straightforwardly for natural language learning (NLL) tasks. For example, many of the papers reviewed by

Eineborg and Lindberg (this volume) show how the standard ILP approach can be applied to part-of-speech tagging.

In this section, we outline NLL problems that require a more complex ILP approach. We will use grammar learning and morphology as examples, beginning with grammar learning.

Table 2. Learning a grammar rule for verb phrases

Training examples		*Background knowledge*	
$vp(1,3)$	$\oplus$	$np(0,1)$	$word('She',0,1)$
		$vp(1,2)$	$word(ran,1,2)$
		$mod(2,3)$	$word(quickly,2,3)$

4.1 Learning from Positive Examples Only

Consider the very simple language learning task described in Table 2. The background knowledge is a set of facts about the sentence *She ran quickly*, which, in practice, would have been derived using a background grammar and lexicon, which we do not include here. An ILP algorithm would easily construct the following single clause theory H:

$$vp(S,F) \leftarrow vp(S,M), mod(M,F)$$

which represents the grammar rule

$$\text{VP} \rightarrow \text{VP MOD}$$

and which is consistent and complete with respect to the background knowledge and the training example. However many other clauses are also consistent and complete, including overgeneral clauses such as:

$$vp(S,F) \leftarrow$$
$$vp(S,F) \leftarrow word(ran,S,M)$$
$$vp(S,F) \leftarrow vp(S,M)$$

So an ILP algorithm needs something more than consistency and completeness if it is to select the correct hypothesis in this case. This problem stems from a common feature of NLP learning tasks: the absence of explicit negative examples. This absence allowed the wildly overgeneral rule $vp(S,F) \leftarrow$ (VPs are everywhere!) to be consistent. A suitably defined hypothesis language which simply disallows many overgeneral clauses can only partially overcome this problem. One option is to invoke a Closed World Assumption (CWA) to produce implicit negative examples. In our example LLL problem this would amount to the (correct) assumption that VPs only occur at the places stated in the examples and

background knowledge. We make these negative examples explicit in Table 3. Although correct in this case, such an assumption will not always be so. If the fact $vp(1,2)$ were missing from background knowledge the CWA would incorrectly assert that the clause $vp(1,2)$ were false.

Table 3. Learning a grammar rule for verb phrases, with explicit negatives.

Training examples		*Background knowledge*	
$vp(1,3)$	$\oplus$	$np(0,1)$	$word('She',0,1)$
$vp(0,1)$	$\ominus$	$vp(1,2)$	$word(ran,1,2)$
$vp(0,2)$	$\ominus$	$mod(2,3)$	$word(quickly,2,3)$
$vp(0,3)$	$\ominus$		
$vp(1,3)$	$\ominus$		
$vp(2,3)$	$\ominus$		

Since a CWA is generally too strong, an alternative is to assume that *most* other potential examples are negative. Muggleton (2000) gives a probabilistic method along these lines for learning from positive examples where the probability of a hypothesis decreases with its generality and increases with its compactness. Boström (1998) also uses a probabilistic approach to positive-only learning based on learning Hidden Markov Models (HMMs).

A different approach rests on the recognition that grammar learning, and many other NLL tasks, are not primarily classification tasks. In this view a grammar is not really concerned with differentiating sentences (positive examples) from non-sentences (negative examples). Instead, given a string of words assumed to be a sentence, the goal is to find the correct *parse* of that sentence. From this view, the problem with $vp(S,F) \leftarrow$ is not so much that it may lead to non-sentences being incorrectly classified as sentences; but rather that it will lead to a given sentence having many different parses—making it harder to find the correct parse. A shift from classification to disambiguation raises many important issues for the application of ILP to NLP which we can not discuss here. This issue is addressed in the papers by Mooney (this volume) and Thompson and Califf (this volume) which present the CHILL algorithm which 'learns to parse' rather than classify. Riezler's paper (this volume) also focusses on disambiguation, employing a statistical approach.

Output Completeness is a type of CWA introduced by Mooney and Califf (1995) which allows a decision list learning system to learn from positive only data—an important consideration for language learning applications. Decision lists are discussed in Section 4.3. We illustrate the notion of output completeness using a simple example.

In Table 4, there are *two* possible plural forms for `fish` whereas there is only one plural form for `lip`. Given these examples, under the output completeness assumption, every example of the form `plural([l,i,p], Y)` where `Y` $\neq$ `[l,i,p,s]` is a negative example. Similarly, every example of the form

Table 4. Positive examples for learning about plurals

```
plural([f,i,s,h], [f,i,s,h]).
plural([f,i,s,h], [f,i,s,h,e,s]).
plural([l,i,p], [l,i,p,s]).
```

`plural([f,i,s,h],Y)` where `Y` $\neq$ `[f,i,s,h]` and `Y` $\neq$ `[f,i,s,h,e,s]` is a negative example. Thus, output completeness assumption allows us to assume that the training data set contains, for every example, the complete set of output values, for every input value present in the data set.

4.2 Multiple Predicate Learning

So far we have been looking at cases where we learn from examples of a single target predicate. In grammar learning we may have to learn clauses for several predicates (multiple predicate learning) and also the examples may not be examples of any of the predicates we need to learn. Consider the learning problem described in Table 5. The target theory is the two clause theory:

$$v(X,Y) \leftarrow word(ran, X, Y)$$
$$vp(S,F) \leftarrow vp(S,M), mod(M,F)$$

Table 5. Grammar completion

Training examples		*Background knowledge*	
$s(0,3)$	$\oplus$	$np(X,Y) \leftarrow word('She', X, Y)$	$word('She', 0, 1)$
		$mod(X,Y) \leftarrow word(quickly, X, Y)$	$word(quickly, 2, 3)$
		$s(X,Y) \leftarrow np(X,Z), vp(Z,Y)$	$word(ran, 1, 2)$
		$vp(X,Y) \leftarrow v(X,Y)$	

The problem in Table 5 is to induce a theory H with certain properties. For example, it must be in the hypothesis language, not be over-general, not increase the ambiguity of the grammar too much, etc. However here we will focus on the constraint that $s(0,3)$ must follow from $B \wedge H$. Neither of our two clauses *directly* define the predicate $s/2$, the predicate symbol of our example, but we can use *abduction* to generate positive examples with the right predicate symbol. Abduction is a form of reasoning introduced by Pierce which can be identified with the following inference rule:

$$\frac{C,\ C \leftarrow A}{A}$$

This formalises the following form of reasoning (Pierce, 1958), quoted in (Flach & Kakas, 2000):

> The surprising fact, C, is observed;
> But if A were true, C would be a matter of course,
> Hence, there is reason to suspect that A is true

In our example we have the positive example $s(0,3)$ which requires explanation. We also have $s(0,3) \leftarrow vp(1,3)$ which follows from B. Abduction allows us derive $vp(1,3)$ which we can then treat as a normal positive example and attempt to learn $vp(S,F) \leftarrow vp(S,M), mod(M,F)$. Abduction is, of course, unsound: we are just guessing that $vp(1,3)$ is true, and sometimes our guesses will be wrong. For example, from $vp(1,3)$ and $vp(X,Y) \leftarrow v(X,Y)$, which are both true, we can abductively derive $v(1,3)$ which is not. Abductive approaches applied to LLL problems can be found in (Muggleton & Bryant, 2000; Cussens & Pulman, 2000) and Cussens and Pulman (this volume).

4.3 Decision List Learning for NLP

Many linguistic phenomena can be explained by rules together with exceptions to those rules, but the pure logic programs induced by standard ILP algorithms do not express exceptions easily. For example, this clause:

```
plural(X,Y) :- mate(X,Y,[],[],[],[s]).
```

says that for any word X, *without exception* the plural Y is formed by adding an ‘`s`’. In a pure logic program this is the only possible interpretation of the rule, even if we have somehow stated the exceptions to the rule elsewhere in the logic program. A solution to this problem is to use first-order decision lists. These can be thought of as an ordered set of clauses with specific clauses at the top and general clauses at the bottom. For example, Table 6 shows a decision list learnt by CLOG (Manandhar, Džeroski, & Erjavec, 1998) from the training examples given in Table 7[1].

Table 6. Decision list induced by CLOG training data in Table 7

```
plural([s,p,y],[s,p,i,e,s]):- !.
plural(X,Y) :- mate(X,Y,[],[],[a,n],[e,n]),!.
plural(X,Y) :- mate(X,Y,[],[],[a,s,s],[a,s,s,e,s]),!.
plural(X,Y) :- mate(X,Y,[],[],[],[s]),!.
```

The first clause in Table 6 is an exception. Although the general rule in English is to change the ending *-y* to *-ies*, CLOG was not able to learn this rule

[1] CLOG is available from `www.cs.york.ac.uk/~suresh/CLOG`

Table 7. Sample training data set for CLOG.

```
plural([l,i,p], [l,i,p,s]).
plural([m,e,m,b,e,r], [m,e,m,b,e,r,s]).
plural([d,a,y], [d,a,y,s]).
plural([s,e,c,o,n,d], [s,e,c,o,n,d,s]).
plural([o,t,h,e,r], [o,t,h,e,r,s]).
plural([l,i,e], [l,i,e,s]).
plural([m,a,s,s], [m,a,s,s,e,s]).
plural([c,l,a,s,s], [c,l,a,s,s,e,s]).
plural([s,p,y], [s,p,i,e,s]).
plural([m,a,n], [m,e,n]).
plural([w,o,m,a,n], [w,o,m,e,n]).
plural([f,a,c,e], [f,a,c,e,s]).
```

because there was only one example in the training set showing this pattern. The second rule states that if a word ends in an *-an* then the plural form replaces it with *-en*. The last rule is the default rule that will apply if the previous rules fail. This default rule will add an *-s* ending to any word. Decision lists are implemented in PROLOG by adding a cut (!) to the end of every clause. PROLOG will then only apply one clause—the first one that succeeds.

There are several reasons why decision lists are attractive for language learning applications. In contrast with pure logic programs, output-completeness provides a straightforward mechanism to learn decision lists from positive only data. Again, in contrast with pure logic programs which can return multiple answers nondeterministically, decision lists are learnt so that the most likely answer is returned when multiple solutions are possible. For example, in part-of-speech (PoS) tagging, every word in a given sentence needs to be assigned a *unique* PoS tag from a *fixed* set of tags. PoS tags are things such as VT (transitive verb), ADJ (adjective), NN (common noun), DET (determiner) etc. Words are often ambiguous. The task of a PoS tagger is to determine from context the correct PoS tag for every word. Thus, in *the forest fires have died*, a PoS tagger should assign NN to *fires* and not VT. Given a sentence, one is interested in the most probable tag sequence and not the set of plausible tag sequences.

Transformation lists have been popularised by Brill's work (Brill, 1995) on learning of PoS tagging rules. As an example, consider the PoS tagging rules given in Table 8 (adapted from Brill, this volume). In transformation based tagging, the tag assigned to a word is changed by the rules depending upon the left and right context.

As far as the syntax for representing both transformation lists and decision lists is concerned there is hardly a difference. For instance, we can easily represent the rules in Table 8 in PROLOG form as given in Table 9. The main difference is that in a decision list once a rule has been applied to a test example no further changes to the example take place. On the other hand, in a transformation list, once a rule has been applied to a test example, the example is still subject to further rule application and it is not removed from the test set. In fact, the

Table 8. Transformation based tagging rules

- Change tag from IN to RB if the current word is *about* and the next tag is CD.
- Change tag from IN to RB if the current word is *about* and the next tag is $.
- Change tag from NNS to NN if the current word is *yen* and the previous tag is CD.
- Change tag from NNPS to NNP if the previous tag is NNP.

rules in a transformation list apply to all the examples in the test set repeatedly until no further application is possible. Thus a transformation list successively transforms an input set of examples until no further application is possible. The resulting set is the solution produced by the transformation list.

Table 9. PROLOG representation of the tagging rules in Table 8

```
tag_rule( _, 'IN'-about, [ 'CD'-_, _ ], 'RB'-about) :- !.
tag_rule( _, 'IN'-about, [ '$'-_, _ ], 'RB'-about) :- !.
tag_rule( [_, 'CD'- _], 'NNS'-yen, _ , 'NN'-yen) :- !.
tag_rule( [ _, 'NNP'-_], 'NNPS'-X, _, 'NNP'-X) :- !.
```

Given this background, it is straightforward to generalise first order decision lists to first order transformation lists thereby lifting Brill's transformation based learning into an ILP setting (see for instance, (Dehaspe & Forrier, 1999)). This further justifies the use of decision lists and its extension to first order transformation lists for language learning applications. Finally, existing work (Quinlan, 1996; Manandhar et al., 1998) demonstrate that there exists relatively efficient decision list learning algorithms.

5 Structuring the Space of Clauses

In order to search the space of clauses (program clauses) systematically, it is useful to impose some structure upon it, e.g., an ordering. One such ordering is based on θ-subsumption, defined below.

Recall first that a substitution $\theta = \{V_1/t_1, ..., V_n/t_n\}$ is an assignment of terms t_i to variables V_i. Applying a substitution θ to a term, atom, or clause F yields the instantiated term, atom, or clause $F\theta$ where all occurrences of the variables V_i are simultaneously replaced by the term t_i.

Let c and c' be two program clauses. Clause c *θ-subsumes* c' if there exists a substitution θ, such that $c\theta \subseteq c'$ (Plotkin, 1969).

To illustrate the above notions, consider the clause c

$$c = daughter(X, Y) \leftarrow parent(Y, X).$$

Applying the substitution $\theta = \{X/mary, Y/ann\}$ to clause c yields

$$c\theta = daughter(mary, ann) \leftarrow parent(ann, mary).$$

Recall that the clausal notation $daughter(X, Y) \leftarrow parent(Y, X)$ stands for

$$\{daughter(X, Y), \overline{parent(Y, X)}\}$$

where all variables are assumed to be universally quantified and the commas denote disjunction. According to the definition, clause c θ-subsumes c' if there is a substitution θ that can be applied to c such that every literal in the resulting clause occurs in c'. Clause c θ-subsumes the clause

$$c' = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

under the empty substitution $\theta = \emptyset$, since $\{daughter(X, Y), \overline{parent(Y, X)}\}$ is a proper subset of $\{daughter(X, Y), \overline{female(X)}, \overline{parent(Y, X)}\}$. Furthermore, clause c θ-subsumes the clause
$c' = daughter(mary, ann) \leftarrow$
$female(mary), parent(ann, mary), parent(ann, tom)$
under the substitution $\theta = \{X/mary, Y/ann\}$.

θ-subsumption introduces a syntactic notion of generality. Clause c *is at least as general as* clause c' ($c \leq c'$) if c *θ-subsumes* c'. Clause c *is more general than* c' ($c < c'$) if $c \leq c'$ holds and $c' \leq c$ does not. In this case, we say that c' *is a specialization of* c and c *is a generalization of* c'. If the clause c' is a specialization of c then c' is also called a *refinement* of c. The only clause refinements usually considered by ILP systems are the minimal (most general) specializations of the clause.

There are two important properties of θ-subsumption:

- If c θ-subsumes c' then c logically entails c', $c \models c'$. The reverse is not always true. As an example, Flach (1992) gives the following two clauses $c = list([V|W]) \leftarrow list(W)$ and $c' = list([X, Y|Z]) \leftarrow list(Z)$. Given the empty list, c constructs lists of any given length, while c' constructs lists of even length only, and thus $c \models c'$. However, no substitution exists that can be applied to c to yield c', since it should map W both to $[Y|Z]$ and to Z which is impossible. Therefore, c does not θ-subsume c'.
- The relation $\leq$ introduces a *lattice* on the set of all clauses (Plotkin, 1969). This means that any two clauses have a least upper bound (*lub*) and a greatest lower bound (*glb*). Both the *lub* and the *glb* are unique up to equivalence (renaming of variables) under θ-subsumption. For example, the clauses

$$daughter(X, Y) \leftarrow parent(Y, X), parent(W, V)$$

and

$$daughter(X, Y) \leftarrow parent(Y, X)$$

θ-subsume one another.

The second property of θ-subsumption leads to the following definition: The *least general generalization* (lgg) of two clauses c and c', denoted by $lgg(c, c')$, is the least upper bound of c and c' in the θ-subsumption lattice (Plotkin, 1969). The rules for computing the lgg of two clauses are outlined later in this chapter.

Note that θ-subsumption and least general generalization are purely syntactic notions since they do not take into account any background knowledge. Their computation is therefore simple and easy to implement in an ILP system. The same holds for the notion of generality based on θ-subsumption. On the other hand, taking background knowledge into account would lead to the notion of *semantic generality* (Niblett, 1988; Buntine, 1988), defined as follows: Clause c is *at least as general* as clause c' with respect to background theory B if $B \cup \{c\} \models c'$.

The syntactic, θ-subsumption based, generality is computationally more feasible. Namely, semantic generality is in general undecidable and does not introduce a lattice on a set of clauses. Because of these problems, syntactic generality is more frequently used in ILP systems.

θ-subsumption is important for inductive logic programming for the following reasons:

- As shown above, it provides a generality ordering for hypotheses, thus structuring the hypothesis space. It can be used to prune large parts of the search space.
- θ-subsumption provides the basis for two important ILP techniques:
  - top-down *searching of refinement graphs*, and
  - *building of least general generalizations* from training examples, relative to background knowledge, which can be used to bound the search of refinement graphs from below or as part of a bottom-up search.

These two techniques will be elaborated upon in the following sections.

6 Searching the Space of Clauses

Most ILP approaches search the hypothesis space (of program clauses) in a top-down manner, from general to specific hypotheses, using a θ-subsumption-based *specialization operator*. A specialization operator is usually called a *refinement operator* (Shapiro, 1983). Given a hypothesis language $\mathcal{L}$, a refinement operator ρ maps a clause c to a set of clauses $\rho(c)$ which are specializations (refinements) of c: $\rho(c) = \{c' \mid c' \in \mathcal{L},\ c < c'\}$.

A refinement operator typically computes only the set of minimal (most general) specializations of a clause under θ-subsumption. It employs two basic syntactic operations on a clause:

- apply a substitution to the clause, and
- add a literal to the body of the clause.

The hypothesis space of program clauses is a lattice, structured by the θ-subsumption generality ordering. In this lattice, a *refinement graph* can be defined as a directed, acyclic graph in which *nodes* are program clauses and *arcs*

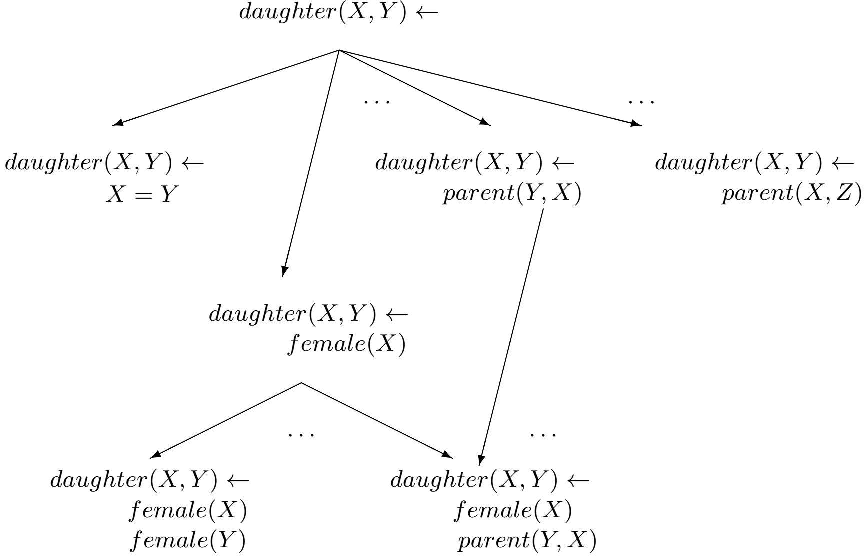

Fig. 2. Part of the refinement graph for the family relations problem.

correspond to the basic refinement operations: substituting a variable with a term, and adding a literal to the body of a clause.

Figure 2 depicts a part of the refinement graph for the family relations problem defined in Table 1, where the task is to learn a definition of the *daughter* relation in terms of the relations *female*, and *parent*.

At the top of the refinement graph (lattice) is the clause

$$c = daughter(X,Y) \leftarrow$$

where an empty body is written instead of the body *true*. The refinement operator ρ generates the refinements of c, which are of the form

$$\rho(c) = \{daughter(X,Y) \leftarrow L\}$$

where L is one of following literals:

- literals having as arguments the variables from the head of the clause: $X = Y$ (this corresponds to applying a substitution X/Y), $female(X)$, $female(Y)$, $parent(X,X)$, $parent(X,Y)$, $parent(Y,X)$, and $parent(Y,Y)$, and
- literals that introduce a new distinct variable Z ($Z \neq X$ and $Z \neq Y$) in the clause body: $parent(X,Z)$, $parent(Z,X)$, $parent(Y,Z)$, and $parent(Z,Y)$.

The search for a clause starts at the top of the lattice, with the clause that covers all example (positive and negative). Its refinements are then considered, then their refinements in turn, and this is repeated until a clause is found which covers only positive examples. In the example above, the clause

$daughter(X, Y) \leftarrow female(X), parent(Y, X)$ is such a clause. Note that this clause can be reached in several ways from the top of the lattice, e.g., by first adding $female(X)$, then $parent(Y, X)$ or vice versa.

The refinement graph is typically searched heuristically level-wise, using heuristics based on the number of positive and negative examples covered by a clause. As the branching factor is very large, greedy search methods are typically applied which only consider a limited number of alternatives at each level. Hill-climbing considers only one alternative at each level, while beam search considers n alternatives, where n is the beam width. Occasionally, complete search is used, e.g., A^* best-first search or breadth-first search. Often the search can be pruned. For example, since we are only interested in clauses that (together with background knowledge) entail at least one example, we can prune the search if we ever construct a clause which entails no positive examples. This is because if a clause covers no positive examples then neither will any of its refinements.

7 Bounding the Search for Clauses

The branching factor of a refinement graph, i.e., the number of refinements a clause has, is very large. This is especially true for clauses deeper in the lattice that contain many variables. It is thus necessary to find ways to reduce the space of clauses actually searched.

One approach is to make the refinement graph smaller by making the refinement operator take into account the types of predicate arguments, as well as input/output mode declarations. For example, we might restrict the $parents(Y, X)$ predicate to only give us the parents of a given child and not give us persons that are the offspring of a given person. Also we could restrict $parents(Y, X)$, so that X can only be instantiated with a term of type *child*. This can be done with a mode declaration $parents(-person, +child)$.

Type and mode declarations can be combined with the construction of a bottom clause that bounds the search of the refinement lattice from below. This is the most specific clause covering a given example (or examples). Only clauses on the path between the top and the bottom clause are considered, significantly improving efficiency. This approach is implemented in the Progol algorithm (Muggleton, 1995).

The bottom clause can be constructed as the relative least general generalization of two (or more) examples (Muggleton & Feng, 1990) or the most specific inverse resolvent of an example (Muggleton, 1991), both with respect to a given background knowledge B. entailment. These methods are discussed below.

7.1 Relative Least General Generalization

Plotkin's notion of least general generalization (*lgg*) (Plotkin, 1969) forms the basis of cautious generalization: the latter assumes that if two clauses c_1 and c_2 are true, it is very likely that $lgg(c_1, c_2)$ will also be true.

The least general generalization of two clauses c and c', denoted $lgg(c, c')$, is the least upper bound of c and c' in the θ-subsumption lattice. It is the most

specific clause that θ-subsumes c and c'. If a clause d θ-subsumes c and c', it has to subsume $lgg(c, c')$ as well. To compute the lgg of two clauses, lgg of terms and literals need to be defined first (Plotkin, 1969).

The lgg of two terms $lgg(t_1, t_2)$ is computed as

1. $lgg(t, t) = t$,
2. $lgg(f(s_1, .., s_n), f(t_1, .., t_n)) = f(lgg(s_1, t_1), .., lgg(s_n, t_n))$.
3. $lgg(f(s_1, .., s_m), g(t_1, .., t_n)) = V$, where $f \neq g$, and V is a variable which represents $lgg(f(s_1, .., s_m), g(t_1, .., t_n))$,
4. $lgg(s, t) = V$, where $s \neq t$ and at least one of s and t is a variable; in this case, V is a variable which represents $lgg(s, t)$.

For example,

$$lgg([a, b, c], [a, c, d]) = [a, X, Y]$$

and

$$lgg(f(a, a), f(b, b)) = f(lgg(a, b), lgg(a, b)) = f(V, V)$$

where V stands for $lgg(a, b)$. When computing lggs one must be careful to use the same variable for multiple occurrences of the lggs of subterms, i.e., $lgg(a, b)$ in this example. This holds for lggs of terms, atoms and clauses alike.

The lgg of two atoms $lgg(A_1, A_2)$ is computed as follows:

1. $lgg(p(s_1, .., s_n), p(t_1, .., t_n)) = p(lgg(s_1, t_1), .., lgg(s_n, t_n))$, if atoms have the same predicate symbol p,
2. $lgg(p(s_1, .., s_m), q(t_1, .., t_n))$ is undefined if $p \neq q$.

The lgg of two literals $lgg(L_1, L_2)$ is defined as follows:

1. if L_1 and L_2 are atoms, then $lgg(L_1, L_2)$ is computed as defined above,
2. if both L_1 and L_2 are negative literals, $L_1 = \overline{A_1}$ and $L_2 = \overline{A_2}$, then $lgg(L_1, L_2) = lgg(\overline{A_1}, \overline{A_2}) = \overline{lgg(A_1, A_2)}$,
3. if L_1 is a positive and L_2 is a negative literal, or vice versa, $lgg(L_1, L_2)$ is undefined.

For example,

$$\begin{aligned} lgg(parent(ann, mary), parent(ann, tom)) &= parent(ann, X) \\ lgg(parent(ann, mary), \overline{parent(ann, tom)}) &\quad \text{is undefined} \\ lgg(parent(ann, X), daughter(mary, ann)) &\quad \text{is undefined} \end{aligned}$$

Taking into account that clauses are sets of literals, the lgg of two clauses is defined as follows. Let $c_1 = \{L_1, .., L_n\}$ and $c_2 = \{K_1, .., K_m\}$. Then $lgg(c_1, c_2) = \{M_{ij} = lgg(L_i, K_j) \mid L_i \in c_1, K_j \in c_2, lgg(L_i, K_j)$ is defined $\}$. If

$$c_1 = daughter(mary, ann) \leftarrow female(mary), parent(ann, mary)$$

and

$$c_2 = daughter(eve, tom) \leftarrow female(eve), parent(tom, eve)$$

then

$$lgg(c_1, c_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

where X stands for $lgg(mary, eve)$ and Y stands for $lgg(ann, tom)$.

In (Cussens & Pulman, 2000), lgg is used as the basis for a bottom-up search for clauses representing grammar rules. The search is implemented in Prolog using the Sicstus Prolog library built-in `term_subsumer/3` to construct lggs. `term_subsumer/3` finds lggs of terms, not clauses, but since the clauses in (Cussens & Pulman, 2000) are always unit (single-literal) clauses we can find lggs by presenting these unit clauses to `term_subsumer/3` as if they were terms. Table 10 shows two grammar rules and their lgg. Table 11 gives human-readable decompiled versions of the three rules in Table 10. These *unification-based grammar* rules are more sophisticated versions of the context-free rule VP $\rightarrow$ VP MOD where the linguistic categories (i.e. VP and MOD) have features which must match up correctly. See Thompson (this volume) for a comparison of context-free and unification-based grammars.

Rule `r67` involves, amongst other things, a generalisation of the 'gap-threading' patterns seen in Rules `r3` and `r24`. Gap-threading is a technique for dealing with movement phenomena in syntax (Pereira, 1981). Adding rule `r3` to the initial incomplete grammar in (Cussens & Pulman, 2000) allows the sentence *All big companies wrote a report quickly.* to be parsed, adding rule `r24` allows *What don't all big companies read with a machine?* to be parsed; adding the lgg rule `r67` allows both to be parsed.

Table 10. Two grammar rules and their lgg

```
cmp_synrule(
vp([ng,ng],f(0,0,0,0,1,1,1,1,1),n),
[vp([ng,ng],f(0,0,0,0,1,1,1,1,1),n),
mod([ng,ng],f(0,0,0,1),f(0,1,1,1))]
).

cmp_synrule(
vp([np([ng,ng],f(0,0,0,0,X257,X257,X257,1,1),
f(0,1,1,1),nonsubj),ng],f(0,0,1,1,1,1,1,1,1),n),
[vp([np([ng,ng],f(0,0,0,0,X257,X257,X257,1,1),
f(0,1,1,1),nonsubj),ng],f(0,0,1,1,1,1,1,1,1),n),
mod([ng,ng],f(0,X187,X187,1),f(0,1,1,1))]
).

cmp_synrule(
vp([X231,ng],f(0,0,X223,X223,1,1,1,1,1),n),
[vp([X231,ng],f(0,0,X223,X223,1,1,1,1,1),n),
mod([ng,ng],f(0,X187,X187,1),f(0,1,1,1))]
).
```

Table 11. Two grammar rules and their lgg (uncompiled version)

```
r3 vp ==> [vp,mod]
vp:[gaps=[ng:[],ng:[]],mor=pl,aux=n]==>
[vp:[gaps=[ng:[],ng:[]],mor=pl,aux=n],
mod:[gaps=[ng:[],ng:[]],of=vp,type=n]]

r24 vp ==> [vp,mod]
vp:[gaps=[np:[gaps=[ng:[],ng:[]],mor=or(pl,s3),type=n,case=nonsubj],
ng:[]],mor=inf,aux=n]==>
[vp:[gaps=[np:[gaps=[ng:[],ng:[]],mor=or(pl,s3),type=n,case=nonsubj],
ng:[]],mor=inf,aux=n],
mod:[gaps=[ng:[],ng:[]],of=or(nom,vp),type=n]]

r67 vp ==> [vp,mod]
vp:[gaps=[X283,ng:[]],mor=or(inf,pl),aux=n]==>
[vp:[gaps=[X283,ng:[]],mor=or(inf,pl),aux=n],
mod:[gaps=[ng:[],ng:[]],of=or(nom,vp),type=n]]
```

The definition of *relative least general generalization (rlgg)* is based on the semantic notion of generality. The *rlgg* of two clauses c_1 and c_2 is the least general clause which is more general than both c_1 and c_2 with respect (*relative*) to background knowledge B. The notion of *rlgg* was used in the ILP system GOLEM (Muggleton & Feng, 1990). To avoid the problems with the semantic notion of generality, the background knowledge B in GOLEM is restricted to ground facts. If K denotes the conjunction of all these facts, the *rlgg* of two ground atoms A_1 and A_2 (positive examples), relative to B can be computed as: $rlgg(A_1, A_2) = lgg((A_1 \leftarrow K), (A_2 \leftarrow K))$.

Given the positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$ and the background knowledge B for the family example, the least general generalization of e_1 and e_2 relative to B is computed as: $rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$ where K denotes the conjunction of the literals $parent(ann, mary)$, $parent(ann, tom)$, $parent(tom, eve)$, $parent(tom, ian)$, $female(ann)$, $female(mary)$, and $female(eve)$.

For notational convenience, the following abbreviations are used: *d-daughter, p-parent, f-female, a-ann, e-eve, m-mary, t-tom, i-ian.* The conjunction of facts from the background knowledge (comma stands for conjunction) is

$$K = p(a, m), p(a, t), p(t, e), p(t, i), f(a), f(m), f(e).$$

The computation of $rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$, produces the following clause

$$\begin{aligned}\mathbf{d}(\mathbf{V_{m,e}}, \mathbf{V_{a,t}}) \leftarrow & p(a, m), p(a, t), p(t, e), p(t, i), f(a), f(m), f(e),\\ & p(a, V_{m,t}), \mathbf{p}(\mathbf{V_{a,t}}, \mathbf{V_{m,e}}), p(V_{a,t}, V_{m,i}), p(V_{a,t}, V_{t,e}),\\ & p(V_{a,t}, V_{t,i}), p(t, V_{e,i}), f(V_{a,m}), f(V_{a,e}), \mathbf{f}(\mathbf{V_{m,e}}).\end{aligned}$$

In the above clause, $V_{x,y}$ stands for $lgg(x,y)$, for each x and y. The three literals in **bold** are those that will remain after literals failing to meet various criteria are removed. Now we will describe these criteria and how they can be used to reduce the size of a large *rlgg*.

In general, a *rlgg* of training examples can contain infinitely many literals or at least grow exponentially with the number of examples. Since such a clause can be intractably large, constraints are used on introducing new variables into the body of the *rlgg*. For example, literals in the body that are not connected to the head by a chain of variables are removed. In the above example, this yields the clause $\mathbf{d}(\mathbf{V_{m,e}},\mathbf{V_{a,t}}) \leftarrow \mathbf{p}(\mathbf{V_{a,t}},\mathbf{V_{m,e}}), p(V_{a,t},V_{m,i}), p(V_{a,t},V_{t,e}), p(V_{a,t},V_{t,i}), \mathbf{f}(\mathbf{V_{m,e}})$. Also, nondeterminate literals (that can give more than one value of output arguments for a single value of the input arguments) may be eliminated. The literals $p(V_{a,t},V_{m,i}), p(V_{a,t},V_{t,e}), p(V_{a,t},V_{t,i})$ are nondeterminate since $V_{a,t}$ is the input argument and a parent can have more than one child. Eliminating these yields the bottom clause $\mathbf{d}(\mathbf{V_{m,e}},\mathbf{V_{a,t}}) \leftarrow \mathbf{p}(\mathbf{V_{a,t}},\mathbf{V_{m,e}}), \mathbf{f}(\mathbf{V_{m,e}})$, i.e., $daughter(X,Y) \leftarrow female(X), parent(Y,X)$. In this simple example, the bottom clause is our target clause. In practice, the bottom clause is typically very large, containing hundreds of literals.

7.2 Inverse Resolution

The basic idea of *inverse resolution* introduced by Muggleton and Buntine (1988), is to invert the *resolution* rule of deductive inference (Robinson, 1965), i.e., to invert the SLD-resolution proof procedure for definite programs (Lloyd, 1987). The basic resolution step in propositional logic derives the proposition $p \vee r$ given the premises $p \vee \overline{q}$ and $q \vee r$. In a first-order case, resolution is more complicated, involving substitutions. Let $res(c,d)$ denote the resolvent of clauses c and d.

To illustrate resolution in first-order logic, we use the grammar example from earlier. Suppose that background knowledge B consists of the clauses $b_1 = vp(1,2)$ and $b_2 = mod(2,3)$ and $H = \{c\} = \{vp(S,E) \leftarrow vp(S,M), mod(M,E)\}$. Let $T = H \cup B$. Suppose we want to derive the fact $vp(1,3)$ from T. To this end, we proceed as follows:

- First, the resolvent $c_1 = res(c,b_1)$ is computed under the substitution $\theta_1 = \{S/1, M/2\}$. This means that the substitution θ_1 is first applied to clause c to obtain $vp(1,E) \leftarrow vp(1,2), mod(2,E)$, which is then resolved with b_1 as in the propositional case. The resolvent of $vp(S,E) \leftarrow vp(S,M), mod(M,E)$ and $vp(1,2)$ is thus $c_1 = res(c,b_1) = vp(1,E) \leftarrow mod(2,E)$.
- The next resolvent $c_2 = res(c_1,b_2)$ is computed under the substitution $\theta_2 = \{E/3\}$. The clauses $vp(1,E) \leftarrow mod(2,E)$ and $mod(2,3)$ resolve in $c_2 = res(c_1,b_2) = vp(1,3)$.

The linear derivation tree for this resolution process is given in Figure 3.

Inverse resolution, used in the ILP system CIGOL (Muggleton & Buntine, 1988), inverts the resolution process using generalization operators based on inverting substitution (Buntine, 1988). Given a well-formed formula W, an *in-*

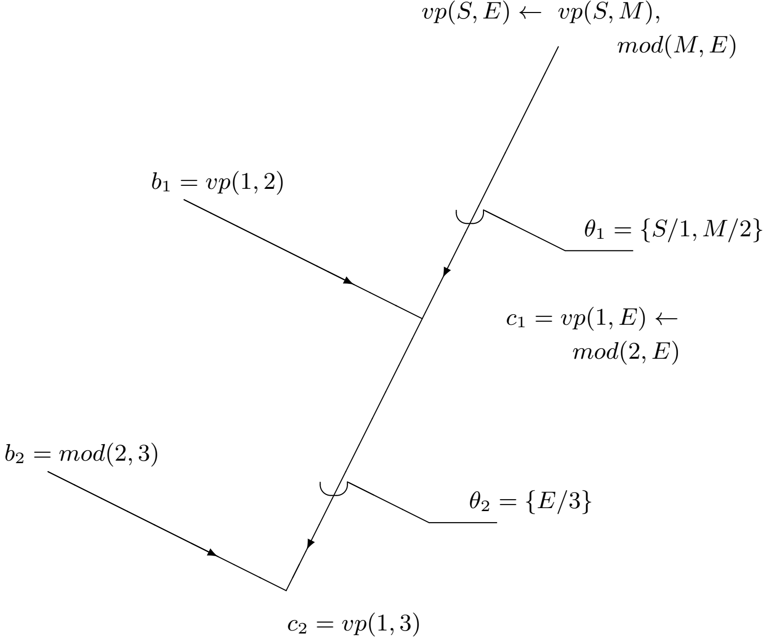

Fig. 3. A linear derivation tree.

verse substitution θ^{-1} of a substitution θ is a function that maps terms in $W\theta$ to variables, such that $W\theta\theta^{-1} = W$.

Let $c = vp(S,E) \leftarrow vp(S,M), mod(M,E)$ and $\theta = \{S/1, M/2, E/3\}$: then $c' = c\theta = vp(1,3) \leftarrow vp(1,2), mod(2,3)$. By applying the inverse substitution $\theta^{-1} = \{1/S, 2/M, 3/E\}$ to c_1, the original clause c is restored: $c = c'\theta^{-1} = vp(S,E) \leftarrow vp(S,M), mod(M,E)$. In the general case, inverse substitution is substantially more complex. It involves the *places* of terms in order to ensure that the variables in the initial W are appropriately restored in $W\theta\theta^{-1}$. In fact, each occurrence of a term can be replaced by a different variable in an inverse substitution.

We will not treat inverse resolution in detail, but will rather illustrate it by an example. Let $ires(c,d)$ denote the inverse resolvent of clauses c and d. As in the example above, let background knowledge B consist of the two clauses $b_1 = vp(1,2)$ and $b_2 = mod(2,3)$. The inverse resolution process might then proceed as follows:

- In the first step, inverse resolution attempts to find a clause c_1 which will, together with b_2, entail e_1. Using the inverse substitution $\theta_2^{-1} = \{3/E\}$, an inverse resolution step generates the clause $c_1 = ires(b_2, e_1) = vp(1,E) \leftarrow mod(2,E)$.
- Inverse resolution then takes $b_1 = vp(1,2)$ and c_1. It computes $c' = ires(b_1, c_1)$, using the inverse substitution $\theta_1^{-1} = \{1/S, 2/M\}$, yielding $c' = vp(S,E) \leftarrow vp(S,M), mod(M,E)$.

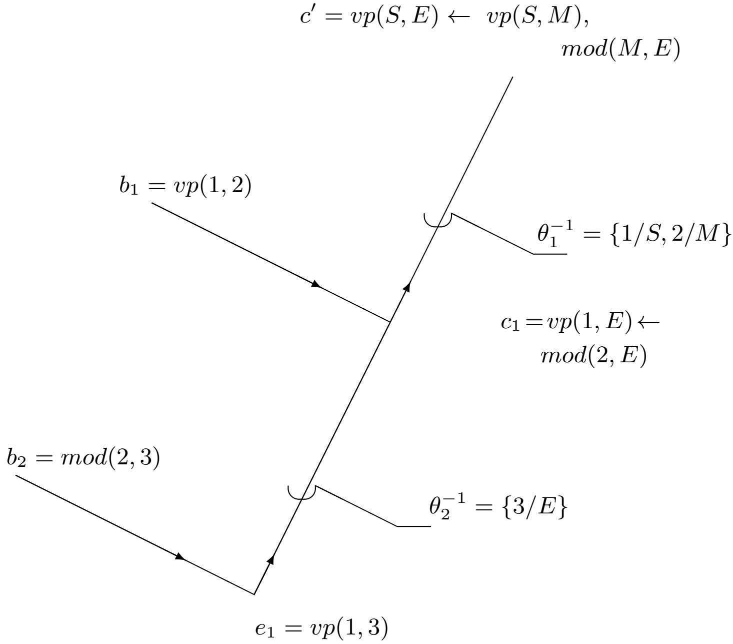

Fig. 4. An inverse linear derivation tree.

The corresponding inverse linear derivation tree is illustrated in Figure 4.

Most specific inverse resolution uses only empty inverse substitutions. In the above example, this would yield the clause $vp(1,3) \leftarrow vp(1,2), mod(2,3)$ as the final inverse resolvent. The process of repeatedly applying inverse resolution to an example to get such a most specific clause is known as *saturation* and was introduced by Rouveirol (1990). In practice, inverse substitutions replacing the occurrence of each constant with the same variable are considered to yield the most specific inverse resolvent, which can the be used as a bottom clause. In our case, this is again the target clause, while in practice the bottom clause will be much more specific/larger, containing a large number of literals.

7.3 Inverse Entailment

Inverse entailment (IE) was introduced in (Muggleton, 1995) and defines the problem of finding clauses from a model-theoretic viewpoint, as opposed to the proof-theoretic definition of inverse resolution. If H is such that

$$B \wedge H \models E$$

then

$$B \wedge \overline{E} \models \overline{H}$$

Let $\overline{\bot}$ be the potentially infinite collection of ground literals which are true in every model of $B, \overline{E}$. Note that $\bot$ is *not* being used to denote falsity as in standard logical notation. It follows that

$$B \wedge \overline{E} \models \overline{\bot} \models \overline{H}$$

and

$$H \models \bot$$

It follows that a subset of solutions for H can be found by considering clauses that θ-subsume the bottom clause $\bot$. If a clause H does subsume $\bot$ we say that H follows from B and E by inverse entailment. IE is implemented (in the Progol algorithm) by saturating E to produce $\bot$ and then searching the set of clauses that θ-subsume $\bot$ in a top-down manner.

Table 12 gives examples of two bottom clauses. Note that in the first example, the bottom clause is not a definite clause; this example shows how the construction of the bottom clause can be used to do abduction.

Table 12. Most specific 'bottom' clauses $\bot$ for various B and E

B	E	$\bot$
$vp(X,Y) \leftarrow v(X,Y)$ $v(X,Y) \leftarrow word(ran,X,Y)$	$vp(1,3)$	$vp(1,3) \vee v(1,3) \vee$ $word(ran,1,3)$
$vp(X,Y) \leftarrow v(X,Y)$ $v(X,Y) \leftarrow word(ran,X,Y)$	$is_english(X,Y) \leftarrow$ $word(ran,X,Y)$	$is_english(X,Y) \leftarrow$ $word(ran,X,Y),$ $vp(X,Y), v(X,Y)$

Neither inverse resolution nor inverse entailment as described above can construct a target clause from an example if the clause is recursive and is used more than once to deduce the example. This is particularly problematic in LLL applications such as grammar learning. Suppose we had an incomplete grammar lacking the target rule $vp(S,E) \leftarrow vp(S,M), mod(M,E)$ and so were unable to parse the sentence *She ran quickly from the telephone*. Suppose now that we had used abduction to (correctly) guess that there should be a VP from vertex 1 to 6, i.e. spanning the words *ran quickly from the telephone*. Suppose also that the rest of the grammar is sufficiently complete to infer that *ran* is a VP, *quickly* is a modifier and *from the telephone* is also a modifier. This gives us the following background B, positive example E and target H:

$$E = vp(1,6)$$
$$B = vp(1,2), mod(2,3), mod(3,6)$$
$$H = vp(S,E) \leftarrow vp(S,M), mod(M,E)$$

Consider now the derivation of E from B and H. This is illustrated in Fig 5 where the last three resolution steps have been squashed into one to conserve

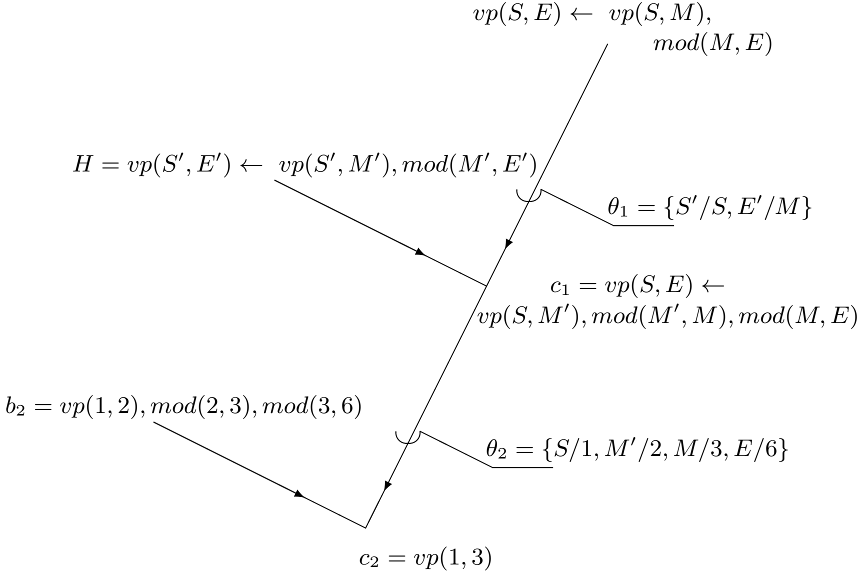

Fig. 5. An abbreviated linear derivation tree.

space. If we attempted to go up the tree by inverse resolution, then we would fail since we need H to derive H in the final step.

Approaching the problem from the IE perspective, we find that

$$\begin{aligned}\bot &= vp(1,6) \leftarrow vp(1,2), mod(2,3), mod(3,6)\\ &= \{vp(1,6)\overline{vp(1,2)}, \overline{mod(2,3)}, \overline{mod(3,6)}\}\end{aligned}$$

and although there is an $vp(S,E) \leftarrow\ vp(S,M), mod(M,E) = H$ such that

$$vp(S,E) \leftarrow\ vp(S,M), mod(M,E) \models vp(1,6) \leftarrow vp(1,2), mod(2,3), mod(3,6)$$

H does not *subsume* $\bot$. This is because there is no substitution θ such that

$$\{vp(S,E), \overline{vp(S,M)}, \overline{mod(M,E)}\}\theta \subseteq \{vp(1,6)\overline{vp(1,2)}, \overline{mod(2,3)}, \overline{mod(3,6)}\}$$

Muggleton (1998) works around this problem by defining enlarged bottom clauses, which contain all possible positive literals except those that follow from $B \wedge \overline{E}$. However, the enlarged bottom clauses are so large as to effectively not restrict the search for clauses.

8 Volume Overview

Grammars. Cussens & Pulman and also Osborne assume the existence of a given initial grammar that must be completed to parse sentences in a training

set of unparsable sentences. In both cases, candidate rules are generated from an unparsable sentence via the chart created by the failed parse. See Thompson (this volume) for a description of chart parsing. From an ILP point of view this chart is that part of the background knowledge *relevant* to the example sentence that produced it.

In Osborne's paper an stochastic context-free grammar is defined on a feature grammar via a context-free (backbone) grammar formed by mapping each category, distinct in terms of features, to an atomic symbol. Riezler takes a more direct approach, defining a log-linear model on the parses of constraint-based grammars—the goal here is disambiguation rather than finding 'missing' grammar rules. Riezler describes how to learn both features (structure) and parameters of such models from unannotated data. Note that the features of the model are not necessarily grammar rules.

Watkinson and Manandhar's paper shares a theme with that of Adriaans earlier work (Adriaans, 1999) in that both use the functional nature of Categorial Grammar to enable them to build up the structure of the sentences. However, both the learning settings and the results of the learning processes are different. While Watkinson and Manandhar use a stochastic compression technique to learn in a completely unsupervised setting, Adriaans's learner, working within the PAC learning model, uses an oracle providing a strongly supervised setting. Secondly, Adriaans uses the partial categories that he builds to develop a context-free grammar, whereas Watkinson and Manandhar build both a parsed corpus and a stochastic Categorial Grammar lexicon (which can be used as a probabilistic Categorial Grammar). Adriaans and de Haas's approach (this volume) is the application of the same techniques as in (Adriaans, 1999) to learning of simple Lambek grammars.

Semantics. Going against the grain of work in statistical computational linguistics, a number of contributions focus on semantics. Indeed, Mooney's paper argues that logical approaches to language learning have "most to offer at the level of producing semantic interpretations of complete sentences" since logical representations of meaning are "important and useful". Mooney backs this up with an overview of work on the CHILL system for learning semantic parsers. CHILL is also examined in the paper from Thompson & Califf, who demonstrate (empirically) the effectiveness of *active learning* where a learning algorithm selects the most informative inputs for a human to annotate. This theme of "how best to marry machine learning with human labor" is central to Brill's paper arguing that we should be "capitalizing on the relative strengths of people and machines".

Boström reports on learning *transfer rules* which translate between semantic representations (quasi-logical forms) for different languages; in this case French and English. Nedellec continues the semantic theme with the ILP system ASIUM, which learns ontologies and verb subcategorization frames from a parsed corpora.

Information extraction. As well as Chill, Thompson & Califf also describe and give experimental results for active learning with the Rapier system. Rapier is a system that learns information extraction (IE) rules using limited syntactic and semantic information. Junker et al also learn IE rules, examining how standard ILP approaches can be used. They also learn rules for text categorisation.

PoS tagging, morphological analysis and phonotactics. Part-of-speech tagging is a popular test-bed for the application of ILP to NLP. An overview of this area is provided by the paper from Eineborg and Lindberg. ILP approaches to tagging often use the tags of neighbouring words to disambiguate the focus word—however, typically, these neighbouring tags are (initially) unknown. Jorge & de Andrade Lopes tackle this problem by learning a succession of theories, starting with a base theory which only has lexical rules and where a theory makes use of the disambiguation achieved by previous theories.

Kazakov gives an overview of ILP approaches to morphology and argues that future work should make more extensive use of linguistic knowledge and be more closely integrated with other learning approaches. Džeroski & Erjavec use Clog to learn rules which assign the correct lemma (canonical form) to a word given the word and its tag. A statistical tagger is used to find the tags. Tjong Kim Sang & Nerbonne examine learning phonotactics to distinguish Dutch words from non-words, finding that the inclusion of background knowledge improves performance.

9 Some Challenges in Language Learning for ILP

The main advantages of ILP from a NLP perspective are that 1) the rules learnt by an ILP system are easily understandable by a linguist, 2) ILP allows easy integration of linguistic background knowledge such as language universal/specific syntactic/morphological/phonological principles and 3) ILP offers a first-order representation both for the hypothesis language and the background language. Each of these advantages are either unavailable or hard to integrate within a purely statistical learning system.

On the other hand, ILP systems suffer from certain disadvantages that make them less competitive compared to well engineered statistical systems. Although first-order representations are learnt, a disadvantage of ILP systems is efficiency and its effect on rule accuracy. One reason it is difficult to solve the efficiency issue is that most ILP implementations are general purpose learning systems that will accept any background knowledge and any training data set. In contrast, most statistical systems are engineered for a specific task. It follows that ILP systems specifically designed for a specific NLP task are likely to be more successful than general-purpose implementations.

ILP systems are ideally suited to language learning tasks which involve learning complex representations. By "complex representation" we mean representations which are not purely attribute-value. For instance, grammars such as

Head-Driven Phrase Structure Grammar (HPSG) (Pollard & Sag, 1987, 1994) employ a representation language similar to a description logic[2] called *typed feature structures* (TFS) (Carpenter, 1992). HPSG comes with a background linguistic theory expressed in the TFS language as schemas that applies to (a subset of) all languages. TFSs can be viewed as terms in a constraint language (Smolka, 1992). To enable ILP systems to learn HPSG grammars it would be necessary to extend refinement operators to allow learning of TFS hierarchies and learning of TFS logic programs. Similarly, learning of Lambek grammars, involves extending refinement operators to the Lambek calculi. Adriaans and de Haas (this volume) can be viewed as an initial effort in this direction.

An additional challenge for ILP is *unsupervised learning* or, in the NLP context, learning from unannotated data. Currently, most ILP systems require annotated training data. However, for tasks such as learning of lexicalist grammars such as HPSG or categorial grammar (CG), such annotated data (in the form of parse trees annotated with HPSG/CG structures) is generally unavailable. Thus, the question of whether it is possible to learn a HPSG/CG lexicon from the HPSG/CG background linguistic theory and a set of grammatical sentences needs investigation. Initial work on unsupervised learning of a CG lexicon is reported in Watkinson and Manandhar (this volume).

For NLP learning tasks where totally unsupervised learning is difficult or infeasible, the best strategy might be to aid the learning process either by 1) providing a small amount of annotated data or 2) starting with a partial but correct theory or 3) using a limited amount of active learning. Combining human input with machine learning may be unavoidable both to shorten the learning time and to make it feasible, see, for instance, Brill (this volume). Use of active learning for information extraction is explored in Thompson and Califf (this volume). Active learning is also employed for learning verb subcategorisation frames in Nedellec (this volume). However, each of these need scaling up in order to be usable in large-scale NLP applications.

NLP querying systems are likely to become very important in the near future for language based querying of web-based information resources. Thus, the application of machine learning techniques for semantic parsing is essential. (Zelle & Mooney, 1996) and Thompson and Califf (this volume) demonstrate the use of ILP techniques for learning semantic parsers. For these tasks, the training examples consists of sentence-meaning pairs. The task of the learner is to learn parse control rules (for a shift reduce parser) and appropriate semantic composition rules. However, further work is essential to scale up these existing approaches.

Concept hierarchies such as WordNet (Fellbaum, 1998) are useful in information extraction tasks. As an information extraction system is deployed it will need to be updated as new domain specific terminology comes into use. Automatic methods for acquisition of domain specific terminology and its automatic integration into a concept hierarchy is necessary since it will not be possible to

[2] By "description logics" we mean knowledge representation languages of the KL-ONE family such as KL-ONE (Brachman & Schmolze, 1985), CLASSIC (Bordiga, Brachman, McGuiness, & Resnick, 1989), LOOM (McGregor, 1988).

collect training data and retrain an IE system once it is deployed. Nedellec (this volume) shows that ILP techniques can be usefully employed for such tasks.

Acknowledgements

Thanks to Stephen Pulman for input to the introductory section and to Stephen Muggleton for help on inverse entailment. Sašo Džeroski is supported by the Slovenian Ministry of Science and Technology.

References

1. Adriaans, P. (1999). *Learning Shallow Context-Free languages under simple distributions.* CSLI-publications, University of Stanford.
2. Bordiga, A., Brachman, R., McGuiness, D., & Resnick, L. (1989). Classic: A structural data model for objects. In *1989 ACM SIGMOD International Conference on Management of Data*, pp. 59–67.
3. Boström, H. (1998). Predicate invention and learning from positive examples only. In *Proc. of the Tenth European Conference on Machine Learning*, pp. 226–237. Springer Verlag.
4. Brachman, R. J., & Schmolze, J. G. (1985). An overview of the kl-one knowledge representation system. *Cognitive Science*, *9*(2), 171–216.
5. Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part of speech tagging. *Computational Linguistics*, 246–253.
6. Buntine, W. (1988). Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, *36*(2), 149–176.
7. Carpenter, B. (1992). *The Logic of Typed Feature Structures.* Cambridge University Press.
8. Cussens, J., & Pulman, S. (2000). Incorporating linguistics constraints into inductive logic programming. In *Proc. of CoNLL-2000 and LLL-2000* Lisbon. Omni Press. To appear.
9. De Raedt, L., & Džeroski, S. (1994). First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, *70*, 375–392.
10. Dehaspe, L., & Forrier, M. (1999). Transformation-based learning meets frequent pattern discovery. In *Language Logic and Learning Workshop* Bled, Slovenia.
11. Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database.* MIT Press, Cambridge, MA.
12. Flach, P. (1992). Logical approaches to machine learning - an overview. *THINK*, *1*(2), 25–36.
13. Flach, P. A., & Kakas, A. C. (Eds.). (2000). *Abduction and Induction: Essays on their Relation and Integration*, Vol. 18 of *Applied Logic Series.* Kluwer, Dordrecht.
14. Hogger, C. (1990). *Essentials of Logic Pogramming.* Clarendon Press, Oxford.
15. Lloyd, J. (1987). *Foundations of Logic Programming* (2nd edition). Springer, Berlin.

16. Manandhar, S., Džeroski, S., & Erjavec, T. (1998). Learning multilingual morphology with CLOG. In Page, D. (Ed.), *Inductive Logic Programming; 8th International Workshop ILP-98, Proceedings*, No. 1446 in Lecture Notes in Artificial Intelligence, pp. 135–144. Springer.
17. McGregor, R. (1988). A deductive pattern matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI88)*, pp. 403–408 Menlo Park, CA.
18. Mooney, R. J., & Califf, M. E. (1995). Induction of first–order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, *3*, 1–24.
19. Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing Journal*, *13*, 245–286.
20. Muggleton, S. (2000). Learning from positive data. *Machine Learning*. Accepted subject to revision.
21. Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, *8*(4), 295–318.
22. Muggleton, S., & Bryant, C. (2000). Theory completion using inverse entailment. In *Proc. of the 10th International Workshop on Inductive Logic Programming (ILP-00)* Berlin. Springer-Verlag. In press.
23. Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proc. Fifth International Conference on Machine Learning*, pp. 339–352 San Mateo, CA. Morgan Kaufmann.
24. Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proc. First Conference on Algorithmic Learning Theory*, pp. 368–381 Tokyo. Ohmsha.
25. Niblett, T. (1988). A study of generalisation in logic programs. In *Proc. Third European Working Session on Learning*, pp. 131–138 London. Pitman.
26. Pereira, F. (1981). Extraposition grammars. *Computational Linguistics*, *7*, 243–256.
27. Pierce, C. (1958). *Collected Papers of Charles Sanders Pierce*. Harvard University Press. Edited by C. Hartsthorne, P. Weiss and A. Burks.
28. Plotkin, G. (1969). A note on inductive generalization. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence 5*, pp. 153–163 Edinburgh. Edinburgh University Press.
29. Pollard, C., & Sag, I. A. (1987). *Information-Based Syntax and Semantics: Volume 1 Fundamentals*, Vol. 13 of *Lecture Notes*. Center for the Study of Language and Information, Stanford, CA.
30. Pollard, C., & Sag, I. A. (1994). *Head-driven Phrase Structure Grammar*. Chicago: University of Chicago Press and Stanford: CSLI Publications.
31. Quinlan, J. R. (1996). Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, *5*, 139–161.
32. Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, *5*(3), 239–266.
33. Robinson, J. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, *12*(1), 23–41.

34. Rouveirol, C. (1990). Saturation: Postponing choices when inverting resolution. In *Proceedings of the Ninth European Conference on Artificial Intelligence*. Pitman.
35. Shapiro, E. (1983). *Algorithmic Program Debugging*. MIT Press, Cambridge, MA.
36. Smolka, G. (1992). Feature constraint logics for unification grammars. *Journal of Logic Programming*, *12*, 51–87.
37. Zelle, J. M., & Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence* Portland, OR.