Inductive Logic Programming

Techniques and Applications

Nada Lavrač and Sašo Džeroski

Jožef Stefan Institute
Ljubljana, Slovenia

N. Lavrač and S. Džeroski.
Inductive Logic Programming: Techniques and Applications.
Ellis Horwood, New York, 1994.

Contents

IV Applications of ILP 197

Foreword

This book is about inductive logic programming (ILP), which is a new technology combining principles of inductive machine learning with the representation of logic programming. The new technology aims at inducing general rules starting from specific observations and background knowledge.

Let me explain why inductive logic programming is currently regarded as (one of) the most important recent development(s) in machine learning research. In my view, ILP = I $\cap$ LP, i.e., ILP is the intersection of techniques and interests in induction and logic programming. This makes inductive logic programming more powerful than traditional techniques that learn from examples, namely, inductive logic programming uses an expressive first-order logic framework instead of the traditional attribute-value framework, and facilitates the use of background knowledge. The first point is important because many domains of expertise cannot be formulated in an attribute-value framework. The second point is also significant because background knowledge is the key to success in nearly all applications of artificial intelligence. At the same time, inductive logic programming has room for both theory and practice. Inductive logic programming has a strong theoretical basis as it inherited many results from logic programming and computational learning theory, and adapted (and continues to adapt) these to fulfill the needs of a new discipline. Inductive logic programming has also very impressive applications in scientific discovery, knowledge synthesis and logic programming. To mention an important achievement: Stephen Muggleton has recently produced — using general purpose inductive logic programming systems — new scientific knowledge in drug design and protein engineering, which is now published in scientific journals [King et al. 1992, Muggleton et al. 1992a].

The authors of this book, Nada Lavrač and Sašo Džeroski, are members of the AI Laboratory of the Jožef Stefan Institute (Ljubljana, Slovenia) directed by Ivan Bratko. The laboratory and its members have an outstanding reputation in applying both machine learning and logic programming to real-life knowledge engineering problems. Nada Lavrač has already co-authored books in both areas: *Prolog through Examples: A Practical Programming Guide* (by Igor Kononenko and Nada Lavrač, Sigma Press, 1988) and *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems* (by Ivan Bratko, Igor Mozetič and Nada Lavrač, The MIT Press, 1989). It is therefore no surprise that Nada Lavrač, now with Sašo Džeroski, combines her two interests in this book.

The book is written from an engineering perspective with applications in mind. Indeed, after an introduction to and a survey of inductive logic programming in Part I, the authors focus on the empirical setting for inductive logic programming in Part II. The empirical setting is currently the one best suited for applications as it aims at synthesizing knowledge from potentially large sets of observations. In this second part, the authors present the first important contribution: the idea that inductive logic programming can benefit from the well-understood traditional induction techniques such as top-down induction of decision trees. This idea forms the basis of the system LINUS (and its extension DINUS) which inherits its efficiency from the traditional learning algorithms. The efficiency of LINUS is very relevant, not only to the users, but also to computational learning theory. Indeed, it was recently discovered that some of the assumptions of LINUS/DINUS are closely related to the class of polynomially learnable predicates in computation learning theory (PAC-learnability). Another significant contribution of LINUS is that it is the first inductive logic programming system able to handle numerical data. In Part III, the authors tackle another key problem when doing real-life applications: imperfect data. To this aim, they investigate and evaluate new heuristics for inductive logic programming, and show in this way that inductive logic programming is ready to handle imperfect data and therefore also real-life applications. Real-life applications are the subject of Part IV, which contains some impressive results in medicine (the award-winning application in rheumatology at the Third Scandinavian Conference on Artificial Intelligence, SCAI-91), in engineering (finite element mesh design), and qualitative modeling.

Besides giving a good introduction to the field of inductive logic programming, this book also shows what (knowledge) engineers can achieve with this new technology. Furthermore, it is easily readable and accessible. Therefore, I believe it will be of interest to a wide audience, including graduates, researchers and practitioners of artificial intelligence and computer science; in particular, those interested in machine learning, knowledge engineering, knowledge discovery in databases and logic programming.

Sint-Amandsberg, April 1993 *Luc De Raedt*

Preface

Inductive logic programming (ILP) is a research area at the intersection of machine learning and logic programming. It aims at a formal framework as well as practical algorithms for inductive learning of relational descriptions in the form of logic programs. From logic programming, ILP has inherited its sound theoretical basis, and from machine learning, an experimental approach and orientation towards practical applications. ILP has already shown its application potential in the following areas: knowledge acquisition, inductive program synthesis, inductive data engineering, and knowledge discovery in databases.

This book is an introduction to this exciting new field of research. It should be of interest to the following readership: knowledge engineers concerned with the automatic synthesis of knowledge bases for expert systems; software engineers who could profit from inductive programming tools; researchers in system development and database methodology, interested in techniques for knowledge discovery in databases and inductive data engineering; and researchers and graduate students in artificial intelligence, machine learning, logic programming, software engineering and database methodology.

As ILP is a relatively young field, the ILP literature consists mainly of conference and workshop proceedings. A book collection of papers [Muggleton 1992a], covering the whole field of ILP, is also available, as well as a book on theory revision in the context of interactive ILP [De Raedt 1992]. Interactive ILP, one of the two major subfields of ILP, is closely related to program debugging and theory revision where a small number of examples is available. The second major subfield of ILP, called empirical ILP, places an emphasis on the extraction of regularities from a large number of possibly erroneous examples. The spirit of empirical ILP is much closer to the existing successful learning systems, and is therefore more suitable for practical applications than interactive ILP. This book extensively covers empirical ILP techniques and applications.

The book is divided into four parts. Part I is an introduction to the field of ILP. Part II describes in detail empirical ILP techniques and systems. Part III is concerned with the problem of handling imperfect data in ILP, whereas Part IV gives an overview of empirical ILP applications.

Part I comprises Chapters 1 to 3. Chapter 1 touches upon the topics covered by the book. Chapter 2 introduces the basic concepts and terminology of logic programming, deductive databases and inductive logic programming. The basic generalization and specialization techniques used in ILP are described in Chapter 3.

Chapters 4 to 7 constitute Part II. Chapter 4 gives an overview of several empirical ILP systems with an emphasis on FOIL [Quinlan 1990], which has been the basis for much of our work. The central chapter of this part, Chapter 5, gives a detailed description of the ILP system LINUS. This includes a description of the LINUS environment, the propositional learning systems incorporated into LINUS, the transformation algorithm, a specification of the hypothesis language and a complexity analysis. It is also shown how LINUS can be extended to generate hypotheses in a more expressive hypothesis language. Chapter 6 describes several experiments in learning relations with LINUS. In Chapter 7, the framework of refinement graphs is used to compare the hypothesis languages and the search complexity of LINUS and FOIL.

Part III starts with Chapter 8, which first describes the problem of handling imperfect data in attribute-value and ILP systems and then gives an overview of techniques and heuristics used in handling imperfect data. Chapter 9 presents the ILP system mFOIL, which incorporates techniques for handling imperfect data from attribute-value systems into the FOIL framework. Its language bias (search space) and search bias (heuristics, search strategy and stopping criteria) are described in detail. An experimental comparison of LINUS, FOIL and mFOIL in a chess endgame domain with artificially introduced errors (noise) in the examples is made in Chapter 10.

Although experiments in artificial domains under controlled amounts of noise reveal important performance aspects of ILP systems, the ultimate test is their application to real-life problems. Part IV gives a detailed description of the applications of LINUS and mFOIL to three different practical learning problems. Comparisons are also made with FOIL and GOLEM [Muggleton and Feng 1990]. Chapter 11 describes the application of LINUS to the problem of learning rules for medical diagnosis, where examples and specialist background knowledge were provided by a medical expert. Chapter 12 compares the performance of mFOIL, FOIL and GOLEM on the task of inducing rules for determining the appropriate resolution of a finite element mesh. Chapter 13 describes the induction of qualitative models from example behaviors and describes the application of mFOIL to this problem, making also a comparison to other systems. Finally, Chapter 14 concludes with an overview of several applications of GOLEM to illustrate the potential of ILP for practical applications.

Acknowledgements

The book is a result of several years of research. Most of it was done at the Artificial Intelligence Laboratory of the Computer Science Department, Jožef Stefan Institute in Ljubljana. We would like to thank Ivan Bratko, the head of the Artificial Intelligence Laboratory, for guiding our research interests towards challenging research topics in AI. Our thanks goes also to Marjan Špegel, the head of the Computer Science Department, for his support during all these years of research.

Our research was based on the tradition of the Artificial Intelligence Laboratory in the areas of machine learning and qualitative modeling. The KARDIO project [Bratko et al. 1989] has been a motivation and a source of ideas for our work. The special purpose learning system QuMAS [Mozetič 1987], which was used in KARDIO for reconstructing a qualitative model of the heart, was based on the idea of transforming a relation learning problem into propositional form. This idea has been further developed into our general ILP environment called LINUS [Lavrač et al. 1991a]. This approach has provided us with a possibility to use a variety of learning techniques, including mechanisms for handling imperfect data, developed in propositional learning systems. Besides using the noise-handling techniques within the LINUS framework, we have also adapted them for direct use in our ILP learner mFOIL [Džeroski and Bratko 1992a], an extension of the FOIL [Quinlan 1990] system. LINUS and mFOIL transcend the approaches of QuMAS and

FOIL mainly by the use of more sophisticated mechanisms for handling imperfect data, which is one of the main topics of this book. We are grateful to Igor Mozetič for his support in the development of LINUS, and to Bojan Cestnik who contributed his expertise and advice in the selection of the noise-handling mechanisms described in this book.

We wish to thank Peter Flach, Igor Kononenko, Stephen Muggleton, Tanja Urbančič and, in particular, Luc De Raedt who contributed significantly to this book by providing helpful comments and advice. We would further like to thank Marko Grobelnik and Dunja Mladenič, who contributed to the development of LINUS and a VAX/VMS version of ASSISTANT incorporated into LINUS, respectively; Bruno Stiglic and Robert Trappl for their support and interest in this work; Ross Quinlan for making his FOIL system available for experimental comparisons; Michael Bain, Ivan Bratko, Bojan Dolšak, Vladimir Pirnat and Ross Quinlan for making their experimental data available; Bojan Orel for his TEXpertise used in the preparation of this book; and our colleagues in the Artificial Intelligence Laboratory for their contributions to the views expressed in this book.

Our research has been continuously supported by the Slovenian Ministry of Science and Technology. The research was also part of the ESPRIT II Basic Research Action no. 3059 ECOLES and the ESPRIT III Basic Research Project no. 6020 Inductive Logic Programming. The Slovenian Ministry of Science and Technology supported Nada Lavrač during her three-month visit to the University of Illinois at Urbana-Champaign, Urbana, Illinois in 1985, and to the George Mason University, Fairfax, Virginia in 1988. The Belgian State-Science Policy Office grant enabled her a six-month stay at the Katholieke Universiteit Leuven, Belgium in 1991–1992. The Slovenian Ministry of Science and Technology and the British Council supported Sašo Džeroski during his eight-month stay at the Turing Institute, Glasgow, UK, also in 1991–1992. We are grateful to Ryszard Michalski (George Mason University), to Maurice Bruynooghe and Luc De Raedt (Katholieke Universiteit Leuven), and to Stephen Muggleton (The Turing Institute) for their support of our work in the stimulating research environments of their institutions.

The inductive logic programming community, gathered in the ESPRIT III project Inductive Logic Programming, provided a stimulating environment for the exchange of ideas and results in the field; this book is a contribution to the body of knowledge developed within this community. We are grateful to Stephen Muggleton and Luc De Raedt for their contributions to ILP research and for their organizational efforts which have made this research group so lively.

Copyright acknowledgements

The following material was taken from the book *Inductive Logic Programming*, edited by Stephen Muggleton, with the kind permission of Academic Press: Figure 5.1, Figure 6.6, Sections 7.4, 7.5, 7.6 and Figure 14.9.

A major part of Chapter 11 appears in *Applied Artificial Intelligence* 7: 273–293, 1993, under the title 'The utility of background knowledge in learning medical diagnostic rules'. It is republished with the permission of the publisher Taylor & Francis.

Thanks to Bojan Dolšak for Figure 12.1, Dunja Mladenić and Aram Karalič for Figure 14.3, and Ashwin Srinivasan for Figure 14.1.

Ljubljana, March 1993 *Nada Lavrač* and *Sašo Džeroski*

Part I

Introduction to ILP

1

Introduction

Inductive logic programming is a research area at the intersection of machine learning and logic programming. It aims at a formal framework and practical algorithms for inductively learning logic programs from examples. This chapter gives an introduction to inductive logic programming. It first acquaints the reader with the basics of inductive concept learning and discusses the role of background knowledge and language bias. It then gives a brief overview of inductive logic programming, addresses the problem of handling imperfect data and discusses the potential of inductive logic programming for practical applications.

1.1 Inductive concept learning

The current interest in *machine learning*, a subfield of artificial intelligence, is twofold. On the one hand, although there is no consensus on the nature of intelligence, it is agreed that the capability of learning is crucial for any intelligent behavior. Machine learning is thus justified from the scientific point of view as part of cognitive science. On the other hand, machine learning techniques can be successfully used in knowledge acquisition tools, which is a justification from the engineering point of view. Namely, it is generally accepted that the main problem in building expert systems (to which the success of artificial intelligence in industry is largely due) is the acquisition of knowledge.

As part of computer science/engineering, the goal of machine learning is to develop methods, techniques and tools for building intelligent *learning machines*, e.g., such learning programs which are able to change themselves in order to perform *better* at a given task. To perform ‘better’ means, for example, to perform more efficiently and/or more accurately. ‘Better’ can also denote the learner’s ability to handle a broader class of problems.

In a broad sense, machine learning paradigms [Carbonell 1989] include inductive learning, analytic (deductive) learning, learning with genetic algorithms, and connectionist learning (learning with neural nets). Furthermore, several learning paradigms can be integrated within a single multistrategy learning system [Michalski and Tecuci 1991]. In a narrower sense, Michie's strong criterion [Michie 1988] defines learning as the ability to acquire new knowledge by requiring the result of learning to be understandable by humans. In this sense, connectionist systems are not considered to be learning systems.

This book focuses on *inductive learning*. It is generally known that induction means reasoning from specific to general. In the case of *inductive learning from examples*, the learner is given some examples from which general rules or a theory underlying the examples are derived. Inductive learning has been successfully applied to a variety of classification and prediction problems, such as the diagnosis of a patient or a plant disease, or the prediction of mechanical properties of steel on the basis of its chemical characteristics. These problems can be formulated as tasks of learning concepts from examples, referred to as *inductive concept learning*, where classification rules for a specific concept must be induced from instances (and non-instances) of that concept.

To define the problem of inductive concept learning one first needs to define a *concept*. If $\mathcal{U}$ is a universal set of objects (or observations), a concept $\mathcal{C}$ can be formalized as a subset of objects in $\mathcal{U}$: $\mathcal{C} \subseteq \mathcal{U}$. For example, $\mathcal{U}$ may be the set of all patients in a register, and $\mathcal{C} \subseteq \mathcal{U}$ the set of all patients having a particular disease. Similarly, the concept 'arch' is the set of all arches in some universe of objects $\mathcal{U}$. To learn a concept $\mathcal{C}$ means to learn to recognize objects in $\mathcal{C}$ [Bratko 1989], i.e., to be able to tell whether $x \in \mathcal{C}$ for each $x \in \mathcal{U}$.

Describing objects and concepts

In machine learning, a formal language for describing objects and concepts needs to be selected. Objects are described in an *object description language*. Concepts can be described in the same language or in a separate *concept description language*.

In an *attribute-value* object description language objects are described by a fixed repertoire of features, also called *attributes*, each of them taking a value from a corresponding prespecified value set. In the game of poker, for instance, objects are poker cards and the selected two attributes for describing cards are the suit and the rank of a card. The set of values for the attribute *Suit* is {*hearts, spades, clubs, diamonds*}, and the set of values for *Rank* is {*7, 8, 9, 10, j, q, k, a* }, where *j*, *q*, *k*, *a* stand for *jack*, *queen*, *king* and *ace*, respectively.

An individual card can be described in an attribute-value language by a conjunctive expression, e.g., $[Suit = diamonds] \wedge [Rank = 7]$. In a different attribute-value language, this same object can be described by a tuple $< diamonds, 7 >$. In the first-order language of *Horn clauses*, used in logic programming, a card can be described by a *ground fact*, e.g., $card(diamonds, 7)$. In this case, *card* is a predicate name, and the values of the two arguments *Rank* and *Suit* of the predicate $card/2$ are the constants *diamonds* and 7.

A pair of cards can be described in an object description language comprising four attributes: $Suit_1$ and $Rank_1$ describing one card, and $Suit_2$ and $Rank_2$ describing the other card. In an attribute-value language, a pair of two cards $< diamonds, 7 >$ and $< hearts, 7 >$ can either be described by a 4-tuple $< diamonds, 7, hearts, 7 >$ or, alternatively, by a conjunctive expression $[Suit_1 = diamonds] \wedge [Rank_1 = 7] \wedge [Suit_2 = hearts] \wedge [Rank_2 = 7]$. In the language of Horn clauses, two alternative descriptions are $pair(diamonds, 7, hearts, 7)$ and $pair(card(diamonds, 7), card(hearts, 7))$, where *pair* is a predicate name and *card* is a function name.

Concepts can be described extensionally or intensionally. A concept is described *extensionally* by listing the descriptions of all of its instances. For example, the concept *pair* in poker can be extensionally described by the set of all pairs of cards which have the same rank, e.g., by the following set of 4-tuples: $pair = \{<hearts, 7, spades, 7>\ ,\ < hearts, 7, clubs, 7 >, \ldots, < diamonds, a, clubs, a >\}$.

Extensional concept descriptions can be undesirable for a number of reasons, including the fact that a concept may contain an infinite number of instances. Consequently, it is preferable to describe concepts *intensionally*, that is, in a separate concept description language which allows for more compact and concise concept descriptions, e.g., in the form of rules. The following rule is, for example, an intensional description of the concept *pair* in an attribute-value concept description language:

$$\begin{array}{lll} \mathit{pair} & \textbf{if} & [\mathit{Rank}_1 = 7] \wedge [\mathit{Rank}_2 = 7] \bigvee \\ & & [\mathit{Rank}_1 = 8] \wedge [\mathit{Rank}_2 = 8] \bigvee \\ & & \ldots \\ & & [\mathit{Rank}_1 = a] \wedge [\mathit{Rank}_2 = a] \end{array}$$

In this attribute-value language, the antecedent of an *if-then* rule is a disjunction of conditions, each condition being a conjunction of attribute-value pairs (expressions of the form $\mathit{Attribute} = \mathit{value}$). As the attributes Suit_1 and Suit_2 do not appear in the above rule, they can have any value in the appropriate range.

In case the concept description language is more powerful and allows for the use of variables, the concept *pair* can be described in a more compact way:

$$\mathit{pair} \quad \textbf{if} \quad \mathit{Rank}_1 = \mathit{Rank}_2$$

In the language of Horn clauses, the above concept description takes the following form:

$$\mathit{pair}(\mathit{Suit}_1, \mathit{Rank}_1, \mathit{Suit}_2, \mathit{Rank}_2) \quad \leftarrow \quad \mathit{Rank}_1 = \mathit{Rank}_2$$

If the description language allows for the use of function symbols for describing structured terms, the *atom* $\mathit{pair}(\mathit{Suit}_1, \mathit{Rank}_1, \mathit{Suit}_2, \mathit{Rank}_2)$ in the above clause could be replaced by the atom with structured terms in the arguments $\mathit{pair}(\mathit{card}(\mathit{Suit}_1, \mathit{Rank}_1), \mathit{card}(\mathit{Suit}_2, \mathit{Rank}_2))$, yielding:

$$\mathit{pair}(\mathit{card}(\mathit{Suit}_1, \mathit{Rank}_1), \mathit{card}(\mathit{Suit}_2, \mathit{Rank}_2)) \leftarrow \mathit{Rank}_1 = \mathit{Rank}_2$$

Definition of inductive concept learning

Having selected description languages for objects and concepts, a procedure is needed that will establish whether a given object belongs to a given concept, i.e., whether the description of the object *satisfies* the description of the concept. If it does, we say that the concept description *covers* the object description, or that the object description *is covered* by the concept description.

Let us, from now on, use the term *fact* for an object description and *hypothesis* for an intensional concept description to be learned. An *example* e for learning a concept $\mathcal{C}$ is a labeled fact, with label $\oplus$, if the object is an instance of the concept $\mathcal{C}$, and label $\ominus$ otherwise (i.e., if it is a non-instance of the concept). Let $\mathcal{E}$ denote a set of examples. The facts from $\mathcal{E}$ labeled $\oplus$ form a set of *positive examples* $\mathcal{E}^+$ and those labeled $\ominus$ constitute a set of *negative examples* $\mathcal{E}^-$ for the concept $\mathcal{C}$. We will take the liberty of assuming that facts in $\mathcal{E}^+$ and $\mathcal{E}^-$ are not labeled explicitly, but rather implicitly by their membership in one of the two sets; as a consequence, we will sometimes use the term example instead of fact and vice versa. For example, when learning the concept *pair* in poker, $pair(card(diamonds, 7), card(spades, 7))$ is a positive example, and $pair(card(diamonds, 7), card(spades, 10))$ is a negative example.

In the so-called *single concept learning*, a concept description is induced from facts labeled $\oplus$ and $\ominus$, i.e., from examples and counter-examples of a single concept $\mathcal{C}$. In *multiple concept learning*, labels denote different concept names representing different *classes*. In this case, the set of training examples $\mathcal{E}$ can be divided into subsets of positive and negative examples for each individual concept (class).

The problem of learning a single concept $\mathcal{C}$ from examples can now be stated as follows:

Inductive concept learning Given a set $\mathcal{E}$ of positive and negative examples of a concept $\mathcal{C}$, find a hypothesis $\mathcal{H}$, expressed in a given concept description language $\mathcal{L}$, such that:

- every positive example $e \in \mathcal{E}^+$ is covered by $\mathcal{H}$,
- no negative example $e \in \mathcal{E}^-$ is covered by $\mathcal{H}$.

To test the coverage, a function

$$covers(\mathcal{H}, e) \tag{1.1}$$

can be introduced, which returns the value *true* if e is covered by $\mathcal{H}$, and *false* otherwise. The implementation of this function depends on the object and concept description languages. For instance, if e is

$$[Suit_1 = diamonds] \wedge [Rank_1 = 7] \wedge [Suit_2 = hearts] \wedge [Rank_2 = 7]$$

and $\mathcal{H}$ is

$$\begin{array}{lll} pair & \textbf{if} & [Rank_1 = 7] \wedge [Rank_2 = 7] \bigvee \\ & & [Rank_1 = 8] \wedge [Rank_2 = 8] \bigvee \\ & & \ldots \\ & & [Rank_1 = a] \wedge [Rank_2 = a] \end{array}$$

the *covers* function simply tests whether e satisfies any of the disjuncts in $\mathcal{H}$. In logic programming, where a hypothesis is a set of program clauses and an example is a ground fact, a forward chaining procedure [Bry 1990] or a form of SLD-resolution rule of inference [Lloyd 1987] can be used to check whether e is entailed by $\mathcal{H}$. For example, $pair(card(diamonds, 7), card(hearts, 7))$ is entailed by

$$pair(card(Suit_1, Rank_1), card(Suit_2, Rank_2)) \leftarrow Rank_1 = Rank_2$$

The function $covers(\mathcal{H}, e)$ can be extended to work for sets of examples in the following way:

$$covers(\mathcal{H}, \mathcal{E}) = \{e \in \mathcal{E} \mid covers(\mathcal{H}, e) = true\} \qquad (1.2)$$

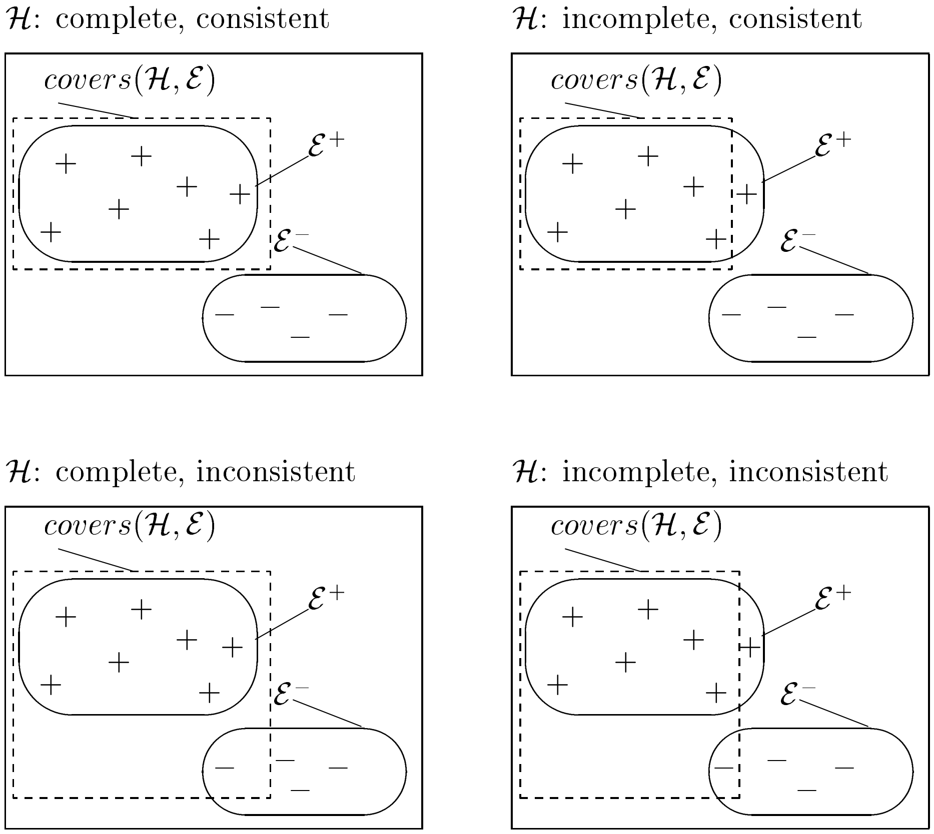
returning the set of facts from $\mathcal{E}$ which are covered by $\mathcal{H}$.

In the problem statement of inductive concept learning it is required that the hypothesis $\mathcal{H}$ covers all the positive examples and none of the negative ones. In this case we say that the hypothesis $\mathcal{H}$ is complete and consistent with respect to the examples $\mathcal{E}$. A hypothesis $\mathcal{H}$ is *complete* with respect to examples $\mathcal{E}$ if it covers all the positive examples, i.e., if $covers(\mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$. A hypothesis $\mathcal{H}$ is consistent with respect to examples $\mathcal{E}$ if it covers none of the negative examples, i.e., if $covers(\mathcal{H}, \mathcal{E}^-) = \emptyset$. Figure 1.1 shows different possible situations regarding the completeness/consistency of a hypothesis.

A similar formalization of the problem of concept learning is given in De Raedt [1992], where the completeness and consistency criteria are together referred to as a *quality criterion.*

Criteria of success

The above stated definition of inductive concept learning requires that $\mathcal{C}$ and $\mathcal{H}$ agree on all examples from $\mathcal{E}$. There is, however, no a priori guarantee that $\mathcal{H}$ will correspond to $\mathcal{C}$ on other objects as well [Bratko 1989]. Since one of the major aims of learning is to classify


$\mathcal{H}$: complete, consistent

$\mathcal{H}$: incomplete, consistent

$\mathcal{H}$: complete, inconsistent

$\mathcal{H}$: incomplete, inconsistent

Figure 1.1: Completeness and consistency of a hypothesis.

unseen objects with respect to $\mathcal{C}$, one can view the induced hypothesis $\mathcal{H}$ also as a *classifier* of new objects. Therefore, the accuracy of classifying unseen objects is the main criterion of the success of the learning system.

Thus, important performance criteria of learning include the following:

Classification accuracy The classification accuracy of $\mathcal{H}$ is measured as the percentage of objects correctly classified by the hypothesis.

Transparency The transparency of $\mathcal{H}$ denotes the extent to which it is understandable to humans. A possible measure is the *length* of the hypothesis, expressed by the total number of conditions in an induced rule set $\mathcal{H}$. Another possibility is to measure the number of bits used in the encoding of the description.

Statistical significance Statistical significance tests can be used to evaluate whether hypothesis $\mathcal{H}$ represents a genuine regularity in the training examples and not a regularity which is due to chance.

Information content The information content or the relative information score of classifier $\mathcal{H}$ [Kononenko and Bratko 1991] scales the classifier's performance according to the difficulty of the classification problem. It is measured by taking into account the prior probability of the concept $\mathcal{C}$ to be learned, i.e., the percentage of objects in $\mathcal{U}$ that belong to $\mathcal{C}$. For example, in medical diagnosis a correct classification into a more probable diagnosis provides less information than a correct classification into a rare diagnosis, which is only represented by a few examples.

1.2 Background knowledge

Concept learning can be viewed as searching the space of concept descriptions [Mitchell 1982]. If the learner has no prior knowledge about the learning problem it learns exclusively from examples. However, difficult learning problems typically require a substantial body of prior knowledge. We will refer to declarative prior knowledge as *background knowledge*. Using background knowledge, the learner might express the generalization of examples in a more natural and concise manner.

The hypothesis language $\mathcal{L}$ and, indirectly, the background knowledge $\mathcal{B}$ determine the search space of possible concept descriptions, also called the *hypothesis space*. Therefore, the background knowledge has to be included in the problem statement of inductive concept learning.

Inductive concept learning with background knowledge Given a set of training examples $\mathcal{E}$ and background knowledge $\mathcal{B}$, find a hypothesis $\mathcal{H}$, expressed in some concept description language $\mathcal{L}$, such that $\mathcal{H}$ is complete and consistent with respect to the background knowledge $\mathcal{B}$ and the examples $\mathcal{E}$.

To make this formulation of inductive concept learning operational, the function *covers* introduced in equations (1.1) and (1.2) must be extended to take into account the background knowledge $\mathcal{B}$:

$$covers(\mathcal{B}, \mathcal{H}, e) = covers(\mathcal{B} \cup \mathcal{H}, e) \tag{1.3}$$

$$covers(\mathcal{B}, \mathcal{H}, \mathcal{E}) = covers(\mathcal{B} \cup \mathcal{H}, \mathcal{E}) \tag{1.4}$$

The completeness and consistency requirements are also modified to read as stated below.

Completeness A hypothesis $\mathcal{H}$ is *complete* with respect to background knowledge $\mathcal{B}$ and examples $\mathcal{E}$ if all the positive examples are covered, i.e., if $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$.

Consistency A hypothesis $\mathcal{H}$ is consistent with respect to background knowledge $\mathcal{B}$ and examples $\mathcal{E}$ if no negative example is covered, i.e., if $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^-) = \emptyset$.

1.3 Language bias

Any mechanism employed by a learning system to constrain the search for hypotheses is named *bias* [Utgoff and Mitchell 1982]. *Declarative bias* denotes explicit, user-specified bias which can preferably be formulated as a modifiable parameter of the system. Bias can either determine how the hypothesis space is searched (*search bias*) or determine the hypothesis space itself (*language bias*).

This section focuses on language bias. By selecting a stronger language bias (a less expressive hypothesis language) the search space becomes smaller and learning more efficient; however, this may prevent the system from finding a solution which is not contained in the less expressive language. This expressiveness/tractability tradeoff underlies much of the current research in inductive learning.

When considering language bias, the *expressiveness* of a hypothesis language becomes a crucial notion. The term *expressiveness* captures two different dimensions. The first dimension is the expressive power of a formalism; in its standard meaning, *stronger expressive power* means that there are concepts that can be represented in the stronger formalism which can not be represented in the weaker formalism. The other dimension is the *transparency* (or *length*) of the concept representation. For example, it is possible to state that two Boolean-valued attributes have the same value in both an attribute-value language and a first-order language. But, whereas in the former we have to say $([A = false] \wedge [B = false]) \bigvee ([A = true] \wedge [B = true])$, we can simply say $A = B$ in the latter.

Various logical formalisms have been used in inductive learning systems to represent examples and concept descriptions. These formalisms are similar to the formalisms for representing knowledge in general, ranging from propositional logic to full first-order predicate calculus. Within this scope one most frequently distinguishes between systems which learn attribute descriptions, and systems which learn first-order relational descriptions.

Several widely known inductive learning algorithms, such as ID3 [Quinlan 1986] and AQ [Michalski 1983, Michalski et al. 1986] use an attribute-value language to represent objects and concepts, and are thus called *attribute-value learners*. They are also called *propositional learners* since an attribute-value language is in its expressive power equivalent to propositional calculus. In both ID3 and AQ, objects are described in terms of their global features, i.e., by values of a fixed collection of *attributes*. To represent concepts, decision trees are used in ID3 and if-then rules in AQ. The two main limitations of propositional learners are the limited expressiveness of the representational formalism and their limited capability of taking into account the available background knowledge.

Another class of learning systems, called *relation learners*, induce descriptions of relations (definitions of predicates). In these systems, objects can be described structurally, i.e., in terms of their components and relations among the components. The given relations constitute the background knowledge. In relation learners, the languages used to represent examples, background knowledge and concept descriptions are typically subsets of first-order logic. In particular, the language of *logic programs* [Lloyd 1987] provides sufficient expressiveness for solving a significant number of relation learning problems. Learners that induce hypotheses in the form of logic programs are called *inductive logic programming* systems.

1.4 Inductive logic programming

Background knowledge plays a central role in relation learning, where the task is to define, from given examples, an unknown relation (i.e., the target predicate) in terms of (itself and) known relations from the background knowledge. If the hypothesis language of a relation learner is the language of logic programs, then learning is, in fact, logic program synthesis and has recently been named *inductive logic programming* (ILP) [Muggleton 1991, 1992a]. In ILP systems, the training examples, the background knowledge and the induced hypotheses are all expressed in a logic program form, with additional restrictions imposed on each of the three languages. For example, training examples are typically represented as ground facts of the target predicate, and most often background knowledge is restricted to be of the same form.

One of the early systems that made use of relational background knowledge in the process of learning structural descriptions was INDUCE [Michalski 1980]. The work on model inference by Shapiro [1983], Plotkin's work on inductive generalization [Plotkin 1969] and the work of Sammut and Banerji [1986] have inspired most of the current efforts in the area of inductive logic programming.

An example ILP problem

Let us illustrate the ILP task on a simple problem of learning family relations.

Example 1.1 (An ILP problem)
The task is to define the target 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 are given in Table 1.1. There are two positive and two negative examples of the target relation.

In the hypothesis language of Horn clauses it is possible to formulate the following definition of the target relation:

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

This definition is consistent and complete with respect to the background knowledge and the training examples. Depending on the background knowledge, the language $\mathcal{L}$ and the complexity of the target

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

Table 1.1: A simple ILP problem: learning the *daughter* relation.

concept, the target predicate definition may consist of a set of clauses, such as:

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

■

Dimensions of ILP

Inductive logic programming systems (as well as other inductive learning systems) can be divided along several dimensions. First of all, they can learn either a single concept or multiple concepts (predicates). Second, they may require all the training examples to be given before the learning process (batch learners) or may accept examples one by one (incremental learners). Third, during the learning process, a learner may rely on an oracle to verify the validity of generalizations and/or classify examples generated by the learner. The learner is called interactive in this case and non-interactive otherwise. Finally, a learner may try to learn a concept from scratch or can accept an initial hypothesis (theory) which is then revised in the learning process. The latter systems are called theory revisors.

Although these dimensions are in principle independent, existing ILP systems are situated at two ends of the spectrum. At one end are batch non-interactive systems that learn single predicates from scratch, while at the other are interactive and incremental theory revisors that learn multiple predicates. Following De Raedt [1992] we call the first type of ILP systems *empirical ILP systems* and the second type *interactive ILP systems.* The second type of systems can alternatively be referred to as *incremental ILP systems.*

Interactive ILP systems include MIS [Shapiro 1983], MARVIN [Sammut and Banerji 1986], CLINT [De Raedt and Bruynooghe 1989, De Raedt and Bruynooghe 1992a] (described also in De Raedt [1992]), CIGOL [Muggleton and Buntine 1988] and other approaches based on inverting resolution [Rouveirol 1991, Wirth 1988, 1989]. Interactive ILP systems typically learn definitions of multiple predicates from a small set of examples and queries to the user. On the other hand, empirical ILP systems typically learn a definition of a single predicate from a large collection of examples. This class of ILP systems includes FOIL [Quinlan 1990], mFOIL [Džeroski 1991], GOLEM [Muggleton and Feng 1990], LINUS [Lavrač et al. 1991a], MARKUS [Grobelnik 1992], etc.

This book is mainly concerned with empirical ILP systems which already show their potential for practical applications. Part I of the book gives an overview of empirical ILP systems, with an emphasis on LINUS and FOIL.

1.5 Imperfect data

The definition of the inductive concept learning task from Section 1.1 requires that $\mathcal{C}$ and $\mathcal{H}$ agree on all examples from $\mathcal{E}$. However, in practice it may happen that the object descriptions and their classifications (labels) presented to an inductive learning system contain various kinds of errors, either random or systematic. In such cases, we say that the learner has to deal with *imperfect data.* In this case, it may be advantageous not to insist that $\mathcal{C}$ and $\mathcal{H}$ agree on all examples from $\mathcal{E}$. A very desirable property of an inductive learning system is then the ability to avoid the effects of imperfect data by distinguishing between genuine regularities in the examples and regularities due to chance or error.

When learning definitions of relations, the following kinds of data imperfection are usually encountered [Lavrač and Džeroski 1992b]:

- noise, i.e., random errors in the training examples and background knowledge,
- insufficiently covered example space, i.e., too sparse training ex-

amples from which it is difficult to reliably detect correlations,

- inexactness, i.e., inappropriate or insufficient description language which does not contain an exact description of the target concept, and
- missing values in the training examples.

Learning systems usually have a single mechanism for dealing with the first three types of imperfect data. Such mechanisms, often called *noise-handling mechanisms*, are typically designed to prevent the induced hypothesis $\mathcal{H}$ from *overfitting* the data set $\mathcal{E}$, i.e., to avoid concept descriptions $\mathcal{H}$ which agree with $\mathcal{C}$ perfectly on the training examples, but poorly on unseen examples. Accordingly, the completeness and consistency criteria in the definition of the inductive learning task need to be relaxed and replaced by a more general quality criterion which would allow the hypothesis to misclassify some of the training examples. Missing values are usually handled by a separate mechanism.

Successors of ID3 and AQ15, e.g., ASSISTANT [Cestnik et al. 1987], CN2 [Clark and Niblett 1989, Clark and Boswell 1991] and C4.5 [Quinlan 1993] have been successfully applied to a wide range of real-life domains and include several techniques for handling imperfect data. These include pruning (pre-pruning and post-pruning) of decision trees and the use of significance tests in rule truncation. Recently, improved probability estimates [Cestnik 1990] have been used to enhance the techniques used in tree pruning [Cestnik and Bratko 1991] and rule truncation [Clark and Boswell 1991, Džeroski et al. 1992a].

Until recently, relation learning systems had been mostly of interactive and incremental nature, relying on a small set of examples. The examples had been assumed correct, as well as the answers provided by the oracle. Thus, ILP systems did not address the issue of data imperfection. However, with the advent of empirical ILP systems, which reached the stage of efficiency where application to practical domains is feasible, the interest in handling imperfect data in ILP increased. Two recent relation learning systems, FOIL and LINUS, include sophisticated noise-handling mechanisms [Lavrač and Džeroski 1992b]. A post-processing mechanism based on reduced error pruning has also been used in FOCL [Pazzani et al. 1991, Brunk and Pazzani 1991], an extension to FOIL that combines empirical and explanation-based learning. Bottom-up ILP systems (e.g., CIGOL and GOLEM) can implement noise-handling by using a measure of information compression, an approach recently elaborated in Muggleton et al. [1992b] and Srinivasan et al. [1992].

One of the main concerns in our work on ILP has been the development and use of noise-handling techniques in empirical ILP. These issues are addressed in detail in Part II of the book.

1.6 Applications of ILP

An important application of ILP is knowledge acquisition in second generation expert systems, which use a first-order representation and possibly a deep model of the domain. A novel type of application is knowledge discovery in databases [Piatetsky and Frawley 1991, Frawley et al. 1991]. Finally, ILP can be used as a tool in various steps of scientific discovery, e.g., theory formation, design of experiments and theory verification [De Raedt 1993].

This section briefly outlines the role of ILP in knowledge acquisition, knowledge discovery in databases and scientific discovery. Other applications are only briefly mentioned here. An overview of several state-of-the-art ILP applications is given in Part III of the book.

Knowledge acquisition

The main problem to be faced when building expert systems is the acquisition of knowledge. Knowledge acquisition takes a lot of time since it is necessary to observe and interview domain experts who are often unable to formulate their expertise in a form suitable for machine use. This problem is known as the *knowledge acquisition bottleneck*. Machine learning contributes to the widening of this bottleneck by the development of tools that assist and partially automate the knowledge acquisition process. State-of-the-art inductive learning technology can be used to construct expert knowledge bases more effectively than traditional dialogue-based techniques for knowledge acquisition [Bratko 1992]. Currently, two trends have become prominent in this respect.

On the one hand, inductive logic programming is concerned with the use of more powerful concept description languages, facilitating the use of domain-specific knowledge – this is the main issue addressed in this book. ILP can be used to automatically construct knowledge bases for expert systems based on deep and qualitative models, for which propositional learning techniques are not sufficient.

An effective ILP knowledge acquisition tool is the system MOBAL [Morik 1991, Kietz and Wrobel 1992]. MOBAL includes a model acquisition tool which abstracts rule models from rules, a sort taxonomy tool which clusters constant terms occurring as arguments in training examples, and a predicate structuring tool which abstracts rule sets to a topology hierarchy. Another learning apprentice system that can be used as an interactive aid for building and refining a knowledge base is DISCIPLE [Tecuci and Kodratoff 1990] which has three integrated learning modules, a knowledge base consisting of action models and object knowledge and an expert system shell.

On the other hand, significant effort has been devoted to the development of toolkits of inductive learning techniques [Morik et al. 1991]. It has been recognized previously in the expert systems community that it is hard to solve the knowledge acquisition problem by using one individual technique and that it is more appropriate to have a 'toolkit' of techniques and select one or more of them according to the nature of the domain and type of knowledge under investigation [Harmon et al. 1988]. The machine learning community has come to the same conclusions, which led to the Machine Learning Toolbox (MLT) ESPRIT Project P2154 [Kodratoff et al. 1992] aimed at developing a workbench of machine learning tools, from which one or more can be selected in order to find the best solution to a specific problem. The *multistrategy learning* paradigm has also emerged recently [Michalski and Tecuci 1991]. A multistrategy learning system typically integrates two or more inference types (e.g., inductive, deductive, abductive) and/or computational mechanisms.

The LINUS environment [Lavrač et al. 1991a] presented in this book integrates various attribute-value inductive learners within a common ILP framework. It can thus be viewed as a toolkit of learning techniques, similar to MLT. The learning systems in the toolkit are a decision tree induction system and two rule induction systems. LINUS contributes to attribute-value learners the ILP framework which allows for learning relations in the presence of background knowledge. To ILP, LINUS brings the techniques for handling imperfect (noisy) data and the potential for practical applications.

Finally, let us mention that the empirical ILP approach to knowledge acquisition is more appropriate for domains where relatively well established domain (background) knowledge is available in addition to a large number of examples of a single concept. For domains where multiple concepts have to be acquired and little (possibly incorrect) background knowledge is available, the interactive ILP paradigm seems to be more appropriate.

Knowledge discovery in databases

Knowledge discovery in databases is concerned with the non-trivial extraction of implicit, previously unknown, and potentially useful information from data stored in databases [Frawley et al. 1991, Piatetsky and Frawley 1991]. Frawley et al. [1991] indicate that one of the main problems in knowledge discovery in databases is the problem of noisy data. As the majority of practically applicable learning systems with noise-handling capabilities are attribute-value learners, research in knowledge discovery in databases has mainly taken place in the context of a single relation, in an attribute-value framework. The problem of inducing intensional relation definitions taking into account the dependencies on other relations has received attention only lately, as one of the main issues addressed in current ILP research.

Current empirical ILP systems already show the potential for knowledge discovery from large collections of data. Empirical ILP systems, such as FOIL, GOLEM and LINUS, are already efficient enough to be applied to real-life domains [Bratko 1992]. However, in order to discover interdependencies among different relations, multiple predicate learners should be preferably applied to the knowledge discovery task. Examples of multiple predicate learners that show the potential for knowledge discovery are MOBAL, MPL [De Raedt et al. 1993a] and CLAUDIEN [De Raedt and Bruynooghe 1993].

Inductive data engineering, an novel application of ILP for knowledge discovery, focuses on the inductive discovery of integrity constraints (similar as CLAUDIEN) as well as on data/knowledge base maintenance/design in accordance with these constraints [Flach 1993].

Scientific knowledge discovery

There exists a parallel between the discovery of scientific theories and the construction of knowledge bases for expert systems [De Raedt 1993]. The discovery process by a scientist and the knowledge acquisition process by a knowledge engineer usually rely on a number of observations or examples which they generalize using knowledge about the domain. The result is a theory which is initially unknown and can be considered a new piece or a new formulation of knowledge. In the generation of a new theory it is useful to perform experiments which would confirm or falsify the currently built theory. Also, the generated theory is in fact a hypothesis which has to be thoroughly tested. ILP can help the scientist and the knowledge engineer in all the indicated phases of theory construction.

ILP can be used in the following stages of scientific discovery:

- in the empirical or interactive generation of general logical theories from specific observations,
- in the interactive generation of crucial experiments while discovering a logical theory, and
- in the systematic testing of a given logical theory by proposing crucial experiments that would falsify the theory [De Raedt 1993].

The generation of logical theories from sets of observations can be performed by both empirical and interactive ILP systems. The empirical ILP system GOLEM already generated theories which are meaningful to scientists and have been published in scientific literature [Muggleton et al. 1992a, King et al. 1992]. Other systems that already show the potential of ILP for scientific discovery from data which may contain noise are FOIL (and mFOIL) and LINUS. The aspect of logical theory formation using empirical ILP systems is elaborated throughout this book.

The second issue, the design of experiments during scientific discovery, is addressed by interactive ILP systems. In this task it is important that all experiments performed are relevant and that they address different settings explained by the theory. Examples of ILP systems potentially applicable for this task are MIS and CLINT. The third issue, the systematic testing of a hypothesized theory that falsifies it or partly confirms it, can also be addressed by interactive ILP systems [De Raedt et al. 1991].

Other applications

Among other ILP applications we first have to mention the software engineering task of inductive *logic program synthesis*. In this application, the task of an ILP learner is to induce a logic program from examples of program behavior. An initial, possibly incorrect program, may also be given. MIS [Shapiro 1983], one of the earliest ILP systems, was primarily applied to this problem (algorithmic program debugging). Another software engineering application of ILP is described in Bratko and Grobelnik [1993].

Another type of application, called *inductive engineering*, is concerned with the construction of a model/design of a device from a specification (examples) of its behavior [Mozetič 1987, Bratko et al. 1989]. This task is also known as design from first principles.

Both empirical and interactive ILP systems can be used in these application tasks. However, empirical ILP systems are more likely to be applied in practice for two reasons. First, there is more experience with learning single concepts from large collections of data than with deriving knowledge bases from a small number of examples. Second, empirical ILP systems are more efficient because of the use of heuristics, because there is no need to take into account dependencies among different concepts, and because no examples are generated [De Raedt and Bruynooghe 1992a].

Several applications illustrate the potential of ILP in learning from real-life data. These applications include learning qualitative models from sample behaviors [Bratko et al. 1991, Džeroski and Bratko 1992a], inducing temporal rules for satellite fault diagnosis [Feng 1991], predicting secondary structure of proteins [Muggleton et al. 1992a] and finite element mesh design [Dolšak and Muggleton 1992, Dolšak 1991, Džeroski and Dolšak 1991]. Some of these applications are outlined in Part III of the book.

2

Inductive logic programming

Inductive logic programming has its theoretical roots in computational logic [Lloyd 1990]. This chapter establishes the basic terminology used in logic programming and deductive databases and formally defines the tasks of empirical and interactive inductive logic programming. It also introduces a way of structuring the search space of program clauses based on the so-called θ-subsumption ordering of clauses introduced by Plotkin [1969].

2.1 Logic programming and deductive database terminology

This section briefly introduces the basic logic programming and deductive database terminology, which will be used throughout the book. For a comprehensive introduction to logic programming and the programming language Prolog, we refer the reader to Bratko [1990]. An introduction to deductive databases can be found in Ullman [1988]. A detailed theoretical treatment of logic programming and deductive databases can be found in Lloyd [1987]. The following definitions are abridged from Lloyd [1987], and take into account the Prolog syntax.

A *first-order alphabet* consists of variables, predicate symbols and function symbols (which include constants). A *variable* is represented by an upper case letter followed by a string of lower case letters and/or digits. A *function symbol* is a lower case letter followed by a string of lower case letters and/or digits. A *predicate symbol* is a lower case letter followed by a string of lower case letters and/or digits.

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. A constant is a function symbol of arity 0, i.e., followed by a bracketed 0-tuple of terms. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an *atomic formula*, or *atom.* Both L and its *negation* $\overline{L}$ are *literals* whenever L is an atomic formula. In this case L is called a *positive literal* and $\overline{L}$ is called a *negative literal.*

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 $L_1 \vee L_2 \vee ... L_m$. A clause can also be represented as a finite set (possibly empty) of literals. Thus the set $\{L_1, L_2, .. \overline{L_i}, \overline{L_{i+1}}, ...\}$ stands for the clause $(L_1 \vee L_2 \vee .. \overline{L_i} \vee \overline{L_{i+1}} \vee ...)$ which is equivalently represented as $L_1 \vee L_2 \vee .. \leftarrow L_i \wedge L_{i+1} \wedge ....$ Most commonly, this same clause is written as $L_1, L_2, .. \leftarrow L_i, L_{i+1}, ...$, where commas on the left-hand side of $\leftarrow$ denote disjunctions, and commas on the right-hand side denote conjunctions. A set of clauses is called a *clausal theory* and represents the conjunction of its clauses.

Literals, clauses and clausal theories are all *well-formed formulae* (*wff*). Let E be a wff or term. Let $vars(E)$ denote the set of variables in E. E is said to be *ground* if and only if $\text{vars}(E) = \emptyset$.

Horn clause A *Horn clause* is a clause which contains at most one positive literal.

Definite program clause A *definite program clause* is a clause which contains exactly one positive literal. It has the form

$$T \leftarrow L_1, \ldots, L_m$$

where $T, L_1, \ldots, L_m$ are atoms.

The positive literal (T) in a definite program clause is called the *head* of the clause. The conjunction of negative literals ($L_1, \ldots, L_m$) is called the *body* of the clause. A Horn clause with no positive literal is a *definite goal.* A *positive unit clause* is a clause of the form $L \leftarrow$, that is, a definite program clause with an empty body. In the Prolog terminology, such a clause is called a *fact* and is denoted simply by L.

Definite program A set of definite program clauses is called a *definite logic program.*

Only atoms are allowed in the body of definite program clauses. In the programming language Prolog [Bratko 1990], however, literals of the form *not* L, where L is an atom, are allowed, where *not* is interpreted under the *negation-as-failure* rule [Clark 1978].

Program clause A *program clause* is a clause of the form

$$T \leftarrow L_1, \ldots, L_m \tag{2.1}$$

where T is an atom, and each of $L_1, \ldots, L_m$ is of the form L or *not* L, where L is an atom.

Normal program A *normal program* is a set of program clauses.

Predicate definition 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 on some simple 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 predicate $daughter/2$. This predicate definition is also a normal 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

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

Below we define three restricted forms of program clauses:

Datalog clause A *Datalog clause* is a program clause with no function symbols of non-zero arity. This means that only variables and constants can be used as predicate arguments.

Constrained clause A *constrained* clause is a program clause in which all variables in the body also appear in the head.

Non-recursive clause A *non-recursive* program clause is a program clause in which the predicate symbol in the head does not appear in any of the literals in the body.

Among the important developments in relational databases are *deductive databases* [Lloyd 1987, Ullman 1988] which allow for both extensional and intensional definitions of relations. The logic programming school in deductive databases [Lloyd 1987] argues that deductive databases can be effectively represented and implemented using logic and logic programming. The definitions below are adapted from Ullman [1988] and Lloyd [1987].

Relation A n-ary *relation* p is a set of tuples, i.e., a subset of the Cartesian product of n domains $D_1 \times D_2 \times \ldots \times D_n$, where a *domain* (or a *type*) is a set of values. It is assumed that a relation is finite unless stated otherwise.

Relational database A *relational database* (RDB) is a set of relations.

A relation in a database is essentially the same as a predicate in a logic program. The basic difference between program clauses (2.1) and database clauses (2.2), defined below, is in the use of types. A clause is *typed* if each variable appearing in arguments of clause literals is associated with a set of values the variable can take. For example, in the relation $lives\_in(X, Y)$, we may want to specify that X is of type *person* and Y is of type *city*.

It should be noted that types provide a simple syntactic way of specifying semantic information and can increase the deductive efficiency by eliminating useless branches of the search space. In logic, the terms *sort* and *many-sorted logic* are used, whereas in logic programming the term *type* is used instead of *sort* [Lloyd 1987]; we will adopt the term *type*.

Database clause A database clause is a typed *program clause* of the form:

$$T \leftarrow L_1, \ldots, L_m \tag{2.2}$$

where T is an atom and $L_1, \ldots, L_m$ are literals.

Deductive database – DDB A deductive database is a set of *database clauses.*

Database clauses use variables and function symbols in predicate arguments. As recursive types and recursive predicate definitions are allowed, the language is substantially more expressive than the language of relational databases. If we restrict database clauses to be non-recursive, we obtain the formalism of deductive hierarchical databases.

Deductive hierarchical database – DHDB A *deductive hierarchical database* is a deductive database restricted to non-recursive predicate definitions and non-recursive types.

Non-recursive types determine finite sets of values which are constants or structured terms with constant arguments. While the expressive power of DHDB is the same as that of RDB, DHDB allows intensional relations which can be much more compact than a RDB representation.

As stated earlier, a predicate definition is a set of program clauses. A predicate can be defined *extensionally* as a set of ground facts or *intensionally* as a set of database clauses. It is assumed that each predicate is defined either extensionally or intensionally. A relation in a database is essentially the same as a predicate definition in a logic program. Table 2.1 relates deductive database [Ullman 1988] and logic programming [Lloyd 1987] terms that will be used throughout the book.

DDB terminology	*LP terminology*
relation name p	*predicate symbol* p
attribute of relation p	*argument of predicate* p
tuple $< a_1, \ldots, a_n >$	*ground fact* $p(a_1, \ldots, a_n)$
relation p – *a set of tuples*	*definition of predicate* p – *a set of ground facts*

Table 2.1: Relating database and logic programming terms.

2.2 Empirical ILP

The definition of inductive concept learning with background knowledge, presented in the introductory chapter, reads as follows: Given a set of training examples $\mathcal{E}$ and background knowledge $\mathcal{B}$, find a hypothesis $\mathcal{H}$, expressed in some concept description language $\mathcal{L}$, such that $\mathcal{H}$ is complete and consistent with respect to the background knowledge $\mathcal{B}$ and the examples $\mathcal{E}$. In inductive logic programming, $\mathcal{E}$ is a set of ground facts, where *positive examples* $\mathcal{E}^+$ are $true$, and *negative examples* $\mathcal{E}^-$ are $false$ in the intended interpretation.

To make the above definition of learning operational for ILP, the following notion of *coverage* is used in the definitions of the function $covers$ in equations (1.3) and (1.4).

Coverage Given background knowledge $\mathcal{B}$, hypothesis $\mathcal{H}$ and example set $\mathcal{E}$, hypothesis $\mathcal{H}$ covers example $e \in \mathcal{E}$ with respect to background knowledge $\mathcal{B}$ if $\mathcal{B} \cup \mathcal{H} \models e$, i.e.,

$$covers(\mathcal{B}, \mathcal{H}, e) = true \ \ if \ \ \mathcal{B} \cup \mathcal{H} \models e \tag{2.3}$$

Consequently, the function $covers(\mathcal{B}, \mathcal{H}, \mathcal{E})$ is defined as:

$$covers(\mathcal{B}, \mathcal{H}, \mathcal{E}) = \{e \in \mathcal{E} \mid \mathcal{B} \cup \mathcal{H} \models e\} \tag{2.4}$$

Again, completeness and consistency require that $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$, and $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^-) = \emptyset$, respectively.

The semantic notion of coverage is used in the definitions of ILP in this chapter. It states that e is covered by $\mathcal{H}$ given background knowledge $\mathcal{B}$ if e is a logical consequence of $\mathcal{B} \cup \mathcal{H}$. This statement is denoted by $\mathcal{B} \cup \mathcal{H} \models e$, where the logical symbol $\models$ stands for semantic (logical) entailment [Lloyd 1987]. Note that, according to the definition of semantic entailment, $\mathcal{B} \cup \mathcal{H} \models e$ if $\mathcal{B} \cup \mathcal{H} \models_I e$ in every possible interpretation I, i.e., whenever an interpretation I makes $\mathcal{B} \cup \mathcal{H}$ true, it also makes e true.

In practice, for a given language bias $\mathcal{L}$, an appropriate proof procedure must be used to check whether an example is entailed by $\mathcal{B} \cup \mathcal{H}$. Most often, the SLD-resolution[1] proof procedure [Lloyd 1987] (with

---

[1] SLD-resolution stands for Linear resolution with Selection function for Definite clauses. SLD resolution is sound which means that for any wffs F and G it holds that $F \models G$ if G can be derived from F using SLD-resolution ($F \vdash_{SLD} G$).

bounded or unbounded depth) is used to check whether an example can be deduced from $\mathcal{B} \cup \mathcal{H}$. For instance, the systems MIS and CIGOL employ a depth bound in order to avoid infinite loops. In depth-bounded SLD-resolution, unresolved goals in the SLD-proof tree at depth h are not expanded and are treated as failed. Alternatively, a forward chaining procedure can be applied to check entailment [Bry 1990].[2]

The above notion of coverage is called *intensional coverage*, as the background knowledge $\mathcal{B}$ is intensional and can contain both ground facts and non-ground clauses. On the other hand, the *extensional notion of coverage* requires extensional background knowledge $\mathcal{B}$ in the form of ground facts only. Both notions were already introduced in MIS [Shapiro 1983], which can work extensionally (the *lazy* strategy) or intensionally (roughly the *adaptive* strategy). Similarly, different notions of coverage can be used in MARKUS [Grobelnik 1992]. Most empirical ILP systems use the extensional notion of coverage, while interactive ILP systems mostly use the intensional notion of coverage.

In the case that $\mathcal{B}$ consists of intensional predicate definitions, extensional ILP systems (such as FOIL [Quinlan 1990] and GOLEM [Muggleton and Feng 1990]) have to transform $\mathcal{B}$ into a ground (h-easy) model $\mathcal{M}$ of the background theory $\mathcal{B}$. This model contains all true ground facts that can be derived from $\mathcal{B}$ by a SLD-proof tree of depth less than h. Instead of $covers(\mathcal{B}, \mathcal{H}, e)$, extensional ILP systems employ the function $covers_{ext}(\mathcal{M}, \mathcal{H}, e)$ [De Raedt et al. 1993a], defined below, to test whether an example e is extensionally covered by $\mathcal{H}$.

Extensional coverage A hypothesis $\mathcal{H}$ extensionally covers an example e with respect to a ground model $\mathcal{M}$ if there exists a clause $c \in \mathcal{H}$, $c = T \leftarrow Q$, and a substitution θ, such that $T\theta = e$ and $Q\theta = \{L_1, \ldots, L_m\}\theta \subseteq \mathcal{M}$. This is denoted by $covers_{ext}(\mathcal{M}, \mathcal{H}, e)$.

The task of empirical ILP, which is concerned with learning a single predicate from a given set of examples, can now be formulated as follows.

---

[2] For a more detailed discussion of the proof procedures for different ILP systems see De Raedt [1992], pp. 75–88.

Empirical ILP

Given:

- a set of training examples $\mathcal{E}$, consisting of true $\mathcal{E}^+$ and false $\mathcal{E}^-$ ground facts of an unknown predicate p,
- a description language $\mathcal{L}$, specifying syntactic restrictions on the definition of predicate p, and
- background knowledge $\mathcal{B}$, defining predicates q_i (other than p) which may be used in the definition of p and which provide additional information about the arguments of the examples of predicate p.

Find:

- a definition $\mathcal{H}$ for p, expressed in $\mathcal{L}$, such that $\mathcal{H}$ is complete and consistent with respect to the examples $\mathcal{E}$ and background knowledge $\mathcal{B}$.

In the following, we refer to the definition of p as the definition of the *target* relation. When learning from noisy examples, the completeness and consistency criteria need to be relaxed in order to avoid overly specific hypotheses [Lavrač and Džeroski 1992b].

Empirical ILP systems include FOIL [Quinlan 1990], mFOIL [Džeroski 1991], GOLEM [Muggleton and Feng 1990], ML-SMART [Bergadano and Giordana 1988], LINUS [Lavrač et al. 1991a], MARKUS [Grobelnik 1992] and RX [Tangkitvanich and Shimura 1992]. LINUS and FOIL upgrade attribute-value learners from the ID3 [Quinlan 1986] and AQ [Michalski 1969, Michalski et al. 1986] families towards a first-order logical framework. A different approach is used in GOLEM, which is based on Plotkin's notion of relative least general generalization (*rlgg*) [Plotkin 1969]. Another ILP system FOCL [Pazzani and Kibler 1992, Brunk and Pazzani 1991] is an extension of FOIL that combines empirical ILP and explanation based learning. The system MOBAL [Morik 1991, Kietz and Wrobel 1992] is an empirical system which learns multiple predicate definitions.

The complexity of learning grows with the expressiveness of the hypothesis language $\mathcal{L}$. Therefore, to reduce the complexity, various restrictions can be applied to clauses expressed in $\mathcal{L}$. For example, some of the current empirical ILP systems constrain $\mathcal{L}$ to function-free program clauses, as is the case in FOIL. LINUS [Lavrač et al. 1991a] induces an even more restricted form of program clauses, i.e., deductive hierarchical database clauses.

The ij-determinacy bias in GOLEM [Muggleton and Feng 1990] is in fact a semantic bias as it depends not only on the form of hypotheses, but also on the background knowledge. Various other restrictions can be used including argument types and symmetry of predicates in pairs of arguments [Lavrač et al. 1991a], input/output modes [Shapiro 1983], program schemata or rule models [Wrobel 1988, Morik 1991], predicate sets [Bergadano et al. 1989], parametrized languages [De Raedt 1992], integrity constraints [De Raedt et al. 1991] and determinations [Russell 1989].

2.3 Interactive ILP

The problem of interactive ILP [De Raedt 1992], which is typically incremental, can be formalized as follows.

Interactive ILP

Given:

- a set of training examples $\mathcal{E}$ possibly about multiple predicates,
- background knowledge $\mathcal{B}$ consisting of predicate definitions assumed to be correct,
- a current hypothesis $\mathcal{H}$ which is possibly incorrect,
- a description language $\mathcal{L}$, specifying syntactic restrictions on the definitions of predicates in $\mathcal{H}$,
- an example e labeled $\oplus$ or $\ominus$, and
- an oracle willing to answer membership queries (label examples as $\oplus$ and $\ominus$) and possibly other questions.

Find:

- a new hypothesis $\mathcal{H}'$, obtained by retracting and asserting clauses belonging to $\mathcal{L}$ from $\mathcal{H}$ such that $\mathcal{H}'$ is complete and consistent with respect to the examples $\mathcal{E} \cup \{e\}$ and background knowledge $\mathcal{B}$.

Whereas empirical ILP is concerned with learning a single target relation from a large collection of possibly noisy examples, interactive ILP systems attempt to learn multiple relations, which may be interdependent, from a small set of correct examples. During the process of learning, interactive ILP systems generate their own examples and ask the user (oracle) about their label (classification $\oplus$ or $\ominus$). They may also ask the user about the validity of the generalizations constructed.

An important issue addressed in interactive ILP is the issue of representation change, i.e., shift of bias. Namely, several interactive ILP systems can change the hypothesis language $\mathcal{L}$ during the learning process. Representation changes include inventing new predicates as well as moving to a more powerful hypothesis language. This invokes the question of bias management, i.e., deciding whether, when and how to shift bias. The question of representation change has only recently been addressed in empirical ILP [Kijsirikul et al. 1992, Muggleton 1992b, Srinivasan et al. 1992].

An early interactive ILP system, MIS [Shapiro 1983], introduced the notion of refinement operator (graph) which is used to structure the space of clauses. MIS searches the refinement graph in a breadth-first top-down (general to specific) manner.

An important representative of interactive ILP systems is CLINT [De Raedt and Bruynooghe 1989, 1992a]. It is based on a sound theoretical framework and includes most of the above-mentioned characteristics of interactive ILP systems. CLINT generates its own examples and asks questions about their truth values, uses integrity constraints, manages a series of hypothesis languages and shifts among them as appropriate [De Raedt 1992].

Several interactive ILP systems are based on inverting the resolution principle [Robinson 1965]. Most of the work done in this area is a special form of the general theory of inverse resolution introduced in CIGOL [Muggleton and Buntine 1988] and refined in Muggleton [1991].

CIGOL employs three kinds of operators: absorption, which generalizes clauses; intraconstruction, which introduces new auxiliary predicates; and truncation, which generalizes unit clauses. The system MARVIN [Sammut and Banerji 1986] employs a variant of the absorption operator, the operators $G1$ and $G2$ introduced by Wirth [1989] are similar to absorption and intraconstruction, while IRES [Rouveirol 1990] and ITOU [Rouveirol 1992] overcome some of the problems with truncation and absorption by introducing a saturation procedure.

2.4 Structuring the hypothesis space

Concept learning can be viewed as a search problem [Mitchell 1982]. States in the search space (also called the *hypothesis space*) are concept descriptions and the goal is to find one or more states satisfying some quality criterion. A learner can be described in terms of the structure of its search space, its search strategy and search heuristics.

This section introduces the θ-subsumption lattice which gives the structure of the search space of program clauses. Having structured the hypothesis space, a learner can search it blindly, in an uninformed way, or heuristically. In uninformed search, the depth-first or breadth-first search strategy can be applied. On the other hand, in heuristic search, the usual search strategies are hill-climbing or beam search. A detailed analysis of heuristics used to guide the search of clauses can be found in Chapter 8.

In ILP, the search space is determined by the language of logic programs $\mathcal{L}$ consisting of program clauses of the form $T \leftarrow Q$, where T is an atom $p(X_1, \ldots, X_n)$ and Q is a conjunction of literals $L_1, \ldots, L_m$. The vocabulary of predicate symbols q_i in the literals L_i of the body of a clause is determined by the predicates from the background knowledge $\mathcal{B}$. If $\mathcal{L}$ allows for recursive predicate definitions, the predicate symbol in L_i can also be the predicate symbol p itself.

The selected language bias $\mathcal{L}$ syntactically restricts the form of clauses which can be formulated from a given vocabulary of predicates, function symbols and constants of the language. To constrain the search space, the language bias $\mathcal{L}$ in actual ILP systems is some restricted form of logic programs, e.g., consisting of function-free program clauses (in FOIL), constrained deductive hierarchical database clauses (in LINUS), or definite clauses (in GOLEM).

In order to search the space of program clauses systematically, it is useful to structure the search space by introducing a partial ordering into a set of clauses based on the θ-subsumption. To define θ-subsumption, we first need to define substitution:

Substitution A *substitution* $\theta = \{X_1/t_1, \ldots, X_k/t_k\}$ is a function from variables to terms. The application $W\theta$ of a substitution θ to a wff W is obtained by replacing all occurrences of each variable X_j in W by the same term t_j [Lloyd 1987].

θ-subsumption 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]. Two clauses c and d are *θ-subsumption equivalent* if c θ-subsumes d and d θ-subsumes c. A clause is *reduced* if it is not θ-subsumption equivalent to any proper subset of itself.

The following examples are used to illustrate the above notions.

Example 2.1 (Substitution, subsumption)
Let c be the clause:

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

A substitution $\theta = \{X/mary, Y/ann\}$ applied to clause c is obtained by applying θ to each of its literals:

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

As pointed out in Section 2.1, 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 the set

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

is a proper subset of $\{daughter(X,Y), \overline{female(X)}, \overline{parent(Y,X)}\}$.

Clause c θ-subsumes the clause

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

under the substitution $\theta = \{Y/X\}$.

Clause c θ-subsumes the clause

$$\begin{aligned} daughter(mary, ann) \leftarrow\ & female(mary),\\ & parent(ann, mary),\\ & parent(ann, tom) \end{aligned}$$

under the substitution $\theta = \{X/mary, Y/ann\}$.

Clauses $c = daughter(X,Y) \leftarrow parent(Y,X), parent(W,V)$ and $d = daughter(X,Y) \leftarrow parent(Y,X)$ θ-subsume one another, i.e., are θ-subsumption equivalent. Clause d is reduced, while clause c is not. ■

θ-subsumption introduces the 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 (refinement)* of c and c is a *generalization* of c'. The only clause refinements usually considered by the learner 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 reduced clauses. This means that any two clauses have a least upper bound (*lub*) and a greatest lower bound (*glb*). Both the *lub* and *glb* are unique up to renaming of variables.

The second property of θ-subsumption leads to the following definition:

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

The rules for computing the *lgg* of two clauses are outlined in Section 3.1.1.

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]. Clause c is *at least as general* as clause c' with respect to background theory $\mathcal{B}$ if $\mathcal{B} \cup \{c\} \models c'$. 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 learning 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.

  - When generalizing c to c', $c' < c$, all the examples covered by c will also be covered by c' (since if $\mathcal{B} \cup \{c\} \models e$ holds then also $\mathcal{B} \cup \{c'\} \models e$ holds). This property is used to prune the search of more general clauses when e is a negative example: if c is inconsistent (covers a negative example) then all its generalizations will also be inconsistent. Hence, the generalizations of c do not need to be considered.

- When specializing c to c', $c < c'$, an example not covered by c will not be covered by any of its specializations either (since if $\mathcal{B} \cup \{c\} \not\models e$ holds then also $\mathcal{B} \cup \{c'\} \not\models e$ holds). This property is used to prune the search of more specific clauses when e is an uncovered positive example: if c does not cover a positive example none of its specializations will. Hence, the specializations of c do not need to be considered.

θ-subsumption provides the basis for two important ILP techniques:

- bottom-up *building of least general generalizations* from training examples, relative to background knowledge, and
- top-down *searching of refinement graphs.*

These two techniques and inverse resolution will be elaborated in the following chapter.

3

Basic ILP techniques

A search of the hypothesis space can be performed bottom-up or top-down. Generalization techniques search the hypothesis space in a bottom-up manner: they start from the training examples (most specific hypotheses) and search the hypothesis space by using generalization operators. Generalization techniques are best suited for interactive and incremental learning from few examples. On the other hand, specialization techniques search the hypothesis space top-down, from the most general to specific concept descriptions, using specialization operators. Specialization techniques are better suited for empirical learning in the presence of noise since top-down search can easily be guided by heuristics. This chapter introduces the basic generalization and specialization ILP techniques. An overview of the basic ILP techniques is also given in Flach [1992]. We further describe the unifying framework for generalization introduced by Muggleton [1991], as well as a unifying framework for specialization [Lavrač et al. 1992b] which is related to the GENCOL framework [De Raedt and Bruynooghe 1992b].

3.1 Generalization techniques

Generalization techniques search the hypothesis space in a bottom-up manner. Bottom-up learners start from the most specific clause (allowed by the language bias) that covers a given example and then generalize the clause until it cannot further be generalized without covering negative examples.

In bottom-up hypothesis generation, a generalization c' of clause c ($c' < c$) can be obtained by applying a θ-subsumption-based *generalization operator*.

Generalization operator Given a language bias $\mathcal{L}$, a generalization operator ρ maps a clause c to a set of clauses $\rho(c)$ which are generalizations of c:

$$\rho(c) = \{c' \mid c' \in \mathcal{L},\ c' < c\}$$

Generalization operators perform two basic syntactic operations on a clause:

1. apply an inverse substitution to the clause, and
2. remove a literal from the body of the clause.

In Sections 3.1.1 and 3.1.2 we briefly sketch two basic generalization techniques: *relative least general generalization* (*rlgg*, used in GOLEM) and *inverse resolution* (used in CIGOL), which are both used in hypothesis generation.

Muggleton [1991] introduced a unifying framework covering both relative least general generalization and inverse resolution, based on the notion of a *most specific inverse resolvent.* This framework is briefly introduced in Section 3.1.3. In this unifying framework, an *rlgg* is an *lgg* of most specific inverse linear derivations. This is a first step towards a unifying theory of empirical and interactive ILP. Another step towards integrating empirical and interactive ILP is the recent work on non-interactive theory revision [Mooney and Richards 1992, Tangkitvanich and Shimura 1992] and multiple predicate learning [De Raedt et al. 1993a].

3.1.1 Relative least general generalization

Plotkin's notion of least general generalization (*lgg*) [Plotkin 1969] is important for ILP since it forms the basis of cautious generalization algorithms which perform bottom-up search of the θ-subsumption lattice. Cautious generalization 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.

Let us recall the definition of the least general generalization of two clauses c and c' from Section 2.4: $lgg(c, c')$ is the least upper bound of c and c' in the θ-subsumption lattice. To actually compute the *lgg* of two clauses, *lgg* of terms, atoms and literals need to be defined first [Plotkin 1969].

lgg of terms $lgg(t_1, t_2)$

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 $lgg(s, t)$.

Examples:

$lgg([a, b, c], [a, c, d]) = [a, X, Y]$.
$lgg(f(a, a), f(b, b)) = f(lgg(a, b), lgg(a, b)) = f(V, V)$ where V stands for $lgg(a, b)$. When computing *lgg*s one must be careful to use the same variable for multiple occurrences of the *lgg*s of subterms, i.e., $lgg(a, b)$ in this example. This holds for *lgg*s of terms, atoms and clauses alike.

lgg of atoms $lgg(A_1, A_2)$

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 $p \neq q$.

lgg of literals $lgg(L_1, L_2)$

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.

Examples:

$lgg(parent(ann, mary), parent(ann, tom)) = parent(ann, X)$.
$lgg(parent(ann, mary), \overline{parent(ann, tom)}) = undefined$.
$lgg(parent(ann, X), daughter(mary, ann)) = undefined$.

lgg of clauses $lgg(c_1, c_2)$

Let $c_1 = \{L_1, ..., L_n\}$ and $c_2 = \{K_1, ..., K_m\}$. Then $lgg(c_1, c_2) = \{L_{ij} = lgg(L_i, K_j) \mid L_i \in c_1, K_j \in c_2$ and $lgg(L_i, K_j)$ is defined $\}$.

Examples:

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 $lgg(ann, tom)$.

The definition of *relative least general generalization* can now be introduced.

Relative least general generalization The *relative least general generalization* ($rlgg$) of two clauses c_1 and c_2 is their least general generalization $lgg(c_1, c_2)$ *relative* to background knowledge $\mathcal{B}$.

This technique will be discussed in some more detail in Section 4.2, which gives an overview of GOLEM. In short, if the background knowledge consists of ground facts, and K denotes the conjunction of all these facts, the $rlgg$ of two ground atoms A_1 and A_2 (positive examples), relative to the given background knowledge K, is defined as:

$$rlgg(A_1, A_2) = lgg((A_1 \leftarrow K), (A_2 \leftarrow K))$$

Example 3.1 (Relative least general generalization)

Given the positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$ and the background knowledge $\mathcal{B}$ for the family example in Section 1.4, the least general generalization of e_1 and e_2 relative to $\mathcal{B}$ 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 $female(eve)$.

The result of computing the $rlgg(e_1, e_2)$ can be found in Section 4.2. Finally, after the elimination of irrelevant literals, the following clause is generated by GOLEM:

$$rlgg(e_1, e_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

■

3.1.2 Inverse resolution

The basic idea of *inverse resolution*, introduced as a generalization technique to ILP by Muggleton and Buntine [1988], is to invert the *resolution* rule of deductive inference [Robinson 1965], e.g., to invert the SLD-resolution proof procedure for definite programs [Lloyd 1987]. The basic resolution step in propositional logic derives the resolvent $p \vee r$ given the premises $p \vee \overline{q}$ and $q \vee r$.

Example 3.2 (A propositional derivation tree)
Given the theory $\mathcal{T} = \{u \leftarrow v, v \leftarrow w, w\}$ suppose we want to derive u. Proposition w resolves with $v \leftarrow w$ to give v, which is then resolved with $u \leftarrow v$ to derive u. The derivation tree for this simple example is given in Figure 3.1.

■

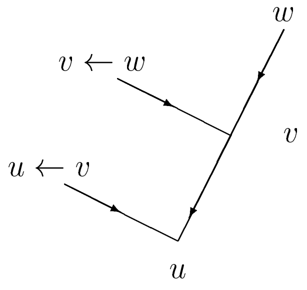


Figure 3.1: A simple propositional derivation tree.

The above example illustrates resolution in the propositional case. In a first-order case, resolution is more complicated, involving substitutions. Let $res(c, d)$ denote the resolvent of clauses c and d.

Example 3.3 (A first-order derivation tree)
To illustrate resolution in first-order logic, we again use the family example from Section 1.4. Suppose that background knowledge $\mathcal{B}$ consists of the clauses $b_1 = female(mary)$ and $b_2 = parent(ann, mary)$ and $\mathcal{H} = \{c\} = \{daughter(X, Y) \leftarrow female(X), parent(Y, X)\}$. Let $\mathcal{T} = \mathcal{H} \cup \mathcal{B}$. Suppose we want to derive the fact $e = daughter(mary, ann)$ from $\mathcal{T}$. To this end, we proceed as follows:

- First, the resolvent $c_1 = res(c, b_1)$ is computed under the substitution $\theta_1 = \{X/mary\}$. This means that the substitution θ_1 is first applied to clause c to obtain $daughter(mary, Y) \leftarrow female(mary), parent(Y, mary)$, which is then resolved with b_1 as in the propositional case. The resolvent of $daughter(X, Y) \leftarrow female(X), parent(Y, X)$ and $female(mary)$ is thus $c_1 = res(c, b_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$.
- The next resolvent $c_2 = res(c_1, b_2)$ is computed under the substitution $\theta_2 = \{Y/ann\}$. The clauses $daughter(mary, Y) \leftarrow parent(Y, mary)$ and $parent(ann, mary)$ resolve in $e = res(c_1, b_2) = daughter(mary, ann)$.

The linear derivation tree for this resolution process is given in Figure 3.2.

■

Inverse resolution, used in the interactive ILP system CIGOL [Muggleton and Buntine 1988], inverts the resolution process using the generalization operator based on inverting substitution [Buntine 1988]. Given a wff W, an *inverse substitution* θ^{-1} of a substitution θ is a function that maps terms in $W\theta$ to variables, such that $W\theta\theta^{-1} = W$.

Example 3.4 (Inverse substitution – a simple example)
Let $c = daughter(X, Y) \leftarrow female(X), parent(Y, X)$ and the substitution $\theta = \{X/mary, Y/ann\}$:

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

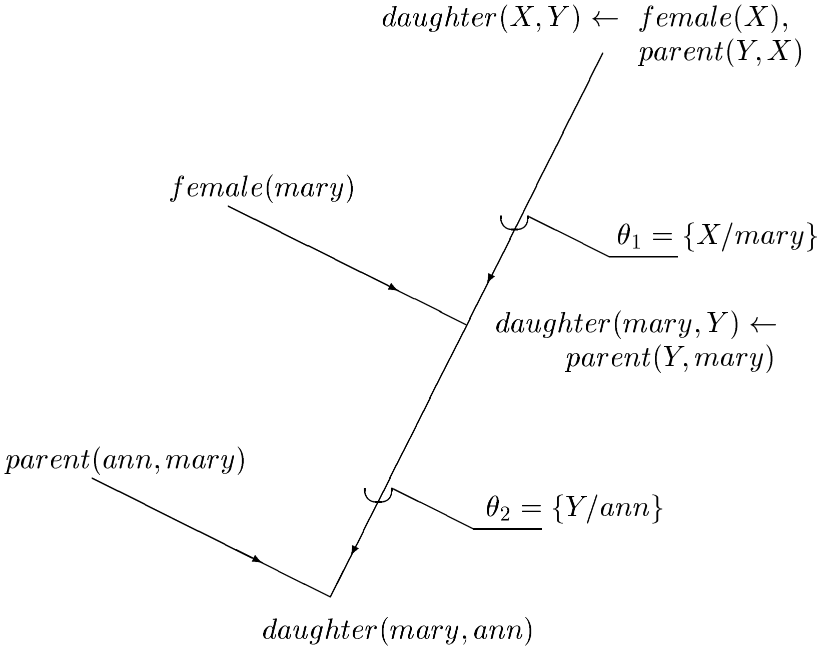


Figure 3.2: A first-order linear derivation tree.

By applying the inverse substitution $\theta^{-1} = \{mary/X, ann/Y\}$ the original clause c is obtained:

$$c = c'\theta^{-1} = daughter(X, Y) \leftarrow female(Y), parent(Y, X)$$

■

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 wff W are appropriately restored in $W\theta\theta^{-1}$.

Example 3.5 (Inverse substitution – with places)
Let $W = loves(X, daughter(Y))$ and $\theta = \{X/ann, Y/ann\}$. Applying the substitution to W results in $W\theta = loves(ann, daughter(ann))$. The inverse substitution $\theta^{-1} = \{(ann, \{< 1 >\})/X, (ann, \{< 2, 1 >\})/Y\}$ specifies that the first occurrence (at *place* $< 1 >$) of the subterm ann in the term $W\theta$ is replaced by variable X and the second occurrence (at place $< 2, 1 >$) is replaced by Y. The use of places in θ^{-1} ensures that $W\theta\theta^{-1} = loves(X, daughter(Y)) = W$.

■

Inverse resolution will not be elaborated theoretically, but will rather be illustrated by an example. Let $ires(c, d)$ denote the inverse resolvent of clauses c and d.

Example 3.6 (Inverse resolution)
As in Example 3.3, let background knowledge $\mathcal{B}$ consist of the two clauses $b_1 = female(mary)$ and $b_2 = parent(ann, mary)$. Let the current hypothesis $\mathcal{H} = \emptyset$. Suppose that the learner encounters the positive example $e_1 = daughter(mary, ann)$. 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 and can be added to the current hypothesis $\mathcal{H}$ instead of e_1. Choosing the inverse substitution $\theta_2^{-1} = \{ann/Y\}$, an inverse resolution step generates the clause $c_1 = ires(b_2, e_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$. Clause c_1 becomes the current hypothesis $\mathcal{H}$, such that $\{b_2\} \cup \mathcal{H} \models e_1$.

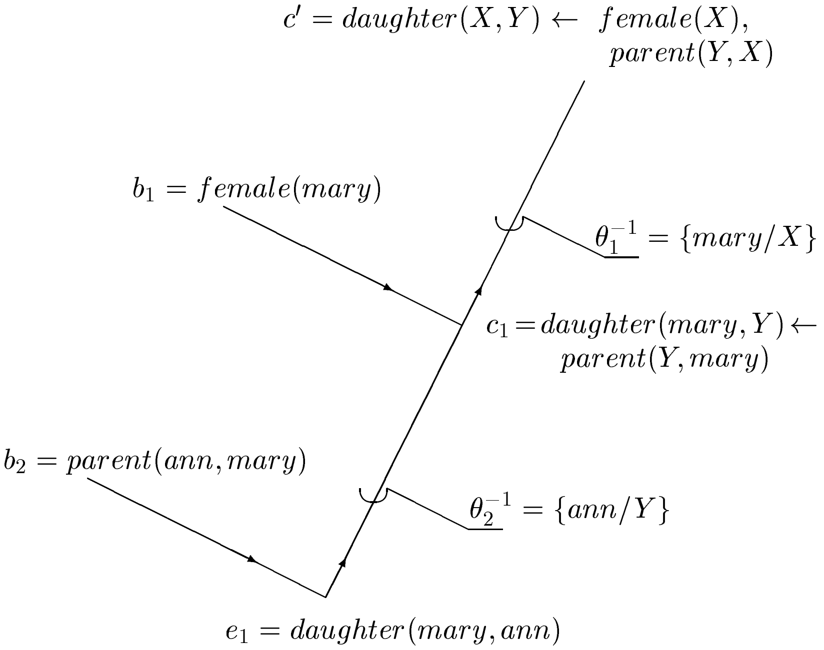


Figure 3.3: An inverse linear derivation tree.

- Inverse resolution then takes $b_1 = female(mary)$ and the current hypothesis $\mathcal{H} = \{c_1\} = \{daughter(mary, Y) \leftarrow parent(Y, mary)\}$. By computing $c' = ires(b_1, c_1)$, using the inverse substitution $\theta_1^{-1} = \{mary/X\}$, it generalizes the clause c_1 with respect to background knowledge $\mathcal{B}$, yielding $c' = daughter(X,Y) \leftarrow female(X), parent(Y,X)$. In the current hypothesis $\mathcal{H}$, clause c_1 can now be replaced by the more general clause c' which together with $\mathcal{B}$ entails the example e_1. Thus, the induced hypothesis is $\mathcal{H} = \{daughter(X,Y) \leftarrow female(X), parent(Y,X)\}$.

The corresponding inverse linear derivation tree is illustrated in Figure 3.3.

■

The generalization operator illustrated in the above example is called the *absorption* operator (the **V** operator), and is used in the ILP systems MARVIN [Sammut and Banerji 1986] and CIGOL [Muggleton and Buntine 1988] to generate hypotheses. Several other inverse resolution operators are used in CIGOL. An important operator is *intra-construction*, an instance of the **W** operator, which combines two **V** operators. This operator generates clauses using predicates which are not available in the initial learner's vocabulary. The ability to introduce new predicates in ILP is referred to as *predicate invention*.

3.1.3 A unifying framework for generalization

Inverse resolution is non-deterministic. Typically, at each inverse resolution step various generalizations of a clause can be performed, depending on the choice of the clause to be inversely resolved upon and the employed inverse substitution.

Example 3.7 (Non-determinism of inverse resolution)
To illustrate the problem of non-determinism, let us again take the setting of Example 3.6 where $e_1 = daughter(mary, ann)$ is the clause to be inversely resolved upon and $\mathcal{B}$ consists of clauses b_1 and b_2.

- First, at each inverse resolution step various clauses from $\mathcal{B}$ can be selected. Suppose that b_2 is selected first.

- Second, the question arises whether the generalization should be conservative, using an empty inverse substitution, or less cautious. In Example 3.6, given $b_2 = parent(ann, mary)$ and $e_1 = daughter(mary, ann)$, the inverse substitution $\theta_2^{-1} = \{ann/Y\}$ was used to generalize e_1 to clause $c_1 = ires(b_2, e_1) = daughter(mary, Y) \leftarrow parent(Y, mary)$. A more cautious generalization could be achieved by using the *most specific inverse substitution*, namely the empty inverse substitution $\theta_2^{-1} = \emptyset$ which would lead to the most specific inverse resolvent of b_2 and e_1: $c_1 = ires_{ms}(b_2, e_1) = daughter(mary, ann) \leftarrow parent(ann, mary)$.

■

To overcome the problem of non-determinism, Muggleton's unifying framework for relative least general generalization and inverse resolution [Muggleton 1991] proposes that *most specific inverse resolution* be used in the generalization process, using the *most specific inverse substitution* at each inverse resolution step. Let $ires_{ms}(c, d)$ denote the most specific inverse resolvent of clauses c and d.

Example 3.8 (Most specific inverse resolution)
Given $b_2 = parent(ann, mary)$ and $e_1 = daughter(mary, ann)$, using the empty substitution $\theta_2^{-1} = \emptyset$, the most specific inverse resolvent $c_1 = ires_{ms}(b_2, e_1)$ is constructed:

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

In the next inverse resolution step, the most specific inverse resolvent c' of c_1 and b_1 is generated, $c' = ires_{ms}(b_1, c_1)$, again using the empty substitution $\theta_1^{-1} = \emptyset$:

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

The *most specific inverse linear derivation tree* for this example is given in Figure 3.4.

■

Muggleton's unifying framework relates two generalization techniques: inverse resolution and relative least general generalization. More specifically, relative least general generalizations are least general generalizations of most specific inverse linear derivations. The main idea of this framework is illustrated by an example.

Example 3.9 (A unifying framework for generalization)
To illustrate Muggleton's unifying framework, let us assume that $\mathcal{B}$ consists of four clauses: b_1, b_2 as in Examples 3.8 and 3.6, and two additional clauses $b_3 = female(eve)$ and $b_4 = parent(tom, eve)$. Suppose that two positive training examples are given $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$. Two most specific inverse linear derivation trees are generated.

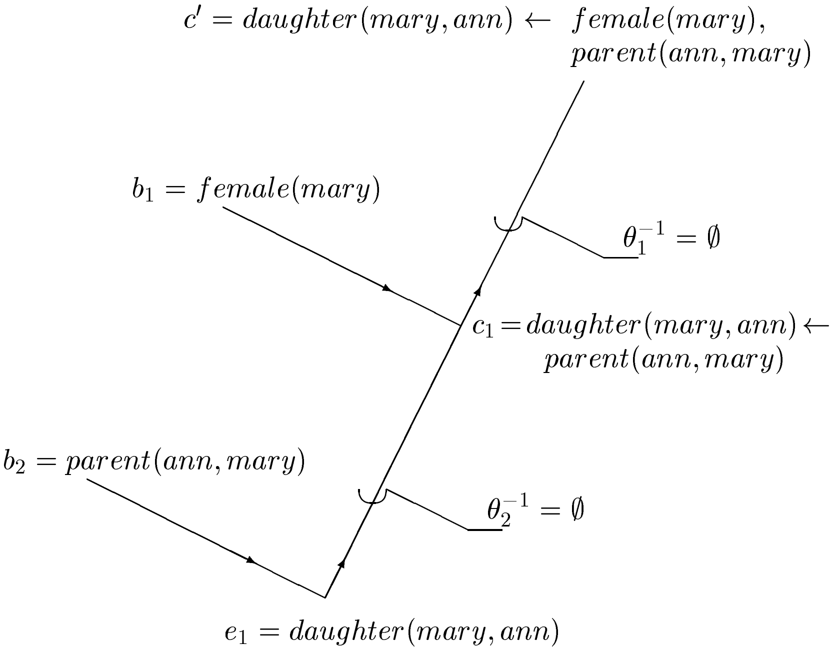


Figure 3.4: A most specific inverse linear derivation tree.

- The first tree is the same as in Example 3.8 illustrated in Figure 3.4. The most specific inverse resolution results in the clause:

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

- The second tree is generated analogously:
  - Under the empty inverse substitution, e_2 is first inversely resolved with b_4 into $c_3 = ires_{ms}(b_4, e_2)$:

    $$c_3 = daughter(eve, tom) \leftarrow parent(tom, eve)$$

  - Next, $c'' = ires_{ms}(b_3, c_3)$ is computed under the empty inverse substitution:

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

- Finally, the clause c is computed as the least general generalization of c' and c''. Clause $c = lgg(c', c'')$ generalizes e_1 and e_2.

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

The above process is illustrated in Figure 3.5.

In Section 3.1.1 we have outlined the method for computing the relative least general generalization of two examples e_1 and e_2:

$$rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$$

In Example 3.9, K is the conjunction of clauses b_1, b_2, b_3 and b_4. The result of computing the $rlgg(e_1, e_2)$ is, after the elimination of irrelevant literals (see Section 4.2), the same as the lgg of the most specific inverse linear derivations of e_1 and e_2:

$$c = rlgg(e_1, e_2) = daughter(X, Y) \leftarrow female(X), parent(Y, X)$$

■

Example 3.9 illustrates only the basic idea of the unifying framework for bottom-up generalization methods as used in CIGOL and GOLEM. In summary, relative least general generalizations are least general generalizations of most specific inverse linear derivations. The reader can find more details in Muggleton [1991].

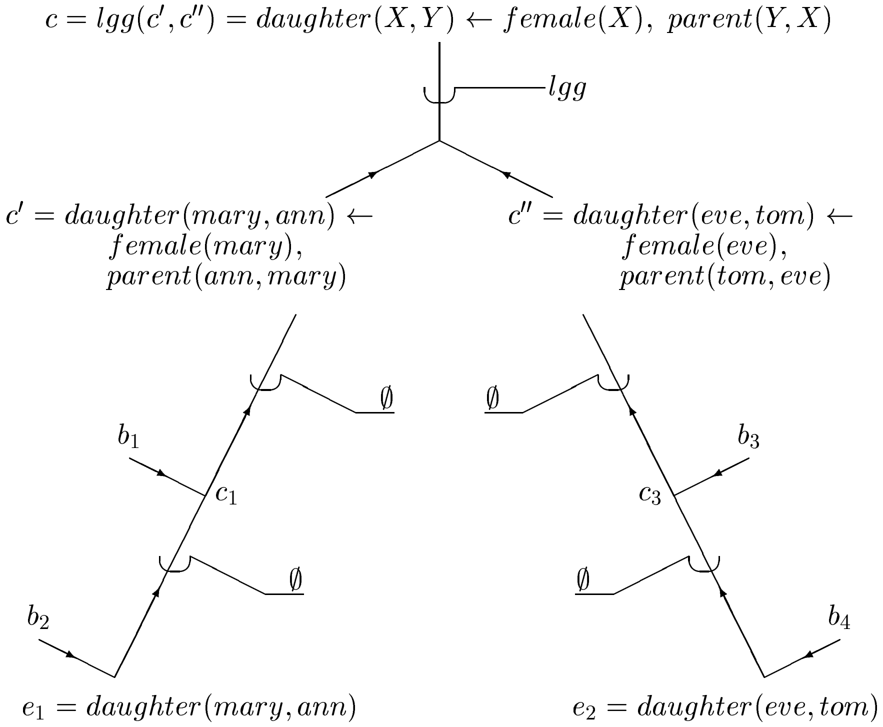


Figure 3.5: Muggleton's unifying framework for generalization.

3.2 Specialization techniques

Specialization techniques search the hypothesis space 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*.

Refinement operator Given a language bias $\mathcal{L}$, 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:

1. apply a substitution to the clause, and

2. add a literal to the body of the clause.

The basic specialization ILP technique is *top-down search of refinement graphs*. Top-down learners start from the most general clauses and repeatedly refine (specialize) them until they no longer cover negative examples; during the search they ensure that the clauses considered cover at least one positive example.

This ILP technique was first used in the interactive system MIS (Model Inference System [Shapiro 1983]). Section 3.2.1 briefly illustrates this technique on a simple example using an interactive MIS-like algorithm. On the other hand, top-down search of refinement graphs is the main technique used in empirical ILP. A unifying framework for empirical ILP algorithms which perform top-down search of refinement graphs is given in Section 3.2.2.

3.2.1 Top-down search of refinement graphs

For a selected language bias $\mathcal{L}$ and given background knowledge $\mathcal{B}$, 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 and used to direct the search from most general to specific hypotheses. A refinement graph is a directed, acyclic graph in which *nodes* are program clauses and *arcs* correspond to the basic refinement operations: substituting a variable with a term, and adding a literal to the body of a clause.

Algorithm 3.1 outlines the basic steps of the MIS algorithm [Shapiro 1983]. It assumes the language $\mathcal{L}$ of definite clauses, and the definitions of background knowledge predicates $\mathcal{B}$.

Algorithm 3.1 (A simplified MIS)

```
Initialize hypothesis $\mathcal{H}$ to a (possibly empty) set of clauses in $\mathcal{L}$.
repeat
  Read the next (positive or negative) example.
  repeat
    if there exists a covered negative example $e$ then
      Delete incorrect clauses from $\mathcal{H}$.
    if there exists a positive example $e$ not covered by $\mathcal{H}$ then
      With breadth-first search of the refinement graph
        develop a clause $c$ which covers $e$ and add it to $\mathcal{H}$.
  until $\mathcal{H}$ is complete and consistent.
  Output: Hypothesis $\mathcal{H}$.
forever
```

MIS interactively accepts new training examples. In the first **if** statement of the internal **repeat** loop, all the incorrect clauses are deleted from $\mathcal{H}$. Clause c in $\mathcal{H}$ is *incorrect* if c is responsible for $\mathcal{H}$ being inconsistent.

The main part of the algorithm is clause construction in the second **if** statement of the internal **repeat** loop: searching for a new clause of the target predicate definition that covers a positive example not covered by the current hypothesis. Search of the refinement graph starts with the most general clause and continues by searching clause refinements in a breadth-first manner. At each step, all minimal refinements are generated and tested for coverage. The acceptable refinements must cover the selected positive example. The process stops when the first acceptable consistent clause is found.

In Example 3.10 a MIS-like algorithm is illustrated. For simplicity, the language $\mathcal{L}$ is restricted to non-recursive definite clauses and only one specialization operation is used, namely, adding a literal to the body of a clause.

Example 3.10 (Searching the refinement graph)
Let us again take the family relations example from Section 1.4. The learner is first given the positive example $e_1 = daughter(mary, ann)$. Formally, the top-level node in the refinement graph is the most general clause $false$ (denoted by $\Box$, or alternatively, by the empty clause $\leftarrow$ which stands for $false \leftarrow true$). Practically, the search for clauses starts with the most general definition of the predicate $daughter$:

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

where an empty body is written instead of the body $true$. Since c covers e_1, the current hypothesis $\mathcal{H}$ is initialized to $\mathcal{H} = \{c\}$. The second example encountered $e_2 = daughter(eve, tom)$ is also positive and is covered by $\mathcal{H}$; therefore, the hypothesis does not need to be modified.

The third example encountered $e_3 = daughter(tom, ann)$ is negative. The learner deletes clause c from $\mathcal{H}$, since it covers the negative example. It now enters the second **if** statement in order to generate a new clause that covers the first positive example e_1. Since c covers the negative example e_3, the learner now generates the set of its refinements (minimal specializations) of the form

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

where L is one of following literals:[1]

- literals having as arguments the variables from the head of the clause: $X = Y$, $female(X)$, $female(Y)$, $parent(X, X)$, $parent(X, Y)$, $parent(Y, X)$, 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 $parent(Z, Y)$,

---

[1] Since the language is restricted to definite clauses, literals of the form *not* L are not considered. Also, as it is restricted to non-recursive clauses, literals with the target predicate $daughter/2$ are not considered.

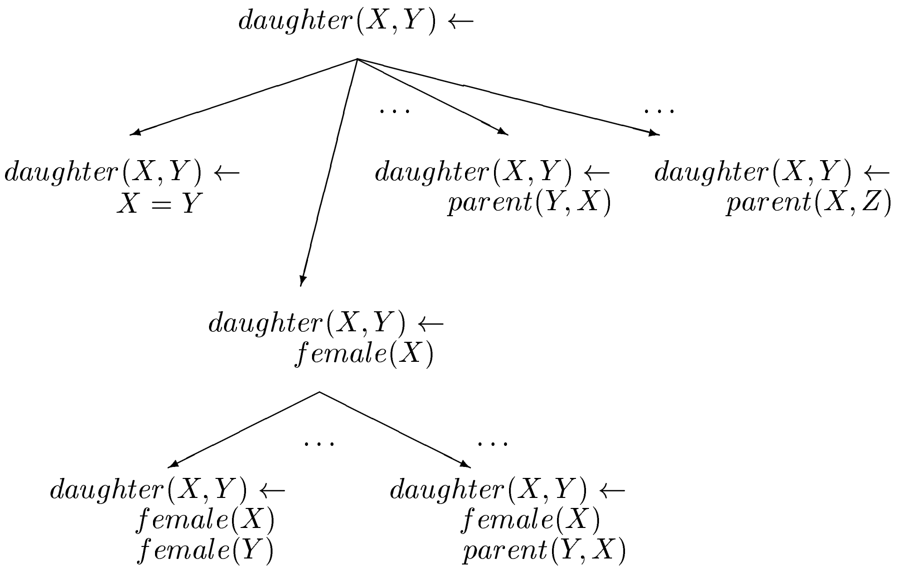


Figure 3.6: Part of the refinement graph for the family relations problem.

Part of the refinement graph for the family relations problem is given in Figure 3.6.

The refinements listed above are now considered one by one. The refinement $daughter(X,Y) \leftarrow X = Y$ is considered first, but is retracted as it does not cover the positive example e_1. The refinement $c' = daughter(X,Y) \leftarrow female(X)$ is considered next and is found to discriminate between positive and negative examples. As such, it is retained in the current hypothesis $\mathcal{H}$. This hypothesis would be the result of learning in the case that the learner encounters no more examples. However, suppose that another negative example $e_4 = daughter(eve, ann)$ is encountered. Clause c' then becomes inconsistent, since it covers e_4. It is deleted from $\mathcal{H}$ and the current hypothesis is again $\mathcal{H} = \emptyset$.

The clause generation process described above is now repeated, and the system tries again to build a clause that covers e_1. In this process, the remaining refinements of c are first considered. Of these, $daughter(X,Y) \leftarrow female(Y)$ and $daughter(X,Y) \leftarrow parent(Y,X)$ are retained in the breadth-first queue since they cover e_1, but are not consistent. The refinements of the clause $daughter(X,Y) \leftarrow X = Y$ are considered next, and are all retracted as they do not cover e_1. Finally, the refinements of c' are considered, among which $daughter(X,Y) \leftarrow female(X), parent(Y,X)$ covers e_1 and is consistent. As it happens, this clause also covers e_2, so the learner stops the search for clauses. ■

3.2.2 A unifying framework for specialization

Most top-down empirical ILP systems are based on the top-down search of refinement graphs, used in MIS. These include FOIL [Quinlan 1990], its successors mFOIL [Džeroski 1991] and FOCL [Pazzani et al. 1991, Brunk and Pazzani 1991], MARKUS [Grobelnik 1992], CLAUDIEN [De Raedt and Bruynooghe 1993] and MPL [De Raedt et al. 1993a].

This section presents a generic top-down empirical ILP algorithm which builds a unifying framework for most of the above empirical ILP systems. In order to present the algorithm the basic notation and terminology need to be established.

- A hypothesis $\mathcal{H} \in \mathcal{P}(\mathcal{L})$[2] is a set of program clauses $\{c \mid c \in \mathcal{L}\}$.
- The set $\mathcal{E}$ is the given (initial) training set. It consists of positive examples $\mathcal{E}^+$ and negative examples $\mathcal{E}^-$, $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$. Let n denote the number of examples in $\mathcal{E}$, $n^\oplus$ of which are positive and $n^\ominus$ negative, $n = n^\oplus + n^\ominus$.
- The hypothesis $\mathcal{H}$ currently built by the learner is called the *current hypothesis*, and the currently built clause c the *current clause*. The current clause c has the form $T \leftarrow Q$. The head T is an atom $p(X_1, \ldots, X_n)$ where p is the target predicate symbol, and the body Q is a conjunction of literals $L_1, \ldots, L_m$.
- The *current training set* $\mathcal{E}_{cur}$ is the set of training examples given to the learner when building the current clause c. It consists of n_{cur} examples, $n^\oplus_{cur}$ positive and $n^\ominus_{cur}$ negative, $n_{cur} = n^\oplus_{cur} + n^\ominus_{cur}$.

---

[2] $\mathcal{P}(\mathcal{L})$ denotes the set of finite subsets of $\mathcal{L}$.

- The *local training set* $\mathcal{E}_c$ is a subset of the current training set $\mathcal{E}_{cur}$ of examples covered by c. Let $n(c)$ denote the number of examples in the local training set, $n^{\oplus}(c)$ of which are positive and $n^{\ominus}(c)$ are negative, $n(c) = n^{\oplus}(c) + n^{\ominus}(c)$.

A *generic top-down empirical ILP algorithm* can now be outlined. It consists of three main steps:

1. *pre-processing* of the training set,
2. *construction* of a hypothesis, and
3. *post-processing* of the hypothesis.

Pre-processing of training examples

Pre-processing handles missing argument values in training examples and, when no negative examples are given, the generation of negative examples. Different ways of generating negative examples are possible. Most frequently, systems use the so-called generation under the 'closed-world assumption'. In this case, all examples which are not given as positive are generated and labeled negative [Quinlan 1990]. This method is only appropriate in exact, finite domains where all positive examples are in fact given.[3] The user must be most careful when applying the different methods for negative example generation (the 'closed-world assumption', 'partial closed-world assumption' or the 'near-misses' mode) and must at least be able to interpret the results accordingly [Lavrač 1990, Lavrač et al. 1991a].

Construction of a hypothesis

Hypothesis construction assumes the current training set $\mathcal{E}_{cur}$ (initially set to the entire training set $\mathcal{E}$) and the current hypothesis $\mathcal{H}$ (initially set to the empty set of clauses). The algorithm consists of two **repeat** loops, referred to as *covering* and *specialization.* The covering loop

---

[3] Even in exact domains the 'closed-world assumption' mode of negative examples generation is sometimes questionable, since most systems assume the same distribution of positive and negative examples in the training and testing set; this assumption can be violated in the case of negative examples generation.

performs hypothesis construction and the specialization loop performs clause construction.

The *covering* loop implements the covering algorithm, known from attribute-value learners AQ [Michalski 1983, Michalski et al. 1986] and CN2 [Clark and Niblett 1989]. It constructs a hypothesis in three main steps:

- construct a clause,
- add the clause to the current hypothesis, and
- remove from the current training set the positive examples covered by the clause.

This loop is typically repeated until all the positive examples are covered.

The main part is clause construction, implemented in the *specialization* loop of the generic ILP algorithm. Clause construction is performed by means of refinement operator ρ which takes the current clause $c = T \leftarrow Q$ and builds a set of its refinements $\rho(c) = \{c' \mid c < c'\}$ by adding a literal L to the body of c. Each refinement c' has the form $c' = T \leftarrow Q'$, where $Q' = Q, L$ (comma stands for conjunction).[4] The specialization loop starts with a clause $c = T \leftarrow$ which has an empty body. In hill-climbing search of the refinement graph, the algorithm keeps one 'best' clause and replaces it with its 'best' refinement at each specialization step, until the necessity stopping criterion is satisfied. In beam search, the algorithm keeps several 'best' clauses instead of one.

Algorithm 3.2 outlines the basic steps of a generic top-down empirical ILP algorithm. It assumes the given (pre-processed) set of training examples $\mathcal{E}$, the language bias $\mathcal{L}$, the definitions of background knowledge predicates $\mathcal{B}$, and the heuristics to be used in clause construction. It also assumes the hill-climbing search of the refinement graph.

The two **repeat** loops are controlled by two *stopping criteria* (also called *quality criteria*).

---

[4] We assume that all specializations of c occur by adding a literal to the body of clause $T \leftarrow Q$, including the specialization in which a variable in the head is replaced by a constant or a structured term. Otherwise, the notation $T' \leftarrow Q'$ should have been used.

Algorithm 3.2 (Generic top-down ILP algorithm)

```
Initialize $\mathcal{E}_{cur} := \mathcal{E}$.
Initialize $\mathcal{H} := \emptyset$.
repeat {covering}
  Initialize $c := T \leftarrow$ .
  repeat {specialization}
    Find the best refinement $c_{best} \in \rho(c)$.
    Assign $c := c_{best}$.
  until Necessity stopping criterion is satisfied.
  Add $c$ to $\mathcal{H}$ to get new hypothesis $\mathcal{H}' := \mathcal{H} \cup \{c\}$.
  Remove positive examples covered by $c$ from $\mathcal{E}_{cur}$ to get new
    training set $\mathcal{E}'_{cur} := \mathcal{E}_{cur} - covers(\mathcal{B}, \mathcal{H}', \mathcal{E}^{+}_{cur})$.
  Assign $\mathcal{E}_{cur} := \mathcal{E}'_{cur}, \mathcal{H} := \mathcal{H}'$.
until Sufficiency stopping criterion is satisfied.

Output: Hypothesis $\mathcal{H}$.
```

- In the clause construction (*specialization*) loop, the *necessity stopping criterion* decides when to stop adding literals to a clause.
- In the hypothesis construction (*covering*) loop, the *sufficiency stopping criterion* decides when to stop adding clauses to the hypothesis.

Different stopping criteria are usually applied in domains with perfect (e.g., exact, non-noisy) data and in domains with imperfect (e.g., noisy) data. In domains with perfect data, the necessity criterion requires consistency which means no covered negative examples, and the sufficiency criterion requires completeness which means coverage of all positive examples. In domains with imperfect data, instead of consistency and completeness, heuristic stopping criteria are applied. Consistency and completeness, as well as heuristic stopping criteria, are based on the number of positive and negative examples covered by a clause or hypothesis.

In intensional ILP systems, the set of examples covered by the current clause c is determined with regard to the current hypothesis $\mathcal{H}$, i.e., as $covers(\mathcal{B}, \mathcal{H} \cup \{c\}, \mathcal{E}_{cur})$. This notion is used in the generic top-down empirical ILP algorithm (Algorithm 3.2). Extensional ILP systems, such as FOIL, use the notion of extensional coverage at the appropriate places in the generic algorithm. As can be seen from the above definition of extensional coverage, the set of examples covered by the current clause is independent of the current hypothesis in extensional ILP systems, i.e., is computed as $covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}_{cur})$.

When defining the heuristics used in ILP systems, we will take $n^{\oplus}(c)$ and $n^{\ominus}(c)$ to be context dependent, i.e., to take into account the extensional/intensional notion of coverage used in the ILP system. In intensional systems, $n^{\oplus}(c)$ and $n^{\ominus}(c)$ will denote the numbers of positive and negative examples in the set $covers(\mathcal{B}, \mathcal{H} \cup \{c\}, \mathcal{E}_{cur})$. In extensional systems, $n^{\oplus}(c)$ and $n^{\ominus}(c)$ will denote the numbers of positive and negative examples in the set $covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}_{cur})$, where $\mathcal{M}$ is a ground model of $\mathcal{B}$.

Different heuristics are discussed in detail in Chapter 8, but are also outlined here for completeness. The simplest heuristic $f(c)$ is the *expected accuracy* of a clause $A(c) = p(\oplus|c)$. The expected accuracy is defined as the probability that an example covered by clause c is positive. *Informativity,* defined as $I(c) = -log_2\ p(\oplus|c)$, is another frequently used heuristic. More elaborate versions of these heuristics include *accuracy gain* $AG(c', c) = A(c') - A(c)$ and *information gain* $IG(c', c) = I(c) - I(c')$, as well as *weighted accuracy gain* $WAG(c', c)$ and *weighted information gain* $WIG(c', c)$. A variant of weighted information gain used in FOIL is described in Section 4.1. Different probability estimates (discussed in detail in Chapter 8) can be used to compute $p(\oplus|c)$, the most frequent (but not the most appropriate) being the *relative frequency* of covered positive examples $p(\oplus|c) = \frac{n^{\oplus}(c)}{n^{\oplus}(c)+n^{\ominus}(c)}$. More appropriate probability estimates are the Laplace estimate and the m-estimate [Cestnik 1990].

Post-processing of a hypothesis

Post-processing of an induced hypothesis $\mathcal{H}$ aims to reduce the complexity of $\mathcal{H}$ and to improve the accuracy of the hypothesis when classifying unseen cases. Note that post-processing of a single clause can also be performed immediately after the specialization loop, which is done in

FOIL (see Algorithm 4.1). Post-processing is performed by removing *irrelevant literals* from a clause and by removing *irrelevant clauses* from the hypothesis. The definitions of the irrelevance of literals/clauses follows.

Irrelevance of literals Let d be the clause obtained by removing literal L from the body of $c \in \mathcal{H}$. Literal L in clause c is *irrelevant* if $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^+) \subseteq covers(\mathcal{B}, \mathcal{H} \cup \{d\} \setminus \{c\}, \mathcal{E}^+)$ and $covers(\mathcal{B}, \mathcal{H} \cup \{d\} \setminus \{c\}, \mathcal{E}^-) \subseteq covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^-)$.[5]

Irrelevance of clauses Clause $c \in \mathcal{H}$ is *irrelevant* if

- $covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^+) \subseteq covers(\mathcal{B}, \mathcal{H} - \{c\}, \mathcal{E}^+)$, and
- $covers(\mathcal{B}, \mathcal{H} - \{c\}, \mathcal{E}^-) \subseteq covers(\mathcal{B}, \mathcal{H}, \mathcal{E}^-)$.

In perfect (non-noisy, exact) domains, irrelevant literals and irrelevant clauses (as defined above) are eliminated. On the other hand, in imperfect domains heuristic *simplification criteria* are used instead. The heuristics that evaluate the quality of a clause/set of clauses, mentioned in clause construction, can be used also as simplification criteria, the most frequent being the expected accuracy $A(c)$ or $A(\mathcal{H})$.

For practical reasons, instead of discussing various simplification criteria to be used in imperfect domains, we simply redefine the notion of *irrelevance* of literals and clauses using the expected accuracy heuristic. In this case, the definitions of literal/clause irrelevance are the following. A literal in clause c is irrelevant if it can be removed from the body of c without decreasing the expected classification accuracy $A(c)$ of the clause. A clause is irrelevant if its elimination from $\mathcal{H}$ does not decrease the accuracy of the hypothesis. The more formal definitions are stated below.

Irrelevance of literals – imperfect data Let d be the clause obtained from c by removing literal L from the body of c. Literal L in clause c is *irrelevant* if $A(c) \leq A(d)$.

Irrelevance of clauses – imperfect data Clause $c \in \mathcal{H}$ is irrelevant if $A(\mathcal{H}) \leq A(\mathcal{H} - \{c\})$.

---

[5] For extensional ILP systems, this amounts to $covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}^+) \subseteq covers_{ext}(\mathcal{M}, \{d\}, \mathcal{E}^+)$ and $covers_{ext}(\mathcal{M}, \{d\}, \mathcal{E}^-) \subseteq covers_{ext}(\mathcal{M}, \{c\}, \mathcal{E}^-)$.

A general post-processing mechanism, as used in an ILP system FOCL [Pazzani et al. 1991, Brunk and Pazzani 1991], is the *reduced error pruning*, which implements a sophisticated way of removing irrelevant literals/clauses using the expected accuracy estimate. The idea of reduced error pruning is based on Quinlan's mechanism for post-processing of if-then rules generated from decision trees [Quinlan 1987a, 1987b]. The post-processing mechanisms of FOIL and LINUS can be viewed as variants of reduced error pruning.

In Section 1.1, the classification accuracy was mentioned only as one of the success criteria. Other criteria could be used as simplification criteria in post-processing and as stopping criteria in clause construction. Therefore, other approaches should also be mentioned, especially the ones based on information compression [Muggleton and Buntine 1988, Muggleton et al. 1992b] and those based on the statistical significance of a clause (a set of clauses) [Kalbfleish 1979, Clark and Niblett 1989].

Part II

Empirical ILP

4

An overview of empirical ILP systems

This chapter gives an overview of several empirical ILP systems. Special attention is paid to FOIL, which has been the basis for a substantial part of the work in this book. The systems GOLEM and MOBAL are also briefly described. Some other empirical ILP systems are touched upon for completeness.

4.1 An overview of FOIL

FOIL [Quinlan 1990] extends some ideas from the attribute-value learning paradigm to the ILP paradigm. In particular, it uses a covering approach similar to AQ [Michalski 1983] and an information based search heuristic similar to ID3 [Quinlan 1986]. From MIS [Shapiro 1983] it inherits the idea of top-down search of refinement graphs.

The hypothesis language $\mathcal{L}$ in FOIL is restricted to function-free program clauses. Constants and compound terms may not appear in induced clauses. The body of a clause is a conjunction of literals A or $not\ A$, where A is an atom. Literals in the body have either a predicate symbol q_i from the background knowledge $\mathcal{B}$, or the target predicate symbol, which means that recursive clauses can be induced. At least one of the variables in the arguments of a body literal must appear in the head of the clause or in one of the literals to its left. No literals that bind a variable to a constant are allowed.[1]

Training examples are function-free ground facts (constant tuples). Background knowledge $\mathcal{B}$ consists of extensional predicate definitions

---

[1] Literals that bind a variable to a constant value have the form $X = value$. The lack of this feature could be circumvented in FOIL by introducing additional predicates into $\mathcal{B}$ which have the form $is\_a\_value(X)$ for each possible $value$. This feature has been recently added to FOIL4 [Cameron-Jones and Quinlan 1993].

given by a finite set of function-free ground facts, i.e., $\mathcal{B} = \mathcal{M}$ where $\mathcal{M}$ denotes a ground model.

The FOIL algorithm is basically the same as the generic empirical ILP algorithm from Section 3.2.2. It consists of three basic steps: pre-processing of training examples, hypothesis construction and hypothesis post-processing.

Negative examples are not necessarily given to FOIL. They may be generated in pre-processing, based on the closed-world assumption. Hypothesis construction in FOIL is basically the same as the *covering* loop of Algorithm 3.2. Post-processing of the induced hypothesis is a form of reduced error pruning, briefly mentioned in Section 3.2.2 and described in some more detail in Sections 8.3 and 8.4.

The *covering* loop of the FOIL algorithm (Algorithm 4.1) assumes the given (pre-processed) set of training examples $\mathcal{E}$, the language bias $\mathcal{L}$ of function-free program clauses, the extensional background knowledge $\mathcal{B}$, as well as the search heuristics and the heuristics used as the stopping criteria.

Algorithm 4.1 (FOIL – the covering algorithm)

```
Initialize $\mathcal{E}_{cur} := \mathcal{E}$.
Initialize $\mathcal{H} := \emptyset$.
repeat {covering}
   Initialize clause $c := T \leftarrow$ .
   Call the $SpecializationAlgorithm(c, \mathcal{E}_{cur})$
      to find a clause $c_{best}$.
   Assign $c := c_{best}$.
   Post-process $c$ by removing irrelevant literals to get $c'$.
   Add $c'$ to $\mathcal{H}$ to get a new hypothesis $\mathcal{H}' := \mathcal{H} \cup \{c'\}$.
   Remove positive examples covered by $c'$ from $\mathcal{E}_{cur}$ to get
      a new training set $\mathcal{E}'_{cur} := \mathcal{E}_{cur} - covers_{ext}(\mathcal{B}, \{c'\}, \mathcal{E}^+_{cur})$.
   Assign $\mathcal{E}_{cur} := \mathcal{E}'_{cur}, \mathcal{H} := \mathcal{H}'$.
until $\mathcal{E}^+_{cur} = \emptyset$ or encoding constraints violated.

Output: Hypothesis $\mathcal{H}$.
```

In the inner *specialization* loop, FOIL seeks a clause of the form

$$p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, ..., L_m$$

Finding a clause consists of a number of refinement steps. Search starts with the clause with an empty body. At each step, the clause $c_i = p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, \ldots, L_{i-1}$ built so far is refined by adding a literal L_i to its body. Literals are atoms A (or negated atoms *not* A) where A is of the form $X_j = X_s$ or $q_k(Y_1, Y_2, ..., Y_{n_k})$, where the X variables appear in c_i, the Y variables either appear in c_i or are new existentially quantified variables, and q_k is a predicate symbol from the background knowledge $\mathcal{B}$. At least one of the Y variables has to appear in c_i. Recursive calls, i.e., atoms of the form $p(Z_1, Z_2, ..., Z_n)$ may also be added to the body of c_i. This requires at least one irreflexive partial ordering $X_s < Z_s$, $1 \leq s \leq n$, to hold in order to prevent the clause from causing infinite recursion (see Chapter 7). A more sophisticated approach to preventing infinite recursion developed along these lines has been recently implemented within FOIL4 [Cameron-Jones and Quinlan 1993].

In the specialization loop, FOIL makes use of a *local training set* which is initially set to the current training set $\mathcal{E}_1 = \mathcal{E}_{cur}$. While $\mathcal{E}_{cur}$ consists of n-tuples, the local training set consists of m-tuples of constants, each of them being a value assignment to the m variables in the current clause. Let $\mathcal{E}_i$ denote the local training set of tuples that satisfy the current clause $c_i = p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, \ldots, L_{i-1}$. Let n_i denote the number of tuples in $\mathcal{E}_i$. Elements of $\mathcal{E}_i^+$ are positive tuples and elements of $\mathcal{E}_i^-$ are negative tuples; the numbers of tuples in these sets are denoted by $n_i^{\oplus}$ and $n_i^{\ominus}$, respectively.[2]

The FOIL specialization loop, implementing the function $SpecializationAlgorithm(c, \mathcal{E}_{cur})$, is given in Algorithm 4.2. In each refinement step, clause c_{i+1} is obtained by adding a literal L_i to the body of the clause $c_i = p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, \ldots, L_{i-1}$, which

---

[2] In the case that $m = n$, the m-tuples of constants in the local training set $\mathcal{E}_i$ consist of those tuples from $\mathcal{E}_{cur}$ that are covered by c_i. Therefore, the number of examples from $\mathcal{E}_{cur}$ covered by c_i could be denoted by $n(c_i)$. However, since the set contains 'extended' tuples (m can be greater than n), the distinction between the local training set $\mathcal{E}_i$ and the set of examples from $\mathcal{E}_{cur}$ covered by c_i needs to be kept and a different notation used.

Algorithm 4.2 (FOIL – the specialization algorithm)

```
Initialize local training set $\mathcal{E}_i := \mathcal{E}_{cur}$.
Initialize current clause $c_i := c$.
Initialize $i := 1$.
while $\mathcal{E}_i^- \neq \emptyset$ or encoding constraints violated do
   Find the best literal $L_i$ to add to the body of $c_i = T \leftarrow Q$
      and construct $c_{i+1} := T \leftarrow Q, L_i$.
   Form a new local training set $\mathcal{E}_{i+1}$ as a set of extensions of
      the tuples in $\mathcal{E}_i$ that satisfy $L_i$.
   Assign $c := c_{i+1}$.
   Increment $i$.
endwhile

Output: Clause $c$.
```

---

covers the set of tuples $\mathcal{E}_i$. The literal L_i is either an atom A of the form $q_k(Y_1, Y_2, \ldots, Y_{n_k})$ or $X_j = X_s$, or *not* A. Some of the variables $Y_1, Y_2, \ldots, Y_{n_k}$ belong to the 'old' variables already occurring in c_i, $\{OV_1, \ldots, OV_{Old}\}$, while some are 'new', $\{NV_1, \ldots, NV_{New}\}$, i.e., introduced by the literal L_i. The set of tuples $\mathcal{E}_{i+1}$ covered by clause c_{i+1} is the set of ground $(Old + New)$-tuples (instantiations of $< OV_1, \ldots,$ OV_{Old}, $NV_1, \ldots, NV_{New} >$) for which the body $L_1, L_2, \ldots, L_{i-1}, L_i$ is true. Producing a new training set $\mathcal{E}_{i+1}$ as a set of extensions of the tuples in $\mathcal{E}_i$ that satisfy L_i can be expressed in the relational algebra terminology by saying that $\mathcal{E}_{i+1}$ is the natural join of $\mathcal{E}_i$ with the relation corresponding to literal L_i.

Table 4.1 illustrates the FOIL specialization algorithm on the family relations problem from Section 1.4. The search starts with clause $c_1 = daughter(X, Y) \leftarrow$. The initial local training set $\mathcal{E}_1$ contains all training examples, two of which are positive ($n_1^\oplus = 2$) and two negative ($n_1^\ominus = 2$). Choosing literal $L_1 = female(X)$ gives rise to the new local training set $\mathcal{E}_2$ of tuples covered by c_2, which contains two positive ($n_2^\oplus = 2$) and one negative tuple ($n_2^\ominus = 1$). Adding the second literal $L_2 = parent(Y, X)$ to the body of clause c_2 produces clause c_3 with a local training set $\mathcal{E}_3$ containing only positive tuples. Thus, a consistent

$Current\ clause\ c_1 :\ daughter(X,Y) \leftarrow$					
$\mathcal{E}_1$	$(mary, ann)$	$\oplus$		$n_1^{\oplus} = 2$	$I(c_1) = 1.00$
	(eve, tom)	$\oplus$		$n_1^{\ominus} = 2$	
	(tom, ann)	$\ominus$	$L_1 = female(X)$		
	(eve, ann)	$\ominus$	$Gain(L_1) = 0.84$	$n_1^{\oplus\oplus} = 2$	
$Current\ clause\ c_2 :\ daughter(X,Y) \leftarrow female(X)$					
$\mathcal{E}_2$	$(mary, ann)$	$\oplus$		$n_2^{\oplus} = 2$	$I(c_2) = 0.58$
	(eve, tom)	$\oplus$		$n_2^{\ominus} = 1$	
	(eve, ann)	$\ominus$	$L_2 = parent(Y,X)$		
			$Gain(L_2) = 1.16$	$n_2^{\oplus\oplus} = 2$	
$Current\ clause\ c_3 :\ daughter(X,Y) \leftarrow female(X), parent(Y,X)$					
$\mathcal{E}_3$	$(mary, ann)$	$\oplus$		$n_3^{\oplus} = 2$	$I(c_3) = 0.00$
	(eve, tom)	$\oplus$		$n_3^{\ominus} = 0$	

Table 4.1: FOIL trace for the family relation problem.

clause c_3 (which covers no negative tuples) is generated.

If a positive literal with new (existentially quantified) variables is added to the body of the clause, the size (arity) of the tuples in the local training set increases. A tuple from $\mathcal{E}_i$ may also give rise to more than one tuple in $\mathcal{E}_{i+1}$. Consider again the problem of learning family relations, with the first literal $L_1 = parent(Y,Z)$ introducing a new variable Z (see Table 4.2). All of the tuples in $\mathcal{E}_1$ have two successors in $\mathcal{E}_2$, which contains eight 3-tuples. The tuples in $\mathcal{E}_{i+1}$ inherit the labels $\oplus$ and $\ominus$ from their parents in $\mathcal{E}_i$.

The choice of literals is directed by an entropy-based search heuristic, called *weighted information gain*, which can be described as follows. If the local set of tuples $\mathcal{E}_i$ contains n_i tuples, $n_i^{\oplus}$ of which are positive and $n_i^{\ominus}$ negative, the information needed to signal that a tuple is positive is $I(c_i) = -log_2(\frac{n_i^{\oplus}}{n_i}) = -log_2(\frac{n_i^{\oplus}}{n_i^{\oplus}+n_i^{\ominus}})$ (see Section 8.4 which explains the informativity and the information gain heuristics in more detail). Let the choice of the next literal L_i give rise to a new tuple set $\mathcal{E}_{i+1}$; the information needed for the same signal is then $I(c_{i+1}) = -log_2(\frac{n_{i+1}^{\oplus}}{n_{i+1}})$. Note that if L_i does not introduce new variables,

$Current\ clause\ c_1:\ daughter(X, Y) \leftarrow$				
$\mathcal{E}_1$	$(mary, ann)$	$\oplus$		$n_1^{\oplus} = 2$
	(eve, tom)	$\oplus$		$n_1^{\ominus} = 2$
	(tom, ann)	$\ominus$	$L_1 = parent(Y, Z)$	
	(eve, ann)	$\ominus$		$n_1^{\oplus\oplus} = 2$
$Current\ clause\ c_2:\ daughter(X, Y) \leftarrow parent(Y, Z)$				
$\mathcal{E}_2$	$(mary, ann, mary)$	$\oplus$		$n_2^{\oplus} = 4$
	$(mary, ann, tom)$	$\oplus$		
	(eve, tom, eve)	$\oplus$		
	(eve, tom, ian)	$\oplus$		
	$(tom, ann, mary)$	$\ominus$		$n_2^{\ominus} = 4$
	(tom, ann, tom)	$\ominus$		
	$(eve, ann, mary)$	$\ominus$		
	(eve, ann, tom)	$\ominus$		

Table 4.2: The effect of new variables on the current training set of tuples in FOIL.

then $\mathcal{E}_{i+1} \subseteq \mathcal{E}_i$, $n_{i+1}^{\oplus} \leq n_i^{\oplus}$ and $n_{i+1}^{\ominus} \leq n_i^{\ominus}$. However, if new variables are introduced, it can happen that $n_{i+1}^{\oplus} > n_i^{\oplus}$ and/or $n_{i+1}^{\ominus} > n_i^{\ominus}$. If $n_i^{\oplus\oplus}$ of the positive tuples in $\mathcal{E}_i$ are represented by one or more tuples in $\mathcal{E}_{i+1}$, the information gained by selecting literal L_i amounts to $Gain(L_i) = WIG(c_{i+1}, c_i) = n_i^{\oplus\oplus} \times (I(c_i) - I(c_{i+1}))$. The literal with the highest gain is selected at each step. $WIG(c_{i+1}, c_i)$ stands for weighted information gain, i.e., the *information gain* $I(c_i) - I(c_{i+1})$ *weighted* by the factor $n_i^{\oplus\oplus}$.

For the example, illustrated in Table 4.1, $I(c_1) = -log_2(\frac{2}{2+2}) = 1.00$, $I(c_2) = -log_2(\frac{2}{2+1}) = 0.58$, and $I(c_3) = -log_2(\frac{2}{2+0}) = 0.00$. As $n_1^{\oplus\oplus} = n_2^{\oplus\oplus} = 2$, we have $Gain(L_1) = 2 \times (I(c_1) - I(c_2)) = 2 \times 0.42 = 0.84$, and similarly $Gain(L_2) = 2 \times 0.58 = 1.16$.

The heuristic allows for efficient pruning, which is responsible for the efficiency of FOIL. Let L_i be a positive literal with $Gain(L_i) = n_i^{\oplus\oplus} \times (I(c_i) - I(c_{i+1}))$. L_i may contain new variables and their replacement with old variables can never increase $n_i^{\oplus\oplus}$. Moreover, such replacement can at best produce a set $\mathcal{E}_{i+1}$ containing only $\oplus$ tuples, i.e., with $I(c_{i+1}) = 0$. The maximum gain that can be achieved by any literal obtained by substituting new variables in L_i with old is $MaximumGain(L_i) = n_i^{\oplus\oplus} \times I(c_i)$. If this maximum gain is less than the best gain achieved by a literal so far, no literals that can be obtained with such replacement are considered.

To implement noise-handling, FOIL employs stopping criteria based on the *encoding length restriction*, which limits the number of bits used to encode a clause to the number of bits needed to explicitly indicate the positive examples covered by it. If a clause c_i covers $n^{\oplus}(c_i)$ positive examples out of n_{cur} examples in the current training set $\mathcal{E}_{cur}$, the number of bits used to encode the clause must not exceed $ExplicitBits(c_i, \mathcal{E}_{cur}) = log_2(n_{cur}) + log_2(\binom{n_{cur}}{n^{\oplus}(c_i)})$ bits. The construction of a clause is stopped (the *necessity stopping criterion*) when it covers no negative examples (is consistent) or when no more bits are available for adding literals to its body (see Section 8.4). In the latter case, the clause is retained in the hypothesis if it is more than 85% accurate or is discarded otherwise. The search for clauses stops (the *sufficiency stopping criterion*) when no new clause can be constructed under the encoding length restriction, or alternatively, when all positive examples are covered.

Improvements of FOIL, implemented in FOIL2.0 [Quinlan 1991], include *determinate* literals (an idea borrowed from GOLEM), types and mode declarations of predicates, as well as post-processing of clauses. Consider a partially developed clause which has some old variables and a corresponding set of tuples. Determinate literals [Quinlan 1991] are literals that introduce new variables for which each $\oplus$ tuple has exactly one extension and each $\ominus$ tuple has at most one extension in the new training set (see Section 5.6 for the definition of determinacy). Unless a literal is found with gain close to the maximum, all determinate literals are added to the body of the clause under development. Upon completion of the clause, irrelevant literals are eliminated. Determinate literals alleviate some of the problems in FOIL, which are due to its short-sighted hill-climbing search strategy.

4.2 An overview of GOLEM

Both the training examples $\mathcal{E}$ and the background knowledge $\mathcal{B}$ in GOLEM are restricted to ground facts. If $\mathcal{B}$ is given in the form of non-ground Horn clauses it has to be converted to an *h-easy* ground model $\mathcal{M}$ [Muggleton and Feng 1990]. Unlike in FOIL, terms containing function symbols are allowed to appear in both the examples and the background knowledge.

GOLEM is based on Plotkin's [1969] notion of relative least general generalization ($rlgg$), which in turn is based on the notion of least general generalization (lgg) introduced in Section 3.1.1.

Let us recall the definition of $rlgg$ from Section 3.1.1. If the background knowledge consists of ground facts, and K represents the conjunction of all these facts, the $rlgg$ of two positive training examples e_1 and e_2 relative to the given background knowledge, is defined as follows:

$$rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$$

Such a clause can contain infinitely many literals or at least grow exponentially with the number of examples. Since such a clause can be intractably large, GOLEM uses some constraints on introducing new variables into the body of the $rlgg$. The variables in the body of the $rlgg$ have to be *determinate*, i.e., their values have to be, directly or indirectly, uniquely determined by the values of the variables in the head of the $rlgg$ (see Section 5.6 for the definition of determinacy). Even under the determinacy constraint, $rlgg$s are usually quite long clauses which contain superfluous (irrelevant) literals in the body. GOLEM uses negative examples and mode declarations (specifying the input and the output arguments of a predicate) to reduce the size of the clauses. If eliminating a literal from the body does not cause the clause to cover negative examples, then the irrelevant literal is discarded and the reduction procedure is continued with the remaining clause.

The $rlgg$ of the positive examples $e_1 = daughter(mary, ann)$ and $e_2 = daughter(eve, tom)$ from the family example of Section 1.4 is computed as follows. 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

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

We further have

$$rlgg(e_1, e_2) = lgg((e_1 \leftarrow K), (e_2 \leftarrow K))$$

which produces the following clause:

$$\begin{gathered}\underline{d(V_{m,e}, V_{a,t}) \leftarrow} p(a,m), p(a,t), p(t,e), p(t,i), f(a), f(m), f(e), \\ p(a, V_{m,t}), \underline{p(V_{a,t}, V_{m,e})}, p(V_{a,t}, V_{m,i}), p(V_{a,t}, V_{t,e}), \\ p(V_{a,t}, V_{t,i}), p(t, V_{e,i}), f(V_{a,m}), f(V_{a,e}), \underline{f(V_{m,e})}\end{gathered}$$

In the above clause, $V_{x,y}$ stands for $lgg(x,y)$, for each x and y. Eliminating the irrelevant literals yields the following clause:

$$d(V_{m,e}, V_{a,t}) \leftarrow p(V_{a,t}, V_{m,e}), f(V_{m,e})$$

which stands for

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

To generate a single clause, GOLEM first randomly picks several pairs of positive examples, computes their determinate *rlgg*s and chooses the one with greatest coverage (the greatest number of examples covered). This clause is further generalized by randomly choosing new positive examples and computing the *rlgg*s of the clause and each of the examples. Again, among the resulting *rlgg*s the one with the greatest coverage is chosen. The generalization process is repeated until the coverage of the best clause stops increasing. In post-processing, the number of literals is reduced by eliminating irrelevant literals. This means that the clause is further generalized.

To construct definitions consisting of more than one clause, GOLEM uses the covering approach: it constructs a clause covering some positive examples, removes the covered examples from the training set and repeats the whole process. It should be noted that clauses induced by GOLEM may be allowed to cover some pre-determined small number of (noisy) negative examples, thus allowing the constructed definition to be inconsistent. A measure of information compression [Muggleton et al. 1992b] was recently used in conjunction with GOLEM to deal with noisy examples.

4.3 An overview of MOBAL

The system MOBAL [Morik 1991], a further development of BLIP [Morik 1989], is an integrated knowledge acquisition environment consisting of several tools. Some of them are: a *model acquisition tool* which abstracts rule models from rules, a *sort taxonomy tool* which clusters constant terms occurring as arguments in training examples $\mathcal{E}$, and a *predicate structuring tool* which abstracts rule sets to a topology hierarchy. Each of these tools provides information that is used in restricting the search space.

MOBAL contains RDT [Kietz and Wrobel 1992], an empirical ILP learner that learns multiple predicate definitions and uses MOBAL's inference engine to generate new facts (to extend the ground model of the current theory) whenever a new predicate definition is induced. Training examples $\mathcal{E}$ are ground facts, and no function symbols are allowed. The arguments of the examples are typed. The types (sorts) can be input by the user or can be computed automatically by the sort taxonomy tool.

In the focus of our interest are *rule models*, which represent a mechanism for reducing the search space and are used as templates for constructing hypotheses. They are provided by the user or can be derived from input rules by the model acquisition tool. In the process of learning, the use of rule models is further restricted by limitations imposed by the topology and the sort taxonomy.

A *rule model* R has the form $T \leftarrow L_1, \ldots, L_m$ where T and L_i are literal schemas. Each *literal schema* L_i has the form $Q_i(Y_1, \ldots, Y_{n_i})$ or *not* $Q_j(Y_1, \ldots, Y_{n_j})$ where all non-predicate variables Y_k are implicitly universally quantified. The *predicate variable* Q_i can be instantiated only to a predicate symbol of the specified arity n_i.

The search for hypotheses in RDT is guided by rule models. The hypothesis space that is actually searched is the set of all predicate ground instantiations of rule models with predicates from the background knowledge $\mathcal{B}$. An *instantiation* Θ is a second-order substitution, i.e., a finite set of pairs P/p where P is a predicate variable and p is a predicate symbol. An instantiation of a rule model R (denoted by $R\Theta$) is obtained by replacing each predicate variable in R by the corresponding predicate symbol from the set Θ. A second-order rule model

R is *predicate ground* if all of its predicate variables are instantiated; in this case, R becomes a first-order rule.

For learning the *daughter* relationship from the family relations problem of Section 1.4 the user also needs to define (among other rule models) the model:

$$P(X,Y) \leftarrow R(X), Q(Y,X)$$

In this case, the correct clause

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

can be induced. Otherwise, RDT would in vain try to find this clause. The above model could also be instantiated to the clause

$$son(X,Y) \leftarrow male(X), parent(Y,X)$$

We can thus see that one rule model can be useful for solving several learning problems.

The space of hypotheses, limited by rule models, is still very large. The hierarchical order of rule models based on extended θ-subsumption allows the system to prune parts of the hypothesis space. For each rule model, all possible instantiations of predicate variables are tried systematically and are tested against the ground facts in $\mathcal{E}$. An instantiation is possible if the predicates that substitute predicate variables have a compatible arity and correspond to the sort and topology limitations. The topology hierarchy states which predicates are to be used in a model. Sorts represent a syntactic way of expressing semantics; for each argument of each predicate a set of possible values is computed. When instantiating the rule model, a substitution of a predicate variable is valid only if the sorts of arguments correspond to sorts of arguments of an already instantiated predicate variable. RDT can also use constants in the induced hypotheses, but the rule models have to be designed accordingly.

4.4 Other empirical ILP systems

Early empirical systems that touched the problem of learning relational descriptions are INDUCE [Michalski 1983] and ML-SMART [Bergadano and Giordana 1988, 1990]. They are both multiple predicate learners, but only for the case of nullary predicates (i.e., predicates corresponding to propositional classes). ML-SMART also has an explanation-based learning capability.

Many empirical ILP systems, based on the FOIL approach, have emerged recently. These include CHAM [Kijsirkul et al. 1991], FORTE [Richards and Mooney 1991], RX [Tangkitvanich and Shimura 1992] and FOCL [Pazzani and Kibler 1992, Pazzani et al. 1991].

CHAM was initially designed to overcome the problems of FOIL caused by hill-climbing in the case of non-discriminating and non-determinate literals. For that purpose it introduces a similarity-based heuristic, called likelihood, which is used in addition to the information gain heuristic. Other differences to FOIL include a seed-based beam search, similar to the one in AQ. A further extension of CHAM, called CHAMP [Kijsirikul et al. 1992], effectively addresses the problem of inventing new predicates, i.e., representation change during the learning process. However, the issue of representation change in the presence of noise is left unresolved.

FOCL, FORTE and RX belong to a family of ILP learners that could be named non-interactive theory revisors. Given an initial definition of the target concept, and an additional set of training examples which disagree with the initial definition, the task is to construct a new definition, based on the old one, which will agree with all the training examples. In the limit, when no initial target concept definition is given, they behave very similar to FOIL. Otherwise, they attempt to detect and repair incorrect clauses or, if positive examples are not covered, to induce clauses that will cover them.

FOCL and RX integrate ILP and explanation-based learning. FOCL uses the information-gain metric from FOIL as a universal measure to decide whether to operationalize existing parts of the definition or inductively construct new parts of the definition. RX also operationalizes the existing theory and constructs new operational parts, but in addition de-operationalizes the constructed definition in the end. Of the above systems, only FOCL is able to handle noisy data [Brunk and Pazzani 1991]. After constructing a consistent and complete definition, a *reduced error pruning* procedure is applied to it. An outline of this procedure can be found in Section 8.4.

The empirical ILP system MARKUS [Grobelnik 1992] is a variant of MIS [Shapiro 1983]. It also belongs to the class of empirical theory revisors, as it can take an initial theory and correct/complete it. The main differences from MIS include: the optimal generation of the refinement graph (without duplicates), the use of iterative deepening instead of breadth-first search, the removal of irrelevant clauses, and the use of negation.

A recent empirical ILP system addressing the problem of learning multiple predicates is MPL (Multiple Predicate Learner [De Raedt et al. 1993a, 1993b]). Although it might seem easy and natural to apply a single predicate learner several times in order to generate definitions of several predicates, there are serious problems with this approach, e.g., when learning mutually recursive predicates. The problems stem from the order dependency of the single predicate learning tasks and the use of the extensional definition of coverage in most empirical ILP systems. Based on ideas from FOIL, MPL overcomes these problems and is able to solve multiple predicate learning problems beyond the scope of other empirical ILP systems.

5

LINUS: Using attribute-value learners in an ILP framework

This chapter presents a method for effectively using background knowledge in learning both propositional and relational descriptions. The method, implemented in the system LINUS, employs propositional learners in a more expressive logic programming framework. LINUS is an ILP system integrating several attribute-value learners. It can be viewed as an ILP toolkit of learning algorithms out of which one or more can be selected in order to find the best solution to a given problem. The chapter first describes three attribute-value learners which are part of the LINUS environment. The method of using background knowledge in learning is described both for attribute-value and ILP problems. The LINUS algorithm, implementing this method, is then outlined, followed by a description of the system's language bias of constrained deductive hierarchical database (DHDB) clauses. Next, the computational complexity of the LINUS learning task is addressed. Finally, an extension to the original LINUS algorithm, called DINUS (Determinate LINUS), is proposed which allows for extending the language bias of constrained DHDB clauses to determinate clauses.

5.1 An outline of the LINUS environment

LINUS is an ILP learner which induces hypotheses in the form of constrained deductive hierarchical database (DHDB) clauses. As input it takes training examples $\mathcal{E}$, given as ground facts, and background knowledge $\mathcal{B}$ in the form of (possibly recursive) deductive database (DDB) clauses. The definitions of DDB and DHDB clauses can be found in Section 2.1.

The main idea in LINUS is to transform the problem of learning relational DHDB descriptions into a propositional learning task [Lavrač et al. 1991a]. This is achieved by the so-called *DHDB interface*, consisting of over 2000 lines of Prolog code. At present, LINUS incorporates the propositional learners ASSISTANT [Cestnik et al. 1987], NEWGEM [Mozetič 1985b] and CN2 [Clark and Boswell 1991]. It is easy to incorporate other propositional learners, out of which one or more can be selected to be applied.

The interface transforms the training examples from the DHDB form into the form of attribute-value tuples. The most important feature of this interface is that, by taking into account the types of the arguments of the target predicate, applications of background predicates and functions are considered as attributes for attribute-value learning. Existing attribute-value learners can then be used to induce if-then rules. Finally, the induced if-then rules are transformed into the form of DHDB clauses by the DHDB interface. The scheme of the LINUS environment is given in Figure 5.1.

LINUS is an integrated environment for learning both attribute and relational descriptions using attribute-value learners. The LINUS algorithm is a descendant of the learning algorithm used in QuMAS (Qualitative Model Acquisition System [Mozetič 1987]) which was used to learn functions of components of a qualitative model of the heart in the KARDIO expert system for diagnosing cardiac arrhythmias [Bratko et al. 1989].

LINUS was successfully applied to the problem of learning diagnostic rules in rheumatology, described in Chapter 11 [Lavrač et al. 1991b, 1993], the problem of learning relational descriptions in several domains known from the machine learning literature [Lavrač et al. 1991a], described also in Chapter 6, and the problem of learning rules for finite element mesh design in CAD [Džeroski 1991], described also in Chapter 12. Using propositional attribute-value learning algorithms, LINUS allows for recent advances in handling imperfect data in these algorithms to be applied easily to real-life, imperfect data. Experiments in a chess endgame domain with a controlled amount of noise, described in Chapter 10 [Džeroski and Lavrač 1991b, Lavrač and Džeroski 1992b], show that LINUS performs well on imperfect, noisy data of relational nature.

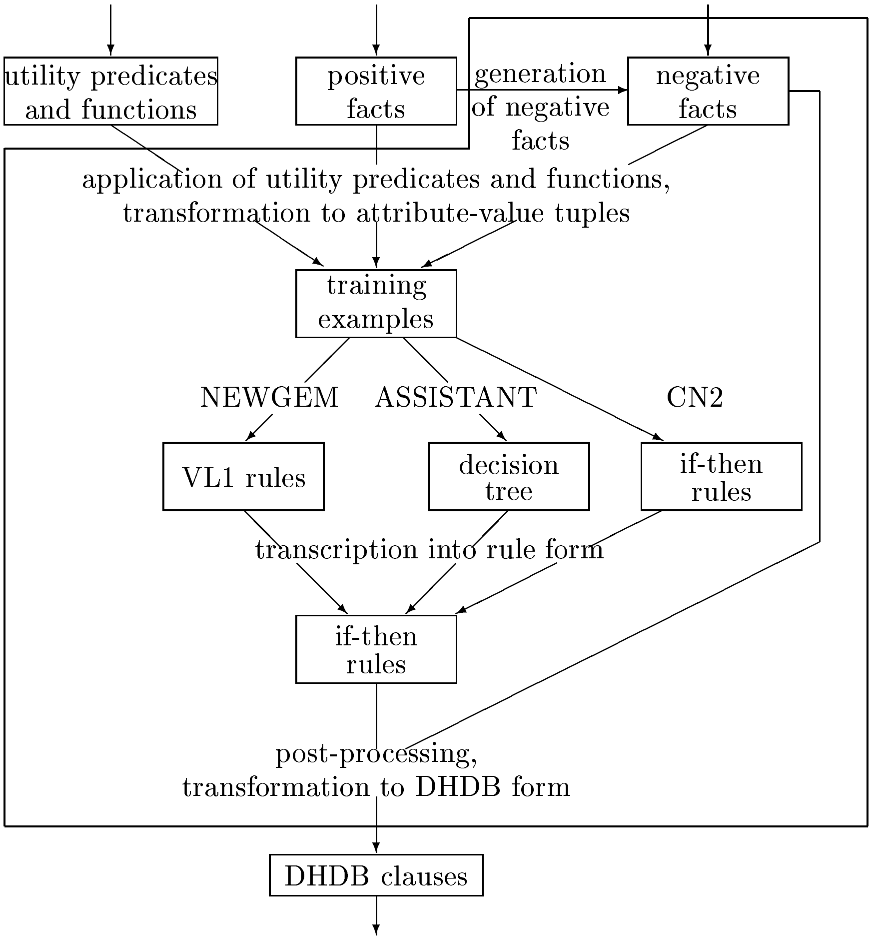


Figure 5.1: An overview of LINUS.

5.2 Attribute-value learning

LINUS incorporates several attribute-value learners into the logic programming environment. Let us first recall the propositional learning setting: examples are tuples of attribute values labeled with a concept name (or labeled $\oplus$ for the positive instances of the concept and $\ominus$ for the negative instances), and the induced hypotheses typically have the form of if-then rules or decision trees. This section illustrates a simple propositional learning task and gives an outline of three propositional learners incorporated into the LINUS environment.

5.2.1 An example learning problem

Suppose that the learning task is to find a description of friendly and unfriendly robots [Wnek et al. 1990] from a set of examples described by six attributes. In our simplified problem, given in Table 5.1, six examples are given, described by five attributes of three different types. Attributes $Smiling$ and Has_tie have values of the type $yes\_no = \{yes, no\}$, attribute $Holding$ has values of the type $holding = \{sword, balloon, flag\}$, and attributes $Head\_shape$ and $Body\_shape$ have values of the type $shape = \{round, square, octagon\}$.

From the examples in Table 5.1, an algorithm from the AQ family induces the following if-then rules:

	Attributes and values				
Class	*Smiling*	*Holding*	*Has_tie*	*Head_shape*	*Body_shape*
friendly	*yes*	*balloon*	*yes*	*square*	*square*
friendly	*yes*	*flag*	*yes*	*octagon*	*octagon*
unfriendly	*yes*	*sword*	*yes*	*round*	*octagon*
unfriendly	*yes*	*sword*	*no*	*square*	*octagon*
unfriendly	*no*	*sword*	*no*	*octagon*	*round*
unfriendly	*no*	*flag*	*no*	*round*	*octagon*

Table 5.1: Examples of friendly and unfriendly robots.

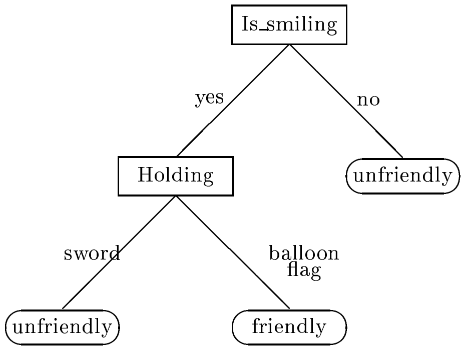


Figure 5.2: Decision tree for the robot world.

$$\begin{array}{lll} Class = friendly & \textbf{if} & [Smiling = yes] \wedge \\ & & [Holding = balloon \vee flag] \\ Class = unfriendly & \textbf{if} & [Smiling = no] \bigvee \\ & & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

Using an ID3-based decision tree building algorithm ASSISTANT, the tree in Figure 5.2 is induced. If transcribed into the form of rules, the tree produces exactly the above if-then rules.

5.2.2 ASSISTANT

ASSISTANT is a descendant of CLS (Concept Learning System [Hunt et al. 1966]) and ID3 [Quinlan 1979, 1986]. The ID3 algorithm generates a hypothesis in the form of a decision tree that enables the classification of new objects. As such, it is a member of the TDIDT (Top-Down Induction of Decision Trees [Quinlan 1986]) family of inductive learning programs.

CLS and ID3 implement a simple mechanism for generating a decision tree from a given set of attribute-value tuples. Each of the interior nodes of the tree is labeled by an attribute, while branches that lead from the node are labeled by the possible values of the attribute. The tree construction process is heuristically guided by choosing the 'most informative' attribute at each step, aimed at minimizing the expected number of tests needed for classification.

Let $\mathcal{E}$ be the set of training examples, and $C_1, \ldots C_N$ the decision classes. ID3 constructs a decision tree by repeatedly calling Algorithm 5.1 in each generated node of the tree.

Algorithm 5.1 (ID3)

```
Initialize $\mathcal{E}_{cur} := \mathcal{E}$.
if All the examples in $\mathcal{E}_{cur}$ are of the same class $C_j$
then Generate a leaf labeled $C_j$.
else
  Select the 'most informative' attribute $A$ with values
    $\{v_1, \ldots, v_n\}$ for the root of the tree.
  Split $\mathcal{E}_{cur}$ into subsets $\mathcal{E}_1, \ldots, \mathcal{E}_n$ according to the values
    $v_1, \ldots, v_n$ of attribute $A$.
  for $i = 1$ to $1 = n$ do
    Recursively build a subtree $T_i$ for $\mathcal{E}_i$.
  endfor

Output: Decision tree $T$ with the root $A$ and subtrees
  $T_1, \ldots, T_n$ as shown in Figure 5.3.
```

Ideally, each leaf is labeled by exactly one class name. However, leaves can also be empty (NULL leaves), or can be labeled by more than one class name (SEARCH leaves). If there are no training examples having attribute values that would lead to a particular leaf then this is

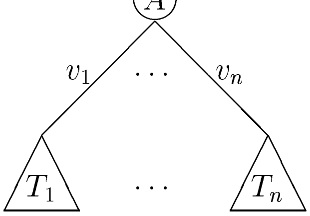


Figure 5.3: Decision tree built by ID3 with attribute A in the root.

a NULL leaf. SEARCH leaves appear when there are identical training examples with different class names.

At each step of the ID3 algorithm, the search heuristic chooses the 'most informative' attribute based on an information-theoretic measure called *entropy*. Let $E(\mathcal{E})$ be the *entropy* of the training set $\mathcal{E}$, i.e., the amount of information needed for the classification of one object [Shanon and Weaver 1964]:

$$E(\mathcal{E}) = -\sum_{j=1}^{N} p_j \cdot \log_2 p_j \tag{5.1}$$

where p_j is the prior probability of class C_j, $j \in \{1, \ldots, N\}$. Usually, p_j is computed as the relative frequency of examples from C_j in $\mathcal{E}$, i.e., $p_j = \frac{n^{C_j}}{n}$, where $n = |\mathcal{E}|$ and n^{C_j} is the number of examples from $\mathcal{E}$ of class C_j. Note that other (better) probability estimates can be used instead of the relative frequency (see Section 8.6 which gives an overview of probability estimates).

The entropy of the training set $\mathcal{E}$, after splitting it according to the values $v_1, \ldots, v_n$ of attribute A equals:

$$E(\mathcal{E}|A) = -\sum_{i=1}^{n} p_i \cdot \sum_{j=1}^{N} \frac{p_{ij}}{p_i} \cdot \log_2 \frac{p_{ij}}{p_i} \tag{5.2}$$

where p_i is the prior probability that A has value v_i, and p_{ij} is the prior probability that an example belonging to class C_j has value v_i of attribute A.

The *informativity* of attribute A is a measure aimed at minimizing the number of tests needed for the classification of a new object. It is defined as:

$$I(A) = E(\mathcal{E}) - E(\mathcal{E}|A) \tag{5.3}$$

At each step of the algorithm, the most informative attribute is the one which maximizes $I(A)$, or equivalently, minimizes $E(\mathcal{E}|A)$.

In ASSISTANT [Cestnik et al. 1987], the basic ID3 learning algorithm is extended in several ways which include: handling incompletely specified training examples, binarization of continuous attributes, binary construction of decision trees, pruning of unreliable parts of the tree and plausible classification based on the 'naive' Bayesian principle.

One of the most important features is tree pruning, used as a mechanism for handling noisy data. Tree pruning is aimed at producing trees which do not overfit possibly erroneous data. In tree pruning, the unreliable parts of a tree are eliminated in order to increase the classification accuracy of the tree on unseen cases.

There are two types of pruning:

- *pre-pruning*, performed during the construction of a decision tree, which decides whether to stop or to continue tree construction at a certain node, and
- *post-pruning*, employed after the decision tree has been constructed, which estimates the expected classification errors in the nodes of the tree and then decides whether to prune the subtrees rooted at those nodes.

The pruning techniques are based on the heuristic called the expected classification accuracy, or alternatively, the expected classification error estimate, mentioned in Section 3.2.2 and described in more detail in Section 8.4. Experiments in different domains have shown that the features enhancing ID3, as implemented in ASSISTANT, generate decision trees which are smaller, simpler and more comprehensible, and at the same time more accurate when classifying new objects [Cestnik et al. 1987, Cestnik and Bratko 1991].

5.2.3 NEWGEM

NEWGEM [Mozetič 1985b] is a descendant of the AQ7-AQ11 series of learning systems [Michalski 1969, 1983] and was further developed into AQ15 [Michalski et al. 1986] and AQ17 [Bloedorn and Michalski 1991].

Both AQ and NEWGEM induce hypotheses in the form of if-then rules. Descriptions are learned separately for each individual class. Objects from a given class are considered its positive examples, and all other objects are negative examples of the concept. Training examples are described as tuples of attribute values. Attributes can be of three types: nominal, linear or structured. Concept descriptions in the form of if-then rules are represented in the VL1 notation (Variable-valued Logic 1 [Michalski 1974]).

In the VL1 notation, a *selector* relates an attribute to a value or to a disjunction of values, e.g., $[Smiling = yes]$ and $[Holding = balloon \vee flag]$, respectively. A *complex* is a conjunction of selectors, e.g., $[Smiling = yes] \wedge [Holding = balloon \vee flag]$. A hypothesis is a set of if-then rules of the form

$$Class = ClassValue \quad \textbf{if} \quad Cover$$

where $ClassValue$ denotes one of the decision classes, and $Cover$ is a a disjunction of complexes covering all positive examples and none of the negative examples of the class. NEWGEM induces the following hypothesis from the examples given in Table 5.1.

$$\begin{array}{lll} Class = friendly & \textbf{if} & [Smiling = yes] \wedge [Holding = balloon \vee flag] \\ Class = unfriendly & \textbf{if} & [Smiling = no] \bigvee \\ & & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

In order to make the terminology more uniform with the rest of the book, we will use the term *condition* instead of selector, the term *body* instead of complex, and a rule with a disjunction of two complexes

$$\begin{array}{lll} Class = unfriendly & \textbf{if} & [Smiling = no] \bigvee \\ & & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

will be written in the form of two rules

$$\begin{array}{lll} Class = unfriendly & \textbf{if} & [Smiling = no] \\ Class = unfriendly & \textbf{if} & [Smiling = yes] \wedge [Holding = sword] \end{array}$$

It should be noted that **if** denotes a relation which is not necessarily implication. Whether the interpretation of **if** as implication is correct depends on the training data [Mozetič and Lavrač 1988]. In the original VL1 notation the symbol $\Leftarrow$ is used; the direction of the arrow ($\Leftarrow$ or $\Rightarrow$) just indicates the natural direction of inference when a rule of this form is used.

The AQ algorithm first introduced the so-called ‘covering’ approach. As mentioned earlier, in the case of N classes $C_1, \ldots C_N$, the task of learning the descriptions of N classes is first transformed into N learning tasks, one for each individual class C_i. For each task, examples of class C_i form $\mathcal{E}^+$ and examples belonging to all other classes form $\mathcal{E}^-$. Given the current training set of examples $\mathcal{E}_{cur}$ (initially set to the entire training set) and the current hypothesis $\mathcal{H}$ (which is most frequently initially set to the empty set of rules), the 'covering' loop of the NEWGEM algorithm constructs a hypothesis in three main steps:

- construct a rule,
- add the rule to the current hypothesis, and
- remove from the current training set the positive examples covered by the rule.

This loop is repeated until all the positive examples are covered. Algorithm 5.2 implements this loop.

Algorithm 5.2 (NEWGEM – the covering algorithm)

```
Initialize $\mathcal{H} := \emptyset$.
Initialize $\mathcal{E}_{cur} := \mathcal{E}^+ \cup \mathcal{E}^-$.
repeat
  Select an uncovered positive example Seed.
  Call the $BeamSearchAlgorithm(\mathcal{E}_{cur}, Seed, ClassValue)$
    to find the best body BestBody.
  $\mathcal{H} := \mathcal{H} \cup \{Class = ClassValue$ if $BestBody\}$,
    i.e., add the best rule to $\mathcal{H}$.
  Remove from $\mathcal{E}_{cur}$
    the positive examples covered by BestBody.
until $\mathcal{E}^+_{cur} = \emptyset$, i.e., $\mathcal{H}$ is complete.

Output: Hypothesis $\mathcal{H}$.
```

AQ starts hypothesis construction from an empty hypothesis or, alternatively, from an initial hypothesis supplied by the user. It generates a hypothesis in steps, each step producing one rule of the hypothesis. The algorithm selects a positive example (called a *Seed*) and starts with the maximally general rule (one with all possible values for all attributes). The body of the rule is then specialized so that it does not cover any negative example, but still covers the *Seed*. The algorithm keeps several alternative rules which maximize the number of covered positive examples and are as simple as possible. The set of alternative rule bodies is called a beam (a *star* in the AQ terminology). When no negative example is covered, the algorithm selects the best rule and adds it to the current hypothesis. The algorithm repeats the procedure until all positive examples are covered.

The procedure for generating a rule (body) to be added to the current hypothesis, invoked in the second step of the **repeat** loop of Algorithm 5.2, is given in Algorithm 5.3.

Algorithm 5.3 (NEWGEM – the beam search algorithm)

```
Initialize $Beam := \{true\}$.
while Any $Body$ in $Beam$ covers negative examples do
  Select a covered negative example.
  Specialize all inconsistent $Bodies$ to uncover the negative
    example, but still cover the $Seed$.
  Keep only the best $BeamSize$ bodies in $Beam$
    i.e., remove the worst bodies if $BeamSize$ is exceeded.
endwhile
Select $BestBody$ from $Beam$
  according to user-defined preference criteria.

Output: $BestBody$.
```

In the initial maximally general rule body, all attributes have all possible values. *Specialization* of a rule body with respect to the *Seed* (that has to be covered) and a negative example (that should not be covered) is done by removing some values from an internal disjunction of values of individual attributes so that the rule remains as general as possible. Notice that alternative specializations are possible which causes the combinatorial complexity of the algorithm.

The *measure of rule quality* determines a partial order of alternative rules and is a parameter (LEF – Lexical Evaluation Function) defined by the user. This is an essential part of the inductive bias used by the system. The measure consists of a list of criteria that are applied to each rule. When the first criterion cannot discriminate between two rules, the second one is used, and so on. The following is a list of default criteria that order rules from the best to worse:

- higher number of covered positive examples,
- lower number of conditions in the rule,
- lower total number of values in internal disjunctions.

The second user-defined parameter (called *BeamSize*) is the number of alternative rules to be considered by the algorithm, i.e., the width of the beam in beam search. Default value for this parameter is 10.

The AQ algorithm enables the generation of rules of different degrees of generality. Rules may be *general* (having the minimal number of attributes, each with the maximal number of disjunctive values), *minimal* (the minimal number of both attributes and values), or *specific* (the maximal number of attributes, each with minimal number of values).

When learning from inconsistent examples, i.e., examples having the same attribute values but belonging to different classes, the AQ algorithm provides three options:

- inconsistent examples are treated as positive examples for each class,
- inconsistent examples are treated as negative examples for each class,
- inconsistent examples are removed from the data.

NEWGEM and AQ15 have the incremental learning facility. The user can supply an initial hypothesis. The programs implement the method of learning with *full memory* (as opposed to learning with *partial memory*). In this type of learning, the program remembers all the training examples seen so far, as well as the rules it formed. New rules are guaranteed to be consistent and complete with respect to all (old and new) training examples [Reinke and Michalski 1988].

5.2.4 CN2

The rule induction system CN2 [Clark and Niblett 1989] combines the ability to cope with noisy data of the algorithms of the TDIDT family with the if-then rule form and the flexible search strategy of the AQ family. It has retained the covering approach and the beam search mechanism from AQ, but has removed its dependency on specific examples (*Seed*) during the search. Furthermore, in order to deal with noisy data, it has extended the search space to rules that do not perform perfectly on the training data. Initially, CN2 used an entropy-based search heuristic to induce ordered rules [Clark and Niblett 1989]. It has recently been extended with the ability to induce unordered rules and to use Bayesian accuracy estimates as search heuristics [Clark and Boswell 1991, Džeroski et al. 1992a]. The heuristics used for handling noisy data are described in detail in Section 8.3.

When learning if-then rules, objects of a given class are labeled $\oplus$ and treated as positive examples $\mathcal{E}^+$, and all other objects are treated as negative examples $\mathcal{E}^-$ of the selected class. Given the current training set of examples $\mathcal{E}_{cur}$ (initially set to the entire training set) and the current hypothesis $\mathcal{H}$ (initially set to the empty set of rules), the outer loop of the CN2 algorithm [Clark and Boswell 1991] for inducing unordered rules (Algorithm 5.4) implements the 'covering' algorithm outlined in Section 5.2.3.

Algorithm 5.4 is an outline of the CN2 covering algorithm, working for a particular class $ClassValue$.

The best body is chosen according to the search heuristic which measures the expected classification accuracy of the rule, estimated by the Laplace probability estimate [Clark and Boswell 1991].

Let N be the total number of classes in the problem. Let $n^{\oplus}(c)$ be the number of positive examples of the class $ClassValue$ and $n(c)$ the total number of examples in the current training set $\mathcal{E}_{cur}$, covered by a rule c. The Laplace classification accuracy estimate $A(c)$ estimates the probability that an example covered by rule c is positive as:

$$A(c) = p(\oplus|c) = \frac{n^{\oplus}(c) + 1}{n(c) + N}$$

In its previous version, instead of the Laplace estimate, CN2 was using an entropy-based heuristic [Clark and Niblett 1989].

Algorithm 5.4 (CN2 – the covering algorithm)

```
Initialize $\mathcal{H} := \emptyset$.
Initialize $\mathcal{E}_{cur} := \mathcal{E}^+ \cup \mathcal{E}^-$.
repeat
  Call the $BeamSearchAlgorithm(\mathcal{E}_{cur}, ClassValue)$ to find
    the $BestBody$ of a rule.
  if $BestBody \neq not\_found$
  then
    Add the best rule to $\mathcal{H}$, i.e.,
      $\mathcal{H} := \mathcal{H} \cup \{Class = ClassValue$ if $BestBody\}$.
    Remove from $\mathcal{E}_{cur}$
      the positive examples covered by $BestBody$.
until $BestBody = not\_found$.

Output: Hypothesis $\mathcal{H}$.
```

The beam search for the best body of a rule in Algorithm 5.5 proceeds in a top-down fashion. The search starts with the most general body (*true*), which covers all the training examples. At each step, a set of candidates (*Beam*) for the best body, as well as the best body found so far are kept. To specialize a body, a condition of the form $X = a$ is added to it as an additional conjunct. All the possible specializations of the candidates from *Beam* are considered. The best body found so far is accordingly updated and the *BeamSize* most promising candidates among the newly generated specializations are chosen.

The best body found so far is, in addition, tested for its statistical significance. This is to ensure that it represents a genuine regularity in the training examples and not a regularity which is due to chance. In this way, the undesirable bias of the entropy and apparent accuracy metrics used to estimate the quality of a clause is, at least partly, avoided. For a detailed treatment of the heuristics in CN2 and their role in handling noisy data, we refer the reader to Section 8.3.

Algorithm 5.5 (CN2 – the beam search algorithm)

```
Initialize $Beam := \{true\}$.
Initialize $NewBeam := \emptyset$.
Initialize $BestBody := not\_found$.
while $Beam \neq \emptyset$ do
  for each $Body$ in $Beam$ do
    for each possible specialization $Spec$ of $Body$ do
      if $Spec$ is better than $BestBody$ and
         $Spec$ is statistically significant
      then
         $BestBody := Spec$.
         Add $Spec$ to $NewBeam$.
      if Size of $NewBeam > BeamSize$
      then remove worst body from $NewBeam$.
    endfor
  endfor
  $Beam := NewBeam$.
endwhile

Output: $BestBody$.
```

5.3 Using background knowledge in learning

This section shows how background knowledge can be effectively applied to induce compact hypotheses in both the attribute-value and the ILP setting.

5.3.1 Using background knowledge in attribute-value learning

The method for using background knowledge in attribute-value learning is outlined on the robots example from Section 5.2.1. It is shown that the induced hypothesis can include new terms, given as background knowledge about the world of robots. Background knowledge can be expressed in the form of *functions* of attribute values or *relations* among attribute values. In learning attribute descriptions, these functions/relations give rise to new attributes which are considered in the learning process. If the background knowledge is represented in the form of functions, the value of a new attribute is computed as a function of the values of existing attributes, in turn for each training example. The range of values for the function is either a finite set of discrete values or an interval of real numbers. On the other hand, if the background knowledge has the form of relations, the only values the new attribute can have are *true* and *false* (if the values of the corresponding attributes of the example do/do not satisfy the relation). In other words, relations are boolean functions.

For example, background knowledge can check for the equality of attribute values for pairs of attributes of the same type (i.e., attributes with the same set of values). In the world of robots from Section 5.2.1 this would lead to two new attributes that test the equalities $Smiling = Has\_tie$ and $Head\_shape = Body\_shape$. For simplicity, let us consider only the new attribute $Head\_shape = Body\_shape$, named *Same_shape*, the values of which are *true* and *false*. Using this idea, initially introduced by Mozetič [1987], an extended set of tuples is generated and used in learning. The set of attribute-value tuples for the world of robots is given in Table 5.2, where *f* stands for the class *friendly* and *u* for *unfriendly*.

Since its two values *true* and *false* completely distinguish between the friendly and unfriendly robots, the new attribute *Same_shape* is the only attribute in the decision tree induced by ASSISTANT. The tree is shown in Figure 5.4.

	Attributes and values					
Class	*Smiling*	*Holding*	*Has_tie*	*Head_shape*	*Body_shape*	*Same_shape*
f	*yes*	*balloon*	*yes*	*square*	*square*	*true*
f	*yes*	*flag*	*yes*	*octagon*	*octagon*	*true*
u	*yes*	*sword*	*yes*	*round*	*octagon*	*false*
u	*yes*	*sword*	*no*	*square*	*octagon*	*false*
u	*no*	*sword*	*no*	*octagon*	*round*	*false*
u	*no*	*flag*	*no*	*round*	*octagon*	*false*

Table 5.2: Examples of friendly and unfriendly robots. The last column gives the values of the new attribute *Same_shape*.

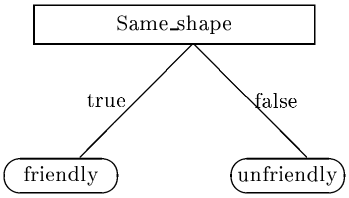


Figure 5.4: Decision tree built by using background knowledge.

If the examples from Table 5.2 are given to NEWGEM, the following if-then rules are induced:

$$\begin{array}{lll} Class = friendly & \textbf{if} & [Same\_shape = true] \\ Class = unfriendly & \textbf{if} & [Same\_shape = false] \end{array}$$

The example shows that new attributes, expressing functions of (relations among) the original attributes that describe the examples, can be more informative than the original attributes. Using background knowledge, a learner can build more compact hypotheses, which can potentially have a better classification accuracy on unseen cases.

5.3.2 Transforming ILP problems to propositional form

The method outlined in Section 5.3.1 is based on the idea that the use of background knowledge can introduce new attributes for learning. This same method can also be used in the ILP framework. To do so, the learning problem is transformed from relational to attribute-value form and solved by an attribute-value learner. This approach is feasible only for a restricted class of ILP problems.

The algorithm which solves ILP problems by transforming them into propositional form consists of the following three steps:

- The learning problem is transformed from relational to attribute-value form.
- The transformed learning problem is solved by an attribute-value learner.
- The induced hypothesis is transformed back into relational form.

In this section, the hypothesis language is restricted to function-free program clauses which are typed, constrained and non-recursive, i.e., to function-free constrained DHDB clauses (see the definitions in Section 2.1). The above algorithm allows for a variety of approaches developed for propositional problems, including noise-handling techniques in attribute-value algorithms, such as ASSISTANT [Cestnik et al. 1987] or CN2 [Clark and Niblett 1989], to be used for learning relations.

The algorithm is illustrated on a simple ILP problem of learning family relationships, similar to the one in Section 1.4. The task is to define the target relation $daughter(X, Y)$, which states that person X is a daughter of person Y, in terms of the background knowledge relations $female$, $male$ and $parent$. All the variables are of type $person$. The type $person$ is defined as $person = \{ann,\ eve,\ pat,\ sue,\ tom\}$. There are two positive and two negative examples of the target relation. The training examples and the relations from the background knowledge are given in Table 5.3, which is similar to Table 1.1.

The first step of the algorithm, i.e., the transformation of the ILP problem into attribute-value form, is performed as follows. The possible applications of the background predicates on the arguments of the target relation are determined, taking into account argument types. Each such application introduces a new attribute. In our example, all variables are of the same type $person$. The corresponding attribute-value learning problem is given in Table 5.4, where f stands for $female$, m for $male$ and p for $parent$. The attribute-value tuples are generalizations (relative to the given background knowledge) of the individual facts about the target relation.

In Table 5.4, *variables* stand for the arguments of the target relation, and *propositional features* denote the newly constructed attributes

Training examples		*Background knowledge*		
$daughter(sue, eve).$	$\oplus$	$parent(eve, sue).$	$female(ann).$	$male(pat).$
$daughter(ann, pat).$	$\oplus$	$parent(ann, tom).$	$female(sue).$	$male(tom).$
$daughter(tom, ann).$	$\ominus$	$parent(pat, ann).$	$female(eve).$	
$daughter(eve, ann).$	$\ominus$	$parent(tom, sue).$		

Table 5.3: Another simple family relationships problem.

	Variables		*Propositional features*							
C	X	Y	$f(X)$	$f(Y)$	$m(X)$	$m(Y)$	$p(X,X)$	$p(X,Y)$	$p(Y,X)$	$p(Y,Y)$
$\oplus$	*sue*	*eve*	*true*	*true*	*false*	*false*	*false*	*false*	*true*	*false*
$\oplus$	*ann*	*pat*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*
$\ominus$	*tom*	*ann*	*false*	*true*	*true*	*false*	*false*	*false*	*true*	*false*
$\ominus$	*eve*	*ann*	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*

Table 5.4: Propositional form of the *daughter* relationship problem.

of the propositional learning task. When learning function-free DHDB clauses, only the new attributes are considered for learning. If we remove the function-free restriction, the arguments of the target relation are used as attributes in the propositional task as well (see Section 5.4.4 for a more detailed explanation).

In the second step, an attribute-value learning program induces the following if-then rule from the tuples in Table 5.4:

$$Class = \oplus \quad \textbf{if} \quad [female(X) = true] \quad \wedge \quad [parent(Y,X) = true]$$

In the last step, the induced if-then rules are transformed into DHDB clauses. In our example, we get the following clause:

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

Note that the same result can be obtained on a similar learning problem, where the target relation $daughter(X,Y)$ is to be defined in terms of the relations $female$, $male$ and $parent$, given the background knowledge from Table 5.5. It illustrates that in this approach intensional background knowledge in the form of non-ground clauses can be used in addition to ground facts.

5.4 The LINUS algorithm

The method outlined in Section 5.3.2 is implemented in the ILP learner LINUS. Given the training examples $\mathcal{E}$ as ground facts defining the predicate p/n, the definitions of background knowledge predicates q_i/n_i and functions f_j/n_j in $\mathcal{B}$, the types of arguments of predicates p/n and q_i/n_i and functions f_j/n_j, and the language bias $\mathcal{L}$ of constrained DHDB clauses, the LINUS learning algorithm works as follows:

Background knowledge			
$mother(eve, sue).$ $mother(ann, tom).$ $father(pat, ann).$ $father(tom, sue).$	$parent(X, Y) \leftarrow$ $mother(X, Y).$ $parent(X, Y) \leftarrow$ $father(X, Y).$	$female(ann).$ $female(sue).$ $female(eve).$	$male(pat).$ $male(tom).$

Table 5.5: Intensional background knowledge for learning the *daughter* relationship.

Algorithm 5.6 (LINUS – learning constrained DHDB clauses)

1. Pre-process the training set to establish the sets $\mathcal{E}^+$ and $\mathcal{E}^-$ of positive and negative examples.
2. Using background knowledge $\mathcal{B}$, transform the facts from $\mathcal{E}^+$ and $\mathcal{E}^-$ from the DHDB form into attribute-value $Tuples$.
3. From $Tuples$ induce if-then $Rules$ by an attribute-value learner.
4. Transform $Rules$ into DHDB $Clauses$.
5. Post-process $Clauses$ to generate hypothesis $\mathcal{H}$.

Output: Hypothesis $\mathcal{H}$.

5.4.1 Pre-processing of training examples

Pre-processing of the training set involves negative example generation when negative examples are not given. It also involves the handling of missing argument values, described in Section 8.2.

The generation of negative facts in the first step of the LINUS algorithm takes into account the type theory and the 'closed-world assumption', which is a frequent assumption in different logic programming applications. LINUS provides for four options of negative example generation, an idea borrowed from QuMAS [Mozetič 1987].

- Negative facts can be given *explicitly*.

- When generating negative facts under the *closed-world assumption* (the *cwa* mode), all possible combinations of values of the n arguments of the target predicate are generated.

- Under the *partial closed-world assumption* (the *pcwa* mode), for a given combination of values of the n_{Ind} independent variables, all the combinations of values of the n_{Dep} dependent variables are generated.

- In the *near_misses* mode, facts are generated by varying only the value of one of the n variables at a time, where $n = n_{Dep} + n_{Ind}$.

The generated facts are used as negative instances of the target relation, except for those given as positive. Here we have to note that LINUS distinguishes between the so-called *dependent* and *independent* variables. Among the n arguments of the target predicate, n_{Dep} variables are treated as *dependent*, which is analogous to a *class* variable in attribute-value learning. The other n_{Ind} arguments are considered *independent* variables, analogous to *attributes* in attribute-value learning. This distinction is important, since LINUS treats dependent and independent variables differently in its two learning modes: learning definitions of relations (the *relation* learning mode) and learning the descriptions of individual classes (the *class* learning mode). Note that there can be more than one dependent variable in LINUS. In this case, a class value is a combination of values of all dependent variables.

In noisy and inexact domains, LINUS has to be run in the *class* learning mode or in the *relation* learning mode with *explicitly* given negative examples. The other three modes of negative examples generation are not appropriate. When one can not assume that the given training set is complete, the *cwa* negative examples generation mode is obviously not appropriate, as well as the *near_misses* mode. However, in some domains the *pcwa* mode could be used; then one has to be aware of the assumption that a given positive example belonging to one particular class does not belong to any other class.

In non-noisy exact domains, the *cwa* mode can be used. However, this mode is inappropriately used in two cases. The first case is when the set of positive examples is not exhaustive. The second case deals with the problem of representing the training examples since negative examples are generated in the ‘object’ space and not in the ‘relation’ space (see comments in Section 6.3).

5.4.2 Post-processing

As mentioned in Section 3.2.2, post-processing eliminates irrelevant literals from the clauses constituting a hypothesis in order to make the induced hypothesis more compact and more accurate when classifying new cases. In non-noisy exact domains a literal L in clause c is irrelevant if the clause c', obtained by eliminating L from c, does not cover any negative examples not covered by c.

In noisy domains, a literal L in a clause c is irrelevant if it can be removed from the body of the clause without decreasing the classification accuracy of the clause on the training set. Algorithm 5.7 eliminates irrelevant literals from a clause.

Algorithm 5.7 (Elimination of irrelevant literals)

```
Initialize $c' := c$.
repeat
   if $A(c) \leq A(c')$ then $c := c'$.
   Compute accuracy estimate $A(c)$ of clause $c$.
   for each literal $L$ in clause $c$ do
      Generate new clause $c_L$
         by eliminating literal $L$ from the body of $c$.
      Compute the accuracy estimate $A(c_L)$.
      if $A(c') \leq A(c_L)$ then $c' := c_L$.
   endfor
until $c = c'$
```

If a clause covers $n^{\oplus}(c)$ examples of the correct class and $n^{\ominus}(c)$ examples of incorrect classes, its expected classification accuracy is estimated in LINUS by the formula proposed in Quinlan [1987a]:

$$A(c) = \frac{n^{\oplus}(c) - 0.5}{n(c)}$$

where $n(c) = n^{\oplus}(c) + n^{\ominus}(c)$. This estimate was used in the experiments of Chapter 11. This is just one of the possible formulas for computing the expected accuracy $A(c)$. Recall that other (better) accuracy estimates, outlined in Chapter 8, could be used instead, e.g., the Laplace estimate [Niblett and Bratko 1986] and the m-estimate [Cestnik 1990].

5.4.3 An example run of LINUS

Let us illustrate in more detail the work of LINUS on a simple example. Consider the problem of learning illegal positions on a chess board with only two pieces, white king and black king. The target relation $illegal(WKf, WKr, BKf, BKr)$ states that the position in which the white king is at (WKf, WKr) and the black king at (BKf, BKr) is illegal. Arguments WKf and BKf are of type *file* (with values a to h), while WKr and BKr are of type $rank$ (with values 1 to 8). Background knowledge is represented by two symmetric predicates $adj\_file(X, Y)$ and $adj\_rank(X, Y)$ which can be applied on arguments of type $file$ and $rank$, respectively (the abbreviation *adj* stands for $adjacent$). The built-in symmetric predicate equality $X = Y$, which works on arguments of the same type, may also be used in the induced clauses.

A description of the individual steps of the algorithm follows.

1. First, in data pre-processing, the sets of positive and negative facts are established. In our domain, given are one positive example (illegal endgame position, labeled $\oplus$) and two negative examples (legal endgame positions, labeled $\ominus$):

$$\begin{array}{rl} \multicolumn{2}{c}{illegal(WKf, WKr, BKf, BKr).} \\ illegal(a, 6, a, 7). & \oplus \\ illegal(f, 5, c, 4). & \ominus \\ illegal(b, 7, b, 3). & \ominus \end{array}$$

2. The given facts are transformed into attribute-value tuples. The algorithm first determines the possible applications of the background predicates on the arguments of the target relation, taking into account argument types. Each such application is considered as an attribute.

In our example, the set of attributes determining the form of the tuples is the following:

⟨ *WKf=BKf, WKr=BKr, adj_file(WKf,BKf), adj_rank(WKr,BKr)* ⟩

Since predicates adj_file and adj_rank are symmetric, their other possible applications $adj\_file(BKf, WKf)$ and $adj\_rank(BKr, WKr)$ are not considered as attributes for learning.

It should be noted that the arguments of the target predicate, WKf, WKr, BKf and BKr, may be included in the selected set of attributes for learning, so that the form of the tuples would be as follows: ⟨ *WKf, WKr, BKf, BKr, WKf=BKf, WKr=BKr, adj_file(WKf,BKf), adj_rank(WKr,BKr)* ⟩. However, knowing that that the illegality of a chess position does not depend on the particular positions of the pieces, but rather their relative positions, we have excluded these arguments from the transformed tuples. In addition, in our experiments, this facilitates the comparison with FOIL, the concept description language of which does not allow for literals of the type $X = value$.

The tuples, i.e., the values of the attributes, are generated by calling the corresponding predicates with argument values from the ground facts of predicate *illegal*. In this case, the attributes can take values *true* or *false*. For the given examples, the following tuples are generated:

⟨ *WKf=BKf, WKr=BKr, adj_file(WKf,BKf), adj_rank(WKr,BKr)* ⟩

$$\begin{array}{rc} \langle true, false, false, true\rangle & \oplus \\ \langle false, false, false, true\rangle & \oplus \\ \langle true, false, false, false\rangle & \ominus \end{array}$$

These tuples are generalizations (relative to the given background knowledge) of the individual facts about the target relation. For example, the first tuple actually has the following meaning:

$$\begin{array}{lll} \forall A \forall B \forall C \forall D : & & \\ illegal(A, B, C, D) & \textbf{if} & A = C \wedge not\ (B = D) \wedge \\ & & not\ adj\_file(A, C) \wedge adj\_rank(B, D). \end{array}$$

3. Next, an attribute-value learning program is used to induce a set of if-then rules. When ASSISTANT is used, the induced decision trees are transcribed into the if-then rule form. NEWGEM induces the

following rule from the above three tuples:

$$Class = \oplus \ \ \textbf{if} \ \ (WKf = BKf) = true \wedge adj\_rank(WKr, BKr) = true$$

4. Finally, the induced if-then rules for $Class = \oplus$ are transformed into DHDB clauses. In our example, we get the following clause:

$$\begin{array}{lcl} illegal(WKf, WKr, BKf, BKr) & \leftarrow & WKf = BKf, \\ & & adj\_rank(WKr, BKr). \end{array}$$

5. In post-processing, no irrelevant literal is found and the hypothesis remains unchanged.

In summary, the learning problem is transformed from a relational to an attribute-value form and solved by an attribute-value learner. The induced hypothesis is then transformed back into relational form.

5.4.4 Language bias in LINUS

Before giving the restrictions imposed by the hypothesis language $\mathcal{L}$ of deductive hierarchical database clauses in LINUS, it is important to consider the form of training examples $\mathcal{E}$ and, in particular, predicates in $\mathcal{B}$ which actually determine the hypothesis space. Training examples have the form of ground facts (which may contain structured, but non-recursive terms) and background knowledge has the form of deductive database clauses (possibly recursive). Variables are typed.

Background knowledge

In LINUS, predicate definitions in the background knowledge $\mathcal{B}$ are of two types:

Utility functions Utility functions f_j/n_j are annotated predicates; mode declarations specify the *input* and *output* arguments, similar to mode declarations in GOLEM and FOIL2.0 [Quinlan 1991]. When applied to ground input arguments from the training examples, utility functions compute the unique ground values of their output arguments. When used in an induced clause, output variables must be bound to constants.

Utility predicates Utility predicates q_i/n_i have only *input* arguments and can be regarded as boolean utility functions having values $true$ or $false$ only.

A reduction of the hypothesis space is achieved by taking into account the pre-specified types of predicate arguments and by exploiting the fact that some utility predicates are symmetric:

Symmetric predicates Utility predicates can be declared *symmetric* in a set of arguments of the same type. For example, a binary predicate $q_i(X, Y)$ is symmetric in $\{X, Y\}$ if X and Y are of the same type, and $q_i(X, Y) = q_i(Y, X)$ for every value of X and Y. A built-in symmetric utility predicate *equality (=/2)* is defined by default on arguments of the same type.

Note that the complexity analysis in Section 5.5 (equation (5.6)) does not take into account the symmetry of predicates and the use of utility functions f_j.

Hypothesis language

In the current implementation of LINUS, the selected hypothesis language $\mathcal{L}$ is restricted to constrained deductive hierarchical database (DHDB) clauses. In DHDB, variables are typed and recursive predicate definitions are not allowed. In addition, all variables that appear in the body of a clause have to appear in the head as well, i.e., only constrained clauses are induced.

To be specific, the body of an induced clause in LINUS is a conjunction of literals, each having one of the following four forms:

1. a binding of a variable to a value, e.g., $X = a$;
2. an equality of pairs of variables, e.g., $X = Y$;
3. an atom with a predicate symbol (utility predicate) and input arguments which are variables occurring in the head of the clause, e.g., $q_i(X, Y)$; and
4. an atom with a predicate symbol (utility function) having as input arguments variables which occur in the head of the clause, and

output arguments with an instantiated (computed) variable value, e.g., $f_j(X, Y, Z), Z = a$.

In the above, X and Y are variables from the head of the clause, and a is a constant of the appropriate type. Literals of form (2) and (3) can be either positive or negative. Literals of the form $X = a$ under items (1) and (4) may also have the form $X > a$ and/or $X < a$, where a is a real-valued constant.

The attributes given to propositional learners are (1) the arguments of the target predicate, (2)–(3) binary-valued attributes resulting from applications of utility predicates and (4) output arguments of utility functions. Attributes under (1) and (4) may be either discrete or real-valued. For cases (2) and (3) an attribute-value learner will use conditions of the form $A = true$ or $A = false$ in the induced rules, where A is an attribute (cf. examples in Sections 5.3.2 and 5.4.3). These are transcribed to literals A and *not* A, respectively, in the DHDB clauses. For case (1) an attribute-value system will use conditions of the form $X = a$, $X > a$ or $X < a$, which can be immediately used in DHDB clauses. For case (4), in addition to the conditions $Z = a$, $Z > a$ and $Z < a$, the literal $f_j(X, Y, Z)$ has to be added to the DHDB clause so that the value of Z can be computed from the arguments of the target predicate.

To guide induction, LINUS uses meta-level knowledge which can exclude any of the above four cases, thus reducing the search space. For example, if only case (1) is retained, the hypothesis language is restricted to an attribute-value language. This can be achieved by using only the arguments of the target relation as attributes for learning. Cases (2)–(4), on the other hand, result from the use of predicates/functions from $\mathcal{B}$ as attributes for learning. The *equality* predicate *(=/2)*, which generates literals of the form (2), is built-in in LINUS.

The complexity of learning with LINUS is analyzed in Section 5.5 and shows the number of new attributes leading to literals of form (1)–(3) above. A detailed analysis of the actual hypothesis space, including the number of attributes which result in literals of form (4), can be found in Chapter 7 [Džeroski and Lavrač 1992].

Language biases for two typical learning tasks

As mentioned above, meta-level knowledge can be used to adjust the language bias in LINUS, which is done by the user according to the type of learning task at hand. Below we list two types of tasks with the appropriate settings of the language bias:

Class learning mode As an attribute-value learner, LINUS is used to induce descriptions of individual classes. In this case, the classes can be different from $\oplus$ and $\ominus$ and are determined by the values of a selected argument (or a set of arguments) of the target relation. The induced clauses have the form

$$Class = ClassValue \;\; \textbf{if} \;\; L_1, \ldots, L_m$$

where $ClassValue$ is a class name and literals L_i can take any of the four forms outlined in Section 5.4.4. In the experiments in learning of medical diagnostic rules, described in Chapter 11, LINUS was run in the class learning mode.

Relation learning mode As an ILP learner, LINUS induces constrained DHDB clauses of the form

$$p(X_1, \ldots, X_n) \leftarrow L_1, \ldots, L_m$$

where p is the name of the target predicate and literals L_i have any of the forms (1)–(4) above. In the experiments with LINUS described in Chapter 6, LINUS was used in the relation learning mode, learning function-free clauses; thus, only literals of form (2) and (3) were actually used.

To summarize, the language bias in LINUS is declarative. The user can set the language bias to any of the above forms, depending on the learning task at hand.

5.5 The complexity of learning constrained DHDB clauses

To analyze the complexity of the ILP learning task obtained by using background knowledge in the learning process, let us consider the number of attributes in the transformed learning task [Lavrač et al. 1991a].

Strongly related to this issue is the search complexity in LINUS estimated by the branching factor of the refinement graph, discussed in Section 7.4.2. In this section, only background knowledge predicates q_i/n_i are considered in the analysis, whereas in Section 7.4.2 utility functions f_j/n_j are considered as well. We will illustrate the complexity formulae on the task of learning family relationships used in Section 5.3.1.

The total number of attributes k_{Attr} to be considered by a propositional learner equals

$$k_{Attr} = k_{Arg} + \sum_{i=1}^{l} k_{New,q_i} \tag{5.4}$$

where $k_{Arg} = n$ is the number of arguments of the target relation p (i.e., *variables* in Table 5.4), and k_{New,q_i} is the number of new attributes resulting from the possible applications of the background predicate q_i on the arguments of the target relation.

Under the function-free restriction, k_{Attr} is equal to the total number of new attributes resulting only from the possible applications of the l background predicates on the arguments of the target relation (i.e., *propositional features* in Table 5.4):

$$k_{Attr} = \sum_{i=1}^{l} k_{New,q_i} \tag{5.5}$$

Suppose that u is the number of distinct types of arguments of the target predicate p, u_i is the number of distinct types of arguments of the background predicate q_i, $n_{i,s}$ is the number of arguments of q_i that are of type $\mathcal{T}_s$ and k_{ArgT_s} is the number of arguments of target predicate p that are of type $\mathcal{T}_s$. Then k_{New,q_i} is computed by the following formula:

$$k_{New,q_i} = \prod_{s=1}^{u_i} (k_{ArgT_s})^{n_{i,s}} \tag{5.6}$$

The $n_{i,s}$ places for arguments of type $\mathcal{T}_s$ can be, namely, filled in $(k_{ArgT_s})^{n_{i,s}}$ ways independently from choosing the arguments of q_i which are of different types.

Since a relation where two arguments are identical can be represented by a relation of a smaller arity, the arguments of a background predicate can be restricted to be distinct. In this case the following formula holds:

$$k_{New,q_i} = \prod_{s=1}^{u_i} \binom{k_{ArgT_s}}{n_{i,s}} \cdot n_{i,s}! \tag{5.7}$$

To illustrate the above formulae on the simple example from Tables 5.3 and 5.4, observe that $q_1 = female$, $q_2 = male$ and $q_3 = parent$. As all arguments are of the same type $\mathcal{T}_1$ ($person$), $u_1 = u_2 = u_3 = 1$ and $k_{ArgT_1} = 2$. Since there is only one type, n_i can be used instead of $n_{i,s}$. In this notation, $n_1 = n_2 = 1$ and $n_3 = 2$. Thus, according to equation (5.6), $k_{New,q_1} = k_{New,q_2} = (k_{ArgT_1})^{n_1} = (k_{ArgT_1})^{n_2} = 2^1 = 2$. This means that there are two applications of each of the predicates $female$ and $male$, namely $female(X)$, $female(Y)$ and $male(X)$, $male(Y)$, respectively. Similarly, $k_{New,q_3} = (k_{ArgT_1})^{n_3} = 2^2 = 4$, the four applications of $parent/2$ being $parent(X,X)$, $parent(X,Y)$, $parent(Y,X)$ and $parent(Y,Y)$. Finally, $k_{Attr} = k_{New,q_1} + k_{New,q_2} + k_{New,q_3} = 2 + 2 + 4 = 8$.

If the built-in background predicate *equality (=/2)* is used, the number of its possible applications equals to:

$$k_{New,=} = \sum_{s=1}^{u} \binom{k_{ArgT_s}}{2} = \sum_{s=1}^{u} \frac{k_{ArgT_s} \cdot (k_{ArgT_s} - 1)}{2}$$

where k_{ArgT_s} denotes the number of arguments of the target predicate that are of the same type $\mathcal{T}_s$, and u is the number of distinct types of arguments of the target relation; from the arguments of type $\mathcal{T}_s$, $\binom{k_{ArgT_s}}{2}$ different attributes of the form $X = Y$ can be constructed.

Assuming a constant upper bound j on the arity of background knowledge predicates, the largest number of attributes that can be obtained in case that all variables are of the same type is of the order $\mathcal{O}(ln^j)$, where l is the number of predicates in the background knowledge and n is the arity of the target predicate. In other words, the size of the transformed learning task is polynomial in the arity n of the target predicate p and the number l of background knowledge predicates. This allows us to use PAC-learnability results for the propositional case in the ILP framework [Džeroski et al. 1992c].

5.6 Weakening the language bias

It was shown [Džeroski et al. 1992c, Lavrač and Džeroski 1992a] that the language bias in LINUS can be weakened, so that LINUS can learn clauses which introduce new variables. To allow for more expressiveness, an idea from GOLEM is borrowed, also used in FOIL2.0 [Quinlan 1991] to alleviate the problems with greedy search. While LINUS allows only 'old' variables from the head of a clause, the idea of determinacy allows for a restricted form of 'new' variables to be introduced into the learned clauses. The very type of restriction (determinacy) allows the LINUS transformation approach to be used in this case as well [Džeroski et al. 1992c, Lavrač and Džeroski 1992a].

By extending the transformation approach of the LINUS algorithm, the same class of logic programs can be learned as induced by GOLEM, namely the class of *determinate logic programs* defined below. In fact, the class of programs induced by the extended LINUS algorithm is slightly larger, as negated literals are allowed in the bodies of clauses. Based on this approach, some positive learnability results were proven for the class of determinate logic programs [Džeroski et al. 1992c] within the PAC-learnability framework of [Valiant 1984] and [Li and Vitányi 1991]. In this section, the language bias of i-determinate clauses is introduced and illustrated on an example. Without loss of generality [Džeroski et al. 1992c, Rouveirol 1992], we will assume a function-free hypothesis language.

5.6.1 The i-determinacy bias

Assuming that an integer constant j is given, only the class of ILP problems where all predicates in $\mathcal{B}$ are of arity at most j will be considered. The notion of *variable depth* is first defined.

Variable depth Consider a clause

$$p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, ..., L_r, ...$$

Variables that appear in the head of the clause have *depth* zero. Let a variable V appear first in literal L_r. Let d be the maximum depth of the other variables in L_r that appear in the clause

$p(X_1, X_2, \ldots, X_n) \leftarrow L_1, L_2, ..., L_{r-1}$. Then the *depth* of variable V is $d+1$.

By setting a maximum variable depth i, the syntactic complexity of clauses in the hypothesis language is restricted.

The following definition of *determinacy*, adapted to the case of function-free clauses, is taken from Džeroski et al. [1992c].

Determinacy A predicate definition is *determinate* if all of its clauses are determinate. A clause is *determinate* if each of its literals is determinate. A literal is *determinate* if each of its variables that do not appear in preceding literals has only one possible binding given the bindings of its variables that appear in preceding literals.

i-determinacy The determinacy restriction is called *i-determinacy* for a given maximum variable depth i.

Given the arity bound j, *i-determinacy* implies *ij-determination* as defined by Muggleton and Feng [1990]. Note that while i and j restrict the syntactic complexity of clauses, the notion of determinacy is essentially a semantic restriction, as it depends on the given training examples and background knowledge.

If we set the hypothesis language to the language of i-determinate clauses, the transformation approach of LINUS is still applicable, as demonstrated in Section 5.7. Determinate literals are a natural extension of the utility functions used presently in LINUS. In the light of the above definitions we can say that, in the current implementation of LINUS, only variables of depth 0 and 1 are allowed and that variables of depth 1 may not be used in other literals, except as described in Section 5.4.4.

The arity bound j is fixed. On the other hand, the parameter i can be increased to increase the expressiveness of the hypothesis language. For a fixed j, the user can consider the following series of languages:

$$\mathcal{L}_0 \text{ - } \mathcal{L}_1 \text{ - } \mathcal{L}_2 \text{ - } \ldots$$

where the expressiveness of $\mathcal{L}_i$ (and thus the complexity of the search space) grows with i. The language $\mathcal{L}_0$ is the language of constrained function-free DHDB clauses. If a solution for the ILP problem cannot be found in the selected language, the next language in the series may be chosen, and the learning process repeated until a complete and consistent solution is found. Along these lines, LINUS could, in principle, shift its bias dynamically, similarly to CLINT [De Raedt 1992] and NINA [Adé and Bruynooghe 1992].

However, there are several problems with shifting bias dynamically. First of all, the series of languages considered has to be in some sense complete. Consider, for example, the case where the target definition is not determinate (is not in any of the languages in the series). In this case the system may continue to shift its bias forever without finding a satisfactory solution. Next, the learning process is repeated for each language in the series, up to the appropriate one. This leads to less efficient learning, although some improvements are possible. The most important problem, however, is when and how to shift the bias if learning data is imperfect, which can easily be the case in empirical ILP. Therefore, we will rather assume a fixed language bias $\mathcal{L}_i$.

5.6.2 An example determinate definition

Let us illustrate the notion of i-determinacy on a simple ILP problem. The task is to define the predicate $grandmother$, where $grandmother(X, Y)$ states that person X is grandmother of person Y, in terms of the background knowledge predicates $father$ and $mother$. The training examples are given in Table 5.6. and background knowledge in Table 5.7.

Training examples	
$grandmother(ann, bob).$	$\oplus$
$grandmother(ann, sue).$	$\oplus$
$grandmother(bob, sue).$	$\ominus$
$grandmother(tom, bob).$	$\ominus$

Table 5.6: Training examples for learning the *grandmother* relationship.

Background knowledge		
$father(zak, tom)$.	$father(pat, ann)$.	$father(zak, jim)$.
$mother(ann, tom)$.	$mother(liz, ann)$.	$mother(ann, jim)$.
$father(tom, sue)$.	$father(tom, bob)$.	$father(jim, dave)$.
$mother(eve, sue)$.	$mother(eve, bob)$.	$mother(jean, dave)$.

Table 5.7: Background knowledge for learning the *grandmother* relationship.

A correct target predicate definition is:

$$\begin{array}{lcl} grandmother(X,Y) & \leftarrow & father(Z,Y), mother(X,Z). \\ grandmother(X,Y) & \leftarrow & mother(U,Y), mother(X,U). \end{array}$$

This hypothesis can be induced in a language which is at least as expressive as $\mathcal{L}_1$. The clauses are determinate (but not constrained), because each occurrence of a new variable (i.e., Z in $father(Z,Y)$ and U in $mother(U,Y)$) has only one possible binding given particular values of the other (old) variables in the literal (i.e., Y in this case); this is the case since each person has exactly one mother and father. The hypothesis is function-free and the maximum depth of any variable is one ($i = 1$). The arity bound j has to be at least two to allow induction of the above definition.

However, the logically equivalent hypothesis

$$\begin{array}{lcl} grandmother(X,Y) & \leftarrow & mother(X,Z), father(Z,Y). \\ grandmother(X,Y) & \leftarrow & mother(X,U), mother(U,Y). \end{array}$$

is not determinate, since the new variable Z in the literal $mother(X,Z)$ can have more than one binding for a fixed value of X (e.g., $X = ann$, $Z = tom$ or $Z = jim$); each person can namely have several children. We can thus see that the property of determinacy of a clause does not depend only on the given training examples and background knowledge, but also on the ordering of literals in the body of a clause.

5.7 Learning determinate clauses with DINUS

In this section, LINUS is treated exclusively as an ILP learner using only background knowledge in the form of predicates which may have both input and output arguments. We show how to weaken the language bias in LINUS and learn determinate DDB clauses. An algorithm that transforms the problem of learning non-recursive function-free i-determinate clauses into a propositional form, called DINUS (Determinate LINUS), is first presented and then illustrated using the example from Section 5.6.2. The back transformation of induced propositional concept descriptions to the program clause form is then briefly described and illustrated by an example. Finally, we discuss how the algorithm can be extended to learn recursive clauses. Dealing with non-recursive clauses is easier, as the recursive case may require querying the user (oracle) in order to complete the transformation process.

5.7.1 Learning non-recursive determinate DDB clauses

For simplicity, we first consider the problem of learning non-recursive clauses. The transformation of the ILP problem of constructing an i-determinate definition for target predicate $p(X_1, X_2, ..., X_n)$ from examples $\mathcal{E}$ and from predicate definitions in $\mathcal{B}$ is described below.

Like the original LINUS algorithm, the extended algorithm also consists of three main steps (the first step contains the most important differences to the the original LINUS algorithm, described in Sections 5.3.2 and 5.4). Note that, for simplicity, pre-processing of training examples and post-processing of hypotheses are not discussed.

- Transform the ILP problem to propositional form.
    1. Construct a list L of determinate literals that introduce new variables of depth at most i. Construct a list V_i of old variables and new variables introduced by determinate literals from L.
    2. Construct a list F of all literals that use predicates from the background knowledge and variables from V_i. The determinate literals from L are excluded from this list, as they do not discriminate between positive and negative examples (since the new variables have a unique binding for each example, due to the determinacy restriction). The resulting list is the list of features (attributes) used for propositional learning.

3. Transform the examples to propositional form. For each example, the truth value of each of the propositional features is determined by calls to the background knowledge $\mathcal{B}$.

- Apply a propositional learning algorithm to induce a propositional concept description.

- Transform the induced propositional description to a set of determinate DDB clauses, adding the necessary determinate literals which introduced the new variables.

DINUS – The transformation algorithm

Algorithm 5.8 describes the transformation of an ILP problem into propositional form, i.e., the first step of the above algorithm for learning non-recursive determinate clauses. The algorithm assumes that the learning task is to find a predicate definition as a set of determinate DDB clauses with $p(X_1, X_2, \ldots, X_n)$ in the head. Given are the training examples $\mathcal{E}$ as ground facts of the target predicate p/n, the language bias $\mathcal{L}_i$ of i-determinate DDB clauses, the definitions of background knowledge predicates q_s/n_s in $\mathcal{B}$, and the types of arguments of predicates p/n and q_s/n_s. The output of the algorithm is a set of examples $\mathcal{E}_f$ for attribute-value learning in the form of tuples f labeled $\oplus$ for $p(a_1, a_2, ..., a_n) \in \mathcal{E}^+$ and labeled $\ominus$ for $p(a_1, a_2, ..., a_n) \in \mathcal{E}^-$.

The knowledge base $\mathcal{B}$ of background predicates may take the form of a (normal) logic program. In step (3) of Algorithm 5.8 two types of queries have to be posed to this knowledge base.

- The first type are *existential* queries, which are used to determine the values of the new variables. Given a partially instantiated goal (literal containing variables that do not appear in previous literals), an existential query returns the set (possibly empty) of all bindings for the unbound variables which satisfy the literal. For example, the query $mother(X, A)$, where $X = ann$ and A is a new variable would return the set of answers $\{A = tom, A = jim\}$. Note that for determinate literals the set of answers is always a singleton.

- The other type of queries are ground (*membership*) queries about background knowledge predicates, where the goal is completely

Algorithm 5.8 (DINUS – the transformation algorithm)

```
Initialize the list of variables $V_0 := \{X_1, X_2, ..., X_n\}$.
Initialize the list of literals $L := \emptyset$.
Initialize the set of tuples $\mathcal{E}_f := \emptyset$.
1. for $r = 1$ to $i$ do
    $D_r := \{q_s(Y_1, Y_2, ..., Y_{j_s}) \mid q_s \in \mathcal{B},\ Y_1, Y_2, ..., Y_{j_s}$ are of the
      appropriate types, $q_s(Y_1, Y_2, ..., Y_{j_s})$ is determinate and contains at least one new variable not in $V_{r-1}$ $\}$.
    $L := L \cup D_r$.
    $V_r := V_{r-1} \cup \{Y \mid Y$ appears in a literal from $D_r\}$.
endfor
2. $F := \{q_s(Y_1, Y_2, ..., Y_{j_s}) \mid q_s \in \mathcal{B},\ Y_1, Y_2, ..., Y_{j_s} \in V_i\} - L$.
3. for each $p(a_1, a_2, ..., a_n) \in \mathcal{E}$ do
    Determine the values of variables in $V_i$ by executing the body
      of the clause $p(X_1, X_2, ..., X_n) \leftarrow L$ with variables $X_1, ..., X_n$
      bound to $a_1, ..., a_n$.
    Given the values of variables in $V_i$, determine the tuple $f$ of
      truth values of literals in $F$, by querying the background
      knowledge $\mathcal{B}$.
    $\mathcal{E}_f := \mathcal{E}_f \cup \{f\}$.
endfor
```

bound. These are used to determine the truth values of the propositional features.

In an actual implementation of Algorithm 5.8, steps (1) and (3) should be interleaved, i.e., the values of the new variables and the propositional features should be calculated for each example as they are introduced. In this way, the determinacy of literals in step (1) is automatically tested when the existential query determining the values of the new variables in a literal is posed. A literal is determinate if the set of answers to the existential query is a singleton for all the examples. If the set of answers for some example is not a singleton, the literal is not determinate.

The DINUS algorithm at work: An example

For the grandmother example, given in Tables 5.6 and 5.7 of Section 5.6.2, we have $i = 1$, $l = 2$, and $n = 2$. Let $j \geq 2$. The literal $father(X, A)$, where X is an old and A is a new variable, is not determinate (e.g., $X = tom$, $A = sue$ or $A = bob$). However, if A is old and X is new, the literal is determinate. As the target predicate is $grandmother(X, Y)$, we have $V_0 = \{X, Y\}$ and $D_1 = \{f(U, X), f(V, Y), m(W, X), m(Z, Y)\}$, where f and m stand for $father$ and $mother$, respectively.

This gives $L = D_1$, $V_1 = \{X, Y, U, V, W, Z\}$. The list F includes literals such as $f(X, X)$, $f(X, Y)$, $f(Z, Y)$, $f(W, X)$ and similarly $m(Z, Z)$, $m(V, Y)$, $m(W, W)$, $m(U, X)$, $m(X, V)$, $m(X, Z)$. In fact, the pairs of arguments of f and m are all the pairs from the Cartesian product $V_1 \times V_1$, excluding the pairs that produce literals from D_1.

The transformation process for the ILP problem as defined by the training examples from Table 5.6 and background knowledge from Table 5.7 is illustrated in Table 5.8. For the two positive and the two negative examples of $g(X, Y)$ (g stands for $grandmother$), given are the values of the old variables X and Y, the values of the new variables U, V, W and Z introduced by the determinate literals from the list L, and the values of two of the propositional features from the list F, namely $m(X, V)$ and $m(X, Z)$, denoted by A_1 and A_2, respectively. Note that only the propositional features (attributes $A_1, A_2, \ldots$) are actually considered in the attribute-value learning task.

	Variables		*New variables*				*Propositional features*			
$g(X,Y)$	X	Y	$f(U,X)$	$f(V,Y)$	$m(W,X)$	$m(Z,Y)$	...	$m(X,V)$	$m(X,Z)$	...
Class	X	Y	U	V	W	Z		A_1	A_2	
$\oplus$	*ann*	*bob*	*pat*	*tom*	*liz*	*eve*		*true*	*false*	
$\oplus$	*ann*	*sue*	*pat*	*tom*	*liz*	*eve*		*true*	*false*	
$\ominus$	*bob*	*sue*	*tom*	*tom*	*eve*	*eve*		*false*	*false*	
$\ominus$	*tom*	*bob*	*zak*	*tom*	*ann*	*eve*		*false*	*false*	

Table 5.8: Propositional form of the *grandmother* learning problem.

DINUS – Back transformation to DDB clause form

The algorithm that converts the induced propositional concept definition to DDB clause form is fairly straightforward. First, transcribe the induced rules into DDB clauses (as in the constrained case). Than add the necessary determinate literals, ensuring that the values of the new variables can be uniquely determined from the values of old variables. For each clause repeat the following process, until all variables in the body of the clause and not in its head are introduced by determinate literals: choose a new variable and add to the clause the literal in L that introduced the new variable. The literals are then ordered according to the maximum depth of variables occurring in them.

To illustrate this transformation to a set of DDB clauses, the grandmother example is used. Suppose that a propositional learner induces the following two rules from the examples in Table 5.8:

$$\begin{array}{lll} Class = \oplus & \textbf{if} & [A_1 = true] \\ Class = \oplus & \textbf{if} & [A_2 = true] \end{array}$$

This description is consistent with the examples, i.e., does not cover any negative example. (Note, however, that it is not too likely that any propositional learner would actually induce this description, since the first rule would suffice for discriminating between the positive and negative examples.) The two rules are transcribed to the clauses:

$$\begin{array}{lll} grandmother(X,Y) & \leftarrow & mother(X,V). \\ grandmother(X,Y) & \leftarrow & mother(X,Z). \end{array}$$

The new variables V and Z in literals $mother(X,V)$ and $mother(X,Z)$ are not introduced by determinate literals, so the literals $father(V,Y)$ and $mother(Z,Y)$ from L have to be added to the first and second clause, respectively. The final form of the logic program would then be:

$$\begin{array}{lll} grandmother(X,Y) & \leftarrow & father(V,Y), mother(X,V). \\ grandmother(X,Y) & \leftarrow & mother(Z,Y), mother(X,Z). \end{array}$$

Expressed in logic, this program stands for:

$$\begin{array}{ll} \forall X \forall Y: & grandmother(X,Y) \leftrightarrow \\ & [\exists V : f(V,Y) \wedge m(X,V)] \vee [\exists Z : m(Z,Y) \wedge m(X,Z)] \end{array}$$

or more precisely for:

$$\begin{aligned}\forall X \forall Y: \quad & grandmother(X,Y) \leftrightarrow \\ & [\exists! V: f(V,Y) \wedge m(X,V)] \vee [\exists! Z: m(Z,Y) \wedge m(X,Z)]\end{aligned}$$

as the value of each of the new variables is uniquely determined.

5.7.2 Learning recursive determinate DDB clauses

To learn recursive determinate definitions, Algorithm 5.8 needs to be slightly modified [Džeroski et al. 1992b]. Let us first distinguish between the case of recursive literals which do not and recursive literals which do introduce new variables. In the first case, membership queries are needed, and in the second, both membership and existential queries about the target relation are needed.

In the first case (when recursive literals do not introduce new variables), only steps (2) and (3) of Algorithm 5.8 need to be modified. In step (2), which constructs the list of literals F, we treat the target predicate p as any of the other background knowledge predicates and form all possible literals with it and the variables available, excluding the literal $p(X_1, X_2, ..., X_n)$. However, in step (3) we cannot evaluate features involving p by querying the background knowledge. In that case we first check whether the ground query is among the training examples for p, in which case we can determine its truth value (*true*, if the example is positive, *false* if negative). If, however, the query cannot be found among the examples, we have to resort to a membership query, that is, we must ask the user whether the answer to the query involved is true or false.

To illustrate the above modification to Algorithm 5.8, consider the task of learning the relation $member(X,Y)$, where X is of type *element* and Y is of type *list*. The background knowledge given includes the predicate $components(X,Y,Z)$, where X and Z are of type *list* and Y is of type *element*. This predicate is defined as $components([A|B],A,B)$ and decomposes a list $[A|B]$ into its head A and tail B. It is a flattened version of the *cons* function symbol. The equality predicate which works on arguments of the same type is also given. In this case j has to be at least 3. The maximum depth of variables is set to $i = 1$.

Table 5.9 gives four training examples and their transformation into propositional form. From the table we can see that the only determinate literal is $c(Y,A,B)$ (c and m stand for *components* and *member*, respec-

	Vars		*New vars*		*Propositional features*					
$m(X,Y)$	X	Y	$c(Y,A,B)$		$c(Y,X,B)$	$X=A$	$Y=B$	$m(X,B)$	$m(A,Y)$	$m(A,B)$
Class	X	Y	A	B	A_1	A_2	A_3	A_4	A_5	A_6
$\oplus$	1	$[1,2]$	1	$[2]$	*true*	*true*	*false*	*false*	*true*	*false*
$\oplus$	1	$[2,1]$	2	$[1]$	*false*	*false*	*false*	$true^*$	$true^*$	$false^*$
$\oplus$	2	$[2]$	2	$[]$	*true*	*true*	*false*	$false^*$	*true*	$false^*$
$\ominus$	1	$[2]$	2	$[]$	*false*	*false*	*false*	$false^*$	*true*	$false^*$

Table 5.9: Propositional form of a recursive ILP problem.

tively), and $V_1 = \{X, Y, A, B\}$, where X and A are of type *element* and Y and B are of type *list*. Taking into account the types of arguments, the modified algorithm produces the list of features in the table. The literals $c(Y,X,Y)$, $c(Y,A,Y)$, $c(B,X,B)$, $c(B,A,B)$, $c(B,X,Y)$ and $c(B,A,Y)$ are also on the list of features, but have been excluded for the sake of readability (they always have the value $false$). Consider the propositional feature A_4 which corresponds to the recursive call $member(X,B)$. The value of this feature can be determined for the first example, since the ground query is in this case $member(1,[2])$, which can be found as a negative training example. However, for the other three examples, membership queries have to be posed (the answers are marked with an asterisk *). A similar observation holds for features A_5 and A_6.

A propositional learner might generate the following rules from the given propositional examples:

$$\begin{array}{lll} Class = \oplus & \textbf{if} & [A_2 = true] \\ Class = \oplus & \textbf{if} & [A_4 = true] \end{array}$$

The rules would be transformed to the following definition:

$$\begin{array}{lll} member(X,Y) & \leftarrow & components(Y,A,B),\ =(X,A). \\ member(X,Y) & \leftarrow & components(Y,A,B),\ member(X,B). \end{array}$$

which is the correct definition of the concept $member(X,Y)$.

The second case occurs if recursive literals with new variables are allowed, which are, for example, necessary for learning the *quicksort* program. In addition to the changes outlined above, the first substep of step (1) of Algorithm 5.8 has also to be changed. In this step, the target predicate p is treated exactly as the other predicates. Determining the values of the new variables in literals involving the target predicate requires in this case the use of *existential* queries about the target concept.

6

Experiments in learning relations with LINUS

This chapter discusses the performance of LINUS on four relation learning tasks taken from the machine learning literature. The domain descriptions are taken from Quinlan [1990] and the results of LINUS are compared to the results obtained by FOIL [Quinlan 1990]. An early version of FOIL (FOIL0) was used in the experiments. All the domains contain non-noisy data. The first three domains involve a very small number of examples. In the fourth domain, the training set of chess endgame positions is large enough to enable the comparison of the LINUS results with the results of FOIL and other machine learning systems in terms of classification accuracy.

6.1 Experimental setup

LINUS was used in the *relation learning mode* (see Section 5.4.4). In order to facilitate the comparison with FOIL, no literals of the form $X = a$ instantiating a variable to a constant were allowed in the induced predicate definitions. This was achieved by selecting only applications of utility predicates as new attributes to be used for learning (see Section 5.4.4).

The arguments of the background predicates used in FOIL were not typed and the same predicates could be used for different types of arguments. In LINUS, each such predicate was replaced by several predicates, one for each combination of predicate argument types. For example, the *adjacent* relation in the chess endgame example was replaced by two relations *adjacent_rank* and *adjacent_file* (see Section 6.5).

In the experiments, ASSISTANT and NEWGEM were used within

LINUS. The results obtained with ASSISTANT and NEWGEM on each of the domains were typically slightly different; they were all comparable to the results obtained by FOIL. In the chess endgame domain, the classification accuracy and the computational efficiency of LINUS was compared with the available results for FOIL, DUCE [Muggleton 1987] and CIGOL [Muggleton et al. 1989].

6.2 Learning family relationships

The family relationships learning task is described in [Hinton 1989] and [Quinlan 1989]. Given are two stylized families of twelve members each, as shown in Figure 6.1.

LINUS was used to learn the relation $mother(A, B)$ from examples of this relation and the background relations $father(X, Y)$, $wife(X, Y)$, $son(X, Y)$ and $daughter(X, Y)$. Negative examples were generated under the closed-world assumption (the *cwa* mode), both for LINUS and FOIL. To illustrate the post-processing feature of LINUS, we first present the intermediate clauses induced by LINUS using ASSISTANT (before post-processing).

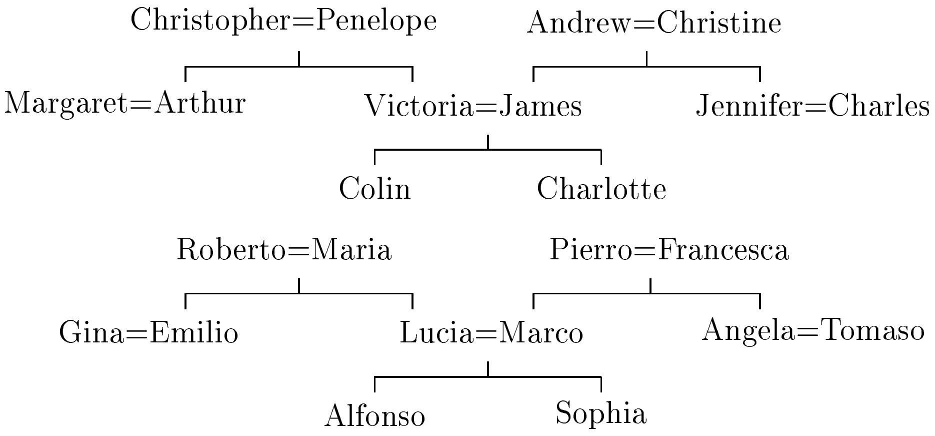


Figure 6.1: Two family trees, where = means *married to*, from Hinton [1989] and Quinlan [1989].

$$
\begin{array}{lcl}
mother(A,B) & \leftarrow & daughter(B,A), \\
 & & not\ father(A,B). \\
mother(A,B) & \leftarrow & not\ daughter(B,A), \\
 & & son(B,A), \\
 & & not\ father(A,B).
\end{array}
$$

The second clause contains the irrelevant literal $not\ daughter(B,A)$, which was removed by post-processing. The resulting predicate definition is equal to the definition induced by LINUS using NEWGEM which contained no irrelevant literals. A comparison of the post-processed clauses with the clauses induced by FOIL is given below.

LINUS	FOIL
negative examples: *cwa*	negative examples: *cwa*
$mother(A,B) \leftarrow$	$mother(A,B) \leftarrow$
$daughter(B,A),$	$daughter(B,A),$
$not\ father(A,B).$	$not\ father(A,C).$
$mother(A,B) \leftarrow$	$mother(A,B) \leftarrow$
$son(B,A),$	$son(B,A),$
$not\ father(A,B).$	$not\ father(A,C).$

We can see that FOIL generated a more specific hypothesis, and that the definitions induced by LINUS and FOIL are very similar and both correct. The hypothesis induced by LINUS can be paraphrased as follows: 'A is the mother of B if B is a child of A and A is not the father of B.' The hypothesis induced by FOIL is more specific (if negation is interpreted as in Prolog), since a new existentially quantified variable appears in the negated literal. It can be summarized as: 'A is the mother of B if B is a child of A and A is not the father of anybody (C).' The definition contains a new variable C which stands for any person. While not necessary in this case, new variables are necessary when learning definitions of relations such as $grandmother(X,Y)$ in terms of the relations $mother(X,Y)$ and $father(X,Y)$. Thus, the hypothesis language of FOIL is more expressive than the one of LINUS which does not allow for the introduction of new variables.

6.3 Learning the concept of an arch

In this example, taken from Winston [1975] and Quinlan [1990], four objects are given. Two of them are arches (positive examples) and two are not (negative examples), as shown in Figure 6.2.

For inducing the target relation $arch(A, B, C)$, stating that A, B and C form an arch with columns B and C and lintel A, the following background relations were used: $supports(X, Y)$, $left\_of(X, Y)$, $touches(X, Y)$, $brick(X)$, $wedge(X)$ and $parallelepiped(X)$.

LINUS was first run with explicitly given (two) negative examples, and then with the negative examples generated in the *cwa* mode. The results of LINUS and FOIL are listed below.

LINUS	FOIL
negative examples: explicitly	negative examples: explicitly
$arch(A, B, C) \leftarrow$ $supports(B, A)$, $not\ touches(B, C)$.	no clauses, because of the encoding length restriction
negative examples: *cwa*	negative examples: *cwa*
$arch(A, B, C) \leftarrow$ $left\_of(B, C)$, $supports(B, A)$, $not\ touches(B, C)$.	$arch(A, B, C) \leftarrow$ $left\_of(B, C)$, $supports(B, A)$, $not\ touches(B, C)$.

The hypothesis induced with explicitly given negative examples can be paraphrased as follows: 'A, B and C form an arch if B supports A and B does not touch C'. Note that this definition is more general

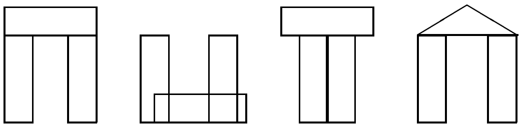


Figure 6.2: Arches and near misses, from Winston [1975] and Quinlan [1990].

than the one obtained in the *cwa* mode; the latter has an additional literal stating that B must be left of C.

With explicitly given negative examples, FOIL was unable to generate a hypothesis because of the encoding length restriction (described in Section 8.4) which restricts the total length of the hypothesis to the number of bits needed to enumerate the training examples explicitly. Using the *cwa* mode of negative examples generation, LINUS gave the same results as FOIL.

Analysis of the results

The hypothesis induced by both LINUS and FOIL covers also the case illustrated in Figure 6.3, which is a negative example of the arch concept. In the case when the negative examples are given explicitly this is reasonable, but it is counter-intuitive in the case where negative examples are generated under the closed-world assumption.

In order to get a more appropriate hypothesis, i.e., the same hypothesis as reported in Winston [1975], we have to be aware of the following problems:

- The problem of representation. Winston used a semantic net representation which is not equivalent to the relational representation selected in LINUS and FOIL. Some information was lost when transcribing the semantic net representation into the selected relations, e.g., the *part_of* relation which is essential in the semantic net representation was not included.

- Inappropriate use of the 'closed-world assumption' mechanism for generating negative examples.

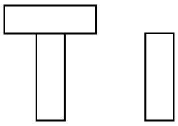


Figure 6.3: A covered negative example.

  - The set of positive examples is not exhaustive, therefore the set of negative examples includes also some positive examples of the arch concept.
  - Negative examples are generated in the 'object' space and not in the 'relation' space. Here the object space denotes the Cartesian product of the corresponding types of the n arguments of the target predicate (in the *cwa* mode all the possible combinations of values of n arguments of the target predicate are generated). In the relation space, each application of a utility predicate is added to the object space as an additional dimension, i.e., the relation space is the Cartesian product of k_{Attr} variables (see equation (5.4)). In this domain (as well as in the family relationships domain), relations are used to describe training examples and do not represent inherent domain knowledge (as is the case in the Eleusis example which follows). In the generation of negative examples, only the object space is affected; utility predicates are applied to the generated negative examples. Since the relations are essential to describing training examples and are not affected, not all possible negative examples are generated.

- Insufficient set of training examples. The hypotheses induced by LINUS and FOIL are correct with regard to the given training set. However, they cover the negative example given in Figure 6.3. As will be shown below, the obtained results can be improved by selecting a more appropriate set of training examples.

Learning from an enlarged training set

LINUS was used on an enlarged set of training examples, given in Figure 6.4, which includes two additional negative examples.

From the enlarged training set, containing two additional negative examples, the hypothesis induced by LINUS using both ASSISTANT and NEWGEM is essentially the same as the one reported by Winston [1975]:

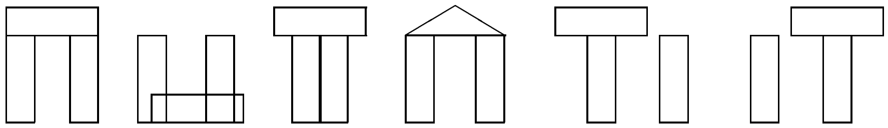


Figure 6.4: An enlarged training set of arches and near misses.

$$
\begin{array}{rcl}
arch(A,B,C) & \leftarrow & supports(B,A), \\
 & & supports(C,A), \\
 & & not\ touches(B,C).
\end{array}
$$

The induced hypothesis corresponds to the part of the semantic net description consisting of 'must' links only [Winston 1975].

6.4 Learning rules that govern card sequences

The Eleusis learning problem was originally addressed by the SPARC/E system [Dietterich and Michalski 1986]. The description here is taken from Quinlan [1990]. In the Eleusis card game, the dealer invents a secret rule specifying a condition under which a card can be added to a sequence of cards. The players attempt to add a card to the current sequence. If a card is a legal successor it is placed to the right of the last card, otherwise it is placed under the last card. The horizontal *main line* represents the sequence as developed so far, while the vertical *side lines* show incorrect plays. Three layouts, reproduced from Quinlan [1990], are given in Figure 6.5.

Each card other than the first in the sequence provides an example for learning the target relation can_follow. The example is labeled $\oplus$ if the card appears in the main line, and $\ominus$ if it is in a side line. The target relation $can\_follow(A,B,C,D,E,F)$ states that a card of rank A and suit B can follow a sequence ending with: a card of rank C and suit D; E consecutive cards of suit D; and F consecutive cards of the same color. Background relations that can be used in induced clauses are the following: $precedes\_rank(X,Y)$, $precedes\_suit(X,Y)$, $lower\_rank(X,Y)$, $face(X)$, $same\_color(X,Y)$, $odd\_rank(X)$, and $odd\_num(X)$.

main line	A♡	7♣	6♣	9♠	10♡	7♡	10♢	J♣	A♢	4♡	8♢	7♣	9♠
side lines			K♢			5♠		Q♢		3♠			9♡
			J♡										6♡
main (ctd)	10♣	K♠	2♣	10♠	J♠								
side (ctd)		Q♡											
		A♢											
main line	J♣	4♣	Q♡	3♠	Q♢	9♡	Q♣	7♡	Q♢	9♢	Q♣	3♡	K♡
side lines	K♣	5♠				4♠		10♢					
		7♠											
main (ctd)	4♣	K♢	6♣	J♢	8♢	J♡	7♣	J♢	7♡	J♡	6♡	K♢	
mainline	4♡	5♢	8♣	J♠	2♣	5♠	A♣	5♠	10♡				
side line	7♣	6♠	K♣	A♡		6♣		A♠					
	J♡	7♡	3♡	K♢									
	4♣	2♣		Q♠									
	10♠	7♠											
	8♡	6♢											
	A♢	6♡											
	2♢	4♣											

Figure 6.5: Three layouts of the Eleusis card game, from Dietterich and Michalski [1986] and Quinlan [1990].

In the first layout, the intended dealer's rule was: 'Completed color sequences must be of odd length and a male card may not appear next to a female card'. Neither FOIL nor LINUS could discover the intended rule, because no information on the gender of cards was encoded in the background relations. LINUS using ASSISTANT with post-processing induced the clauses given below.

$$\begin{array}{rcl}
can\_follow(A,B,C,D,E,F) & \leftarrow & same\_color(B,D). \\
can\_follow(A,B,C,D,E,F) & \leftarrow & odd\_num(F), \\
 & & not\ precedes\_suit(D,B). \\
can\_follow(A,B,C,D,E,F) & \leftarrow & odd\_num(F), \\
 & & not\ precedes\_rank(C,A).
\end{array}$$

While the clauses induced by LINUS using both ASSISTANT and NEWGEM cover all the positive examples, the ones induced by FOIL are not complete: they do not cover the positive example $can\_follow(10, hearts, 9, spades, 1, 3)$. This is due to the encoding length stopping criterion which prevents further search for clauses. Although the encoding length restriction is useful when dealing with noisy data, it is unsuitable for non-noisy domains. The use of the encoding length restriction could be controlled by a parameter: it should be applied if the domain is noisy or inexact, but should not be applied to exact domains with no noise, such as the domains described in this chapter. Here are the clauses induced by LINUS using NEWGEM and FOIL:

LINUS using NEWGEM
layout 1
negative examples: explicitly

$can\_follow(A, B, C, D, E, F) \leftarrow$
  $same\_color(B, D)$.
$can\_follow(A, B, C, D, E, F) \leftarrow$
  $odd\_num(F)$,
  $odd\_rank(A)$.
$can\_follow(A, B, C, D, E, F) \leftarrow$
  $notface(A)$,
  $lower\_rank(C, A)$.

FOIL
layout 1
negative examples: explicitly

$can\_follow(A, B, C, D, E, F) \leftarrow$
  $same\_color(B, D)$.
$can\_follow(A, B, C, D, E, F) \leftarrow$
  $odd\_num(F)$,
  $odd\_rank(A)$.

In the second layout, both LINUS using NEWGEM and FOIL correctly induced the intended rule: 'Play alternate face and non-face cards'.

LINUS using NEWGEM
layout 2

$can\_follow(A, B, C, D, E, F) \leftarrow$
  $face(A)$,
  $notface(C)$.
$can\_follow(A, B, C, D, E, F) \leftarrow$
  $face(C)$,
  $notface(A)$.

FOIL
layout 2

$can\_follow(A, B, C, D, E, F) \leftarrow$
  $face(A)$,
  $notface(C)$.
$can\_follow(A, B, C, D, E, F) \leftarrow$
  $face(C)$,
  $notface(A)$.

LINUS using ASSISTANT (with post-processing) generated a hypothesis containing a superfluous clause which was not removed in post-processing.

$$\begin{aligned}
can\_follow(A,B,C,D,E,F) &\leftarrow face(C),\\
&\ \ not\ face(A).\\
can\_follow(A,B,C,D,E,F) &\leftarrow face(A),\\
&\ \ not\ face(C).\\
can\_follow(A,B,C,D,E,F) &\leftarrow not\ odd\_num(E).
\end{aligned}$$

In the third layout, the intended rule is: 'Play a higher card in the suit preceding that of the last card; or, play a lower card in the suit following that of the last card'. FOIL discovered only one clause, approximately describing the first part of the rule. LINUS using ASSISTANT discovered an approximation of the whole rule: 'Play a higher or equal card in the suit preceding that of the last card; or, play a lower card in the suit following that of the last card'. The induced clauses are given below.

$$\begin{aligned}
can\_follow(A,B,C,D,E,F) &\leftarrow lower\_rank(A,C),\\
&\ \ precedes\_suit(D,B).\\
can\_follow(A,B,C,D,E,F) &\leftarrow precedes\_suit(B,D),\\
&\ \ not\ lower\_rank(A,C).
\end{aligned}$$

Again, the hypothesis induced by LINUS is complete, while FOIL's is not. Using NEWGEM, LINUS generated exactly the intended dealer's rule.

LINUS using NEWGEM layout 3	FOIL layout 3
$can\_follow(A,B,C,D,E,F) \leftarrow$ $lower\_rank(A,C),$ $precedes\_suit(D,B).$	
$can\_follow(A,B,C,D,E,F) \leftarrow$ $lower\_rank(C,A),$ $precedes\_suit(B,D).$	$can\_follow(A,B,C,D,E,F) \leftarrow$ $not\ lower\_rank(A,C),$ $precedes\_suit(B,D).$

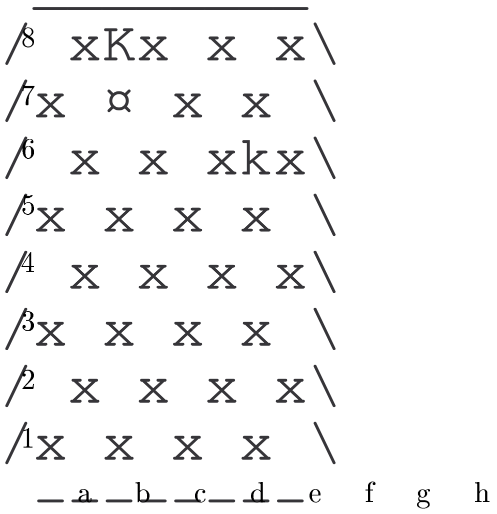


Figure 6.6: An example illegal white-to-move position in the KRK endgame.

6.5 Learning illegal chess endgame positions

In the chess endgame domain White King and Rook versus Black King, described in Muggleton et al. [1989] and Quinlan [1990], the target relation $illegal(WKf, WKr, WRf, WRr, BKf, BKr)$ states whether the position where the White King is at (WKf, WKr), the White Rook at (WRf, WRr) and the Black King at (BKf, BKr) is an illegal White-to-move position. Figure 6.6 depicts the example illegal position $illegal(g, 6, c, 7, c, 8)$.

In FOIL, the background knowledge for this learning task is represented by two relations, $adjacent(X, Y)$ and $less\_than(X, Y)$, indicating that rank/file X is adjacent to rank/file Y and rank/file X is less than rank/file Y, respectively. The arguments of these background predicates are not typed and the same predicates are used for both types of arguments. In LINUS, each of these predicates is replaced by two predicates, one for each type of arguments. Thus, LINUS uses the following relations: $adjacent\_file(X, Y)$ and $less\_file(X, Y)$ with arguments of type $file$ (with values a to h), $adjacent\_rank(X, Y)$ and $less\_rank(X, Y)$ with arguments of type $rank$ (with values 1 to 8), and equality $X = Y$, used for both types of arguments.

The experiments with LINUS and FOIL were performed on the same training and testing sets as used in Muggleton et al. [1989]. There were five small sets of 100 examples each and five large sets of 1000 examples each. Each of the sets was used as a training set for FOIL and LINUS (using ASSISTANT and NEWGEM). The induced sets of clauses were then tested as described in Muggleton et al. [1989]: the clauses obtained from a small set were tested on the 5000 examples from the large sets and the clauses obtained from each large set were tested on the remaining 4500 examples.

The classification procedures used in LINUS were the classification procedures of the corresponding attribute-value learners. For the rules obtained by ASSISTANT (by transcribing the decision trees into rules) and NEWGEM, the classification procedure was as follows: if an example is covered by the clauses of the *illegal* predicate definition (rules for the class $\oplus$), it is classified as positive, otherwise as negative.

Table 6.1 gives the results in the chess endgame experiment. The classification accuracy is given by the percentage of correctly classified testing instances and by the standard deviation (sd), averaged over 5 experiments. The first two rows are taken from Muggleton et al. [1989], the third is from Quinlan [1990] and the last three rows present the results of our experiments. Note that the results from Quinlan [1990]

	100 training examples		1000 training examples	
System	Accuracy	Time	Accuracy	Time
CIGOL	77.2%	21.5 hr	N/A	N/A
DUCE	33.7%	2 hr	37.7%	10 hr
FOIL–different sets	92.5% sd 3.6%	1.5 sec	99.4% sd 0.1%	20.8 sec
FOIL	90.8% sd 1.7%	31.6 sec	99.7% sd 0.1%	4.0 min
LINUS–ASSISTANT	98.1% sd 1.1%	55.0 sec	99.7% sd 0.1%	9.6 min
LINUS–NEWGEM	88.4% sd 4.0%	30.0 sec	99.7% sd 0.1%	4.3 min

Table 6.1: Results of learning illegal positions in the KRK chess endgame.

were not obtained on the same training and testing sets. The times in the first two rows are for a Sun 3/60, in the third for a DECStation 3100, in the fourth for a Sun 3/50 and in the last two rows CPU times are given for a VAX-8650 mainframe. The times given for LINUS include transformation to attribute-value form, learning and transformation into DHDB clauses.

In brief, on the small training sets LINUS using ASSISTANT (with post-processing) outperformed FOIL. According to the t-test for dependent samples, this result is significant at the 99.5% level. Although LINUS using NEWGEM performed slightly worse than FOIL, this result is not significant (even at the 80% level). The clauses obtained with LINUS are as short and understandable (transparent) as FOIL's. On the large training sets both systems performed equally well. Both LINUS and FOIL performed much better than DUCE and CIGOL.

Although LINUS is slower than FOIL, it is much faster than DUCE and CIGOL. LINUS is slowed down mainly by the parts implemented in Prolog, that is the DHDB interface and especially the post-processor. A more efficient implementation would significantly improve its speed. For illustration, for the small training sets, the average time spent on transforming to attribute-value form, learning and transforming to DHDB form was 16, 6 and 36 seconds for ASSISTANT, and 16, 11 and 3 seconds for NEWGEM, respectively.

For illustration, the clauses induced by LINUS using ASSISTANT (with post-processing) from one of the sets of 100 examples are given below.

$$\begin{array}{lcl}
illegal(A,B,C,D,E,F) & \leftarrow & C = E. \\
illegal(A,B,C,D,E,F) & \leftarrow & D = F. \\
illegal(A,B,C,D,E,F) & \leftarrow & adjacent\_file(A,E),\ B = F. \\
illegal(A,B,C,D,E,F) & \leftarrow & adjacent\_file(A,E), \\
 & & adjacent\_rank(B,F). \\
illegal(A,B,C,D,E,F) & \leftarrow & A = E,\ adjacent\_rank(B,F). \\
illegal(A,B,C,D,E,F) & \leftarrow & A = E,\ B = F.
\end{array}$$

These clauses may be paraphrased as: ‘A position is illegal if the Black King is on the same rank or file as (i.e., is attacked by) the Rook, or the White King and the Black King are next to each other, or the White King and the Black King are on the same square’. Although these clauses are neither consistent nor complete, they correctly classify 98.5% of the unseen cases.

7

ILP as search of refinement graphs

In this chapter, the framework of refinement graphs is used to describe FOIL and LINUS. Refinement operators and refinement graphs are first defined for MIS and then for FOIL and LINUS. Their hypothesis spaces are determined by the refinement operators they use. This enables the comparison of FOIL and LINUS in terms of the expressiveness of their hypothesis languages, the complexity of their refinement graphs, as well as the associated search costs. This chapter is based on the definition and analysis of refinement graphs in Džeroski and Lavrač [1992].

7.1 ILP as search of program clauses

Let us recall that most learning techniques described in the literature can be viewed as search techniques, either heuristic or exhaustive [Mitchell 1982]. The states in the search space are concept descriptions and the goal is to find one or more states satisfying the quality criterion. This section gives a brief summary of ILP as search, discussed already in Section 2.4 and Chapter 3.

A learning system can be described in terms of the structure of its search space, its search heuristics and search strategy. It is possible to structure the search space by using generalization and specialization operators. For a selected language bias $\mathcal{L}$ and given background knowledge $\mathcal{B}$, specialization (refinement) operators define a so-called *refinement graph* which gives the structure of the hypothesis space.

In ILP, concept descriptions are sets of program clauses. As discussed in Chapter 3, search of the space of program clauses can proceed *top-down*, from general to specific (MIS, FOIL, LINUS) or *bottom-up*, from specific to general (CIGOL, GOLEM). In top-down systems, the search for a clause starts with the most general clause and continues by iterative searching of a refinement graph defined by specialization (refinement) operators. In bottom-up systems, the search starts with the most specific clause covering some positive examples which is generalized in the clause generalization process.

Having defined the structure of the hypothesis space of an ILP system, the search strategy and heuristics remain to be specified. *Best-first search* is the desired search strategy, but time complexity usually requires the choice of some simpler strategy such as *beam search* or even *hill-climbing.* For example, the search strategy used in CIGOL is best-first search, while MIS employs basically breadth-first search with some improvements. FOIL searches the refinement graph in a greedy, hill-climbing manner. Finally, the search strategy in LINUS depends on the attribute-value algorithm used. For example, in LINUS using CN2 the search strategy is beam search.

There are two types of search heuristics: heuristics which direct the search and heuristics which decide when to stop the search, called *stopping criteria.* For example, in the case of top-down (general to specific) search, a measure of discrimination between positive and negative examples can be used as a heuristic for directing the search, and in the case of bottom-up (specific to general), a measure of information compression.

Two types of stopping criteria can be distinguished. The first determines when to stop building a clause, sometimes called the *necessity stopping criterion.* As outlined in Section 3.2.2, in exact domains this criterion usually requires consistency with respect to the set of training examples and background knowledge (stop refining a clause when it no longer covers negative examples). The second criterion, sometimes called the *sufficiency stopping criterion*, determines when to stop building new clauses for the target predicate definition. Most often, the search for new clauses is terminated when all positive examples have been covered, i.e., when the induced hypothesis is *complete* with respect to the training set and the background knowledge. In order to obtain more compact concept descriptions and to achieve noise tolerance, alternative stopping criteria may be applied, which are usually of a statistical nature.

Being empirical ILP systems, FOIL and LINUS consider all examples in one run. They are thus able to use statistical heuristics for directing and terminating the search. Such heuristics are essential for handling noisy training examples and have been successfully used in attribute-value learners. Heuristics used in FOIL and LINUS are outlined in Chapter 2. A more elaborate discussion of different heuristics can be found in Chapter 8.

7.2 Defining refinement graphs

Let us recall the definitions of θ-subsumption and generality from Section 2.4. Let c and c' be two program clauses. Clause c is at least as general as clause c' ($c \leq c'$) if c θ-subsumes c', i.e., if there exists a substitution θ, such that $c\theta \subseteq c'$ [Plotkin 1969]. 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.

Let $\mathcal{L}$ be a set of program clauses. The following definitions are adapted from Shapiro [1983] and Laird [1988].

Refinement operator A *refinement operator* ρ is a mapping from $\mathcal{L}$ to finite subsets of $\mathcal{L}$, $\rho : \mathcal{L} \rightarrow \mathcal{P}(\mathcal{L})$, such that all clauses in $\rho(c)$ are specializations of the clause c.

Refinement Let ρ be a refinement operator and c and c' two clauses in $\mathcal{L}$. Clause c' is a *refinement* of clause c if c' is a specialization of c, i.e., if $c' \in \rho(c)$.

Refinement graph A *refinement graph* is a directed, acyclic graph in which *nodes* are program clauses and *arcs* correspond to refinement operations, i.e., there is an arc from c to c', $c, c' \in \mathcal{L}$, if $c' \in \rho(c)$.

A refinement operator ρ maps a clause c to a set of its refinements $\rho(c)$ by employing two basic syntactic operations on clauses: substituting a variable with a term, and adding a literal to the body of a clause.

A refinement operator ρ induces a generality ordering on $\mathcal{L}$. For a given refinement operator ρ, the relation 'is more general' is denoted by $<_\rho$. For two clauses c and c', $c <_\rho c'$ holds if there exists a directed path from c to c' in the refinement graph induced by ρ. Informally,

$c <_\rho c'$ means that c' can be obtained from c by a series of refinement steps.

7.3 A MIS refinement operator

To illustrate the notion of a refinement operator, we briefly describe the ρ_1 refinement operator of MIS [Shapiro 1983].

Assuming a given fixed set of predicate and function symbols, the ρ_1 refinement operator is defined as in Shapiro [1983, pp. 119–120].

Clause c' is a *refinement* of clause c, i.e., $c' \in \rho_1(c)$, if:

- $c = \square$ and $c' = p(X_1, X_2, \ldots, X_n) \leftarrow$ [1] where $\square$ stands for the empty clause, p belongs to the set of given predicate symbols, and X_1,X_2, …, X_n are distinct variables.
- c is a definite program clause and c' is obtained from c by adding to the body of c a literal which has one of the following forms:
    1. $X_j = X_s$, where both X_j and X_s occur in c. Actually, in the original definition, c' is obtained from c by unifying X_j and X_s, but this is equivalent to adding the literal $X_j = X_s$ to the body of c.
    2. $X_j = f_s(Y_1, Y_2, \ldots, Y_{n_s})$, where f_s belongs to the set of given function symbols and $Y_1, Y_2, \ldots, Y_{n_s}$ are variables not occurring in c. Again, in the original definition, c' is obtained from c by instantiating X_j to $f_s(Y_1, Y_2, \ldots, Y_{n_s})$.
    3. $q_i(Y_1, Y_2, \ldots, Y_{n_i})$, where q_i belongs to the given set of predicate symbols and $Y_1, Y_2, \ldots, Y_{n_i}$ occur in c.

Given the predicate symbol *member*, the zero-place function symbol (constant) [] and the two-place function symbol $[X|Y]$ $(cons(X, Y))$, $\rho_1(member(X, Y) \leftarrow\ ) = \{member(X, X) \leftarrow\ ,\ member([], X) \leftarrow,$ $member(X, [A|B]) \leftarrow,\ member(X, Y) \leftarrow member(Y, Y)\}$. The first refinement is generated by using the first, the next two using the second and the last using the third rule from the definition of ρ_1.

---

[1] The search for a clause of the target predicate p actually starts with the clause $p(X_1, X_2, \ldots, X_n) \leftarrow true$. This clause is refined by adding literals to its body. For convenience, an empty body is written instead of *true*.

Finally, let us mention that MIS searches the refinement graph top-down, in a breadth-first fashion. The pruning mechanism it uses is based on the following property of refinements: if a clause does not cover an example, none of its refinements will (see Section 2.4).

7.4 Refinement operators for FOIL and LINUS

This section defines the refinement operators for FOIL and LINUS. As the work of LINUS actually depends on the underlying attribute-value algorithm used, the use of a particular attribute-value learner, CN2, is assumed in order to describe the refinement operator for LINUS and the way in which LINUS searches the program clause space. In that case, both FOIL and LINUS search the hypothesis space in a top-down manner and the operators they use are specialization (refinement) operators.

7.4.1 Refinement operator for FOIL

The input to FOIL consists of:

- a target predicate p of arity n, defined by a set of ground facts[2],
- a finite set of predicates q_i of arities n_i, each defined by a set of ground facts; for each predicate and its negation FOIL computes the corresponding, possibly empty, sets R_i and $\bar{R}_i$ of irreflexive partial orderings between pairs of arguments.

For example, in the chess endgame domain, described in Section 6.5, the target predicate is $illegal(A, B, C, D, E, F)$ and background knowledge $\mathcal{B}$ consists of the predicates $adjacent(X, Y)$ and $less\_than(X, Y)$. In this case $p = illegal$, $n = 6$, $q_1 = adjacent$, $n_1 = 2$, $q_2 = less\_than$, $n_2 = 2$, $R_1 = \emptyset$, $\bar{R}_1 = \emptyset$, $R_2 = \{X < Y\}$ and $\bar{R}_2 = \emptyset$.

The *refinement operator* ρ_{FOIL} maps a clause c to a set of clauses which are more specific than c. Clause c' is a *refinement* of clause c, i.e., $c' \in \rho_{FOIL}(c)$, if:

---

[2] In FOIL, ground facts are restricted to constant tuples, i.e., to contain only constant argument values and no structured terms.

- $c = \Box$ and $c' = p(X_1, X_2, \ldots, X_n) \leftarrow$ where $\Box$ stands for the empty clause, p is the target predicate symbol, and X_1,X_2, $\ldots$, X_n are distinct variables.

- c is a program clause and c' is obtained from c by adding to the body of c a literal which has one of the following forms:

    1. $X_j = X_s$ or *not* $X_j = X_s$ where both X_j and X_s occur in c.
    2. $q_i(Y_1, Y_2, \ldots, Y_{n_i})$ or *not* $q_i(Y_1, Y_2, \ldots, Y_{n_i})$, where $q_i \neq p$ and at least one of $Y_1, Y_2, \ldots, Y_{n_i}$ appears in c.
    3. $p(Y_1, Y_2, \ldots, Y_n)$ or *not* $p(Y_1, Y_2, \ldots, Y_n)$, where Y_1,Y_2, $\ldots$, Y_n occur in c and there must exist at least one X_i from the head of c, such that an irreflexive partial ordering $X_i < Y_i$ holds, i.e., $X_i < Y_i$ belongs to the set of orderings O_c associated with clause c.

The irreflexive partial orderings are used to avoid constructing recursive clauses which could invoke themselves with the same arguments they were called with (thus causing an infinite loop). If a literal $p(Y_1, Y_2, \ldots, Y_n)$ appears in the body of a clause with a head $p(X_1, X_2, \ldots, X_n)$, then a terminating recursion is guaranteed if $X_i < Y_i$ holds for some i. For a more detailed treatment of this subject see Quinlan [1990]. A recent development in the use of orderings to avoid infinite recursion is described in Cameron and Quinlan [1993].

With each clause c is associated a set O_c of orderings of its variables, established by the literals in its body. O_c is determined according to the rules given below:

- Associated with the clause $p(X_1, X_2, \ldots, X_n) \leftarrow$ is an empty set of orderings.

- When adding literals to the body of clause c, O_c changes as follows:

    1. If literal $X_j = X_s$ is added to the body of clause c with a set of orderings O_c then the new clause c' will have a set of orderings $O_{c'}$, where all occurrences of X_j and X_s in O_c are unified to obtain $O_{c'}$.

2. If literal $q_i(Y_1, Y_2, \ldots, Y_{n_i})$ is added to the body of clause c with set of orderings O_c to obtain clause c' with set of orderings $O_{c'}$, and q_i has a set of orderings R_i, then $O_{c'} = transitive\_closure(O_c \cup R_i)$; analogously, we have $O_{c'} = transitive\_closure(O_c \cup \bar{R}_i)$ for literal *not* $q_i(Y_1, Y_2, \ldots, Y_{n_i})$ with set of orderings $\bar{R}_i$.

3. $O_{c'} = O_c$, when a literal of the form $p(Y_1, Y_2, \ldots, Y_n)$ or *not* $p(Y_1, Y_2, \ldots, Y_n)$ is added to the body of clause c to obtain clause c'.

For example, if clause c is $illegal(A, B, C, D, E, F) \leftarrow less\_than(A, G)$, then $O_c = \{A < G\}$, as $O_{illegal(A,B,C,D,E,F)\leftarrow} = \emptyset$ and $R_{less\_than(A,G)} = \{A < G\}$. The literal $illegal(G, B, C, D, E, F)$ may be added to the body of c, since $A < G$ holds ($A < G \in O_c$) but $illegal(F, B, C, D, E, F)$ may not, since no ordering holds for it ($A < F \notin O_c$).

The *branching factor* of a node in a refinement graph is the number of literals that can be added to the body of the clause c in the node. It can be calculated by summing the numbers of literals added in each of the above cases. The number of literals is twice the number of different choices of arguments for the background predicates, once for positive and once for negative literals. Suppose we have a clause c with *Old* distinct variables (referred to as old variables). The number of literals added in each case is given below.

1. $b_1 = 2 \times \binom{Old}{2} = Old \times (Old - 1)$: there are $\binom{Old}{2}$ distinct choices of pairs of old variables.

2. $b_{2,i} = 2 \times [(Old + n_i - 1)^{n_i} - (n_i - 1)^{n_i}]$,[3] where n_i is the arity of predicate q_i: namely, n_i variables have to be chosen from *Old* variables and $n_i - 1$ new variables (at least one of the chosen n_i variables must be old).

---

[3] This is just an upper bound, which does not take into account that some literals containing new variables produce equivalent clauses (alphabetic variants) when added to c. For instance, the literals $between(X, A, Y)$ and $between(Y, A, X)$ produce equivalent clauses when added to the clause $illegal(A, B, C, D, E, F) \leftarrow less\_than(A, G)$

3. $b_3 = 2 \times \sum_{i=1}^{k}(-1)^{i+1}\binom{k}{i} Old^{n-i}$,[4] where n is the arity of the target predicate p and O_c contains k different orderings $X_i < Y_i$, such that X_i is in the head of c: given an ordering $X_i < Y_i$ from O_c, we can place any of the Old variables in the remaining $n-1$ argument places; having k different orderings, this gives rise to $\binom{k}{1} Old^{n-1}$ choices of arguments, but we must then remove the choices where two orderings hold ($\binom{k}{2} Old^{n-2}$), and then add the choices where three orderings hold ($\binom{k}{3} Old^{n-3}$), etc.

The branching factor of a node c in the refinement graph is then given by the following formula:

$$BF_{FOIL}(c) = b_1 + \sum_{i=1}^{l} b_{2,i} + b_3 \tag{7.1}$$

where l stands for the number of background knowledge predicates q_i.

For the clause $c = illegal(A,B,C,D,E,F) \leftarrow less\_than(A,G)$ in the chess endgame example, we have $Old = 7$, $k = 1$, $n = 6$, $n_1 = n_2 = 2$, and the branching factor for FOIL is:

$$BF_{FOIL}(c) = 7\times 6 + 2\times[2\times((7+2-1)^2-(2-1)^2)] + 2\times 1\times 7^{6-1} = 33908$$

7.4.2 Refinement operator for LINUS

The input to LINUS consists of:

- a target predicate p of arity n, defined by a set of ground facts, and the types of its arguments $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$,
- a finite set of predicates q_i of arities n_i, with their corresponding predicate definitions[5] and types of arguments $\mathcal{T}_{i,1}, \ldots, \mathcal{T}_{i,n_i}$,
- a finite set of annotated predicates (functions) f_j of arities n_j, with their corresponding predicate definitions, types of input arguments $\mathcal{T}_{j,1}, \ldots, \mathcal{T}_{j,n_j}$, and the type of the output argument $\mathcal{O}_j$,

---

[4] This formula assumes that among the k orderings in O_c, no two orderings $X_i < Y_i$ and $X_i < Y_j$, $i \neq j$, hold. Slight modifications of the formula are necessary should this be the case.

[5] Predicate definitions are sets of typed program clauses, i.e., sets of deductive database clauses. These predicates can be recursive.

- a set of type definitions of the form $\mathcal{T}_l = \{v_{l,1}, \ldots, v_{l,d_l}\}$, where $v_{l,1}$, $\ldots$, v_{l,d_l} are different constants or terms with constant argument values.

In LINUS, the background knowledge for the chess endgame domain consists of the predicates *adjacent_rank*, *less_rank*, *adjacent_file* and *less_file*, which work on arguments of types *rank* and *file*, respectively (see Section 6.5). Thus, we have: $p = illegal$, $n = 6$, $\mathcal{T}_1 = \mathcal{T}_3 = \mathcal{T}_5 = file$, $\mathcal{T}_2 = \mathcal{T}_4 = \mathcal{T}_6 = rank$, $p_1 = adjacent\_rank$, $n_1 = 2$, $\mathcal{T}_{1,1} = \mathcal{T}_{1,2} = rank$, $p_2 = adjacent\_file$, $n_2 = 2$, $\mathcal{T}_{2,1} = \mathcal{T}_{2,2} = file$, $p_3 = less\_rank$, $n_3 = 2$, $\mathcal{T}_{3,1} = \mathcal{T}_{3,2} = rank$, $p_4 = less\_file$, $n_4 = 2$, $\mathcal{T}_{4,1} = \mathcal{T}_{4,2} = file$, $rank = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $file = \{a, b, c, d, e, f, g, h\}$.

Clause c' is a refinement of clause c, i.e., $c' \in \rho_{LINUS}(c)$, if:

- $c = \Box$ and $c' = p(X_1, X_2, \ldots, X_n) \leftarrow$, where $\Box$ stands for the empty clause and X_1,X_2, $\ldots$, X_n are distinct variables of types $\mathcal{T}_1$, $\mathcal{T}_2$, $\ldots$, $\mathcal{T}_n$ as specified for the arguments of target predicate p.

- c is a DHDB clause and c' is obtained from c by adding to the body of c a literal which has one of the following forms:

  1. $X_j = v$, where X_j is a variable of type $\mathcal{T}_j$ which occurs in c and $v \in \mathcal{T}_j$.
  2. $X_j = X_s$ or *not* $X_j = X_s$, where X_j and X_s are variables of the same type $\mathcal{T}_j$ and both occur in c.
  3. $q_i(Y_1, Y_2, \ldots, Y_{n_i})$ or *not* $q_i(Y_1, Y_2, \ldots, Y_{n_i})$, where Y_1,Y_2, $\ldots$, Y_{n_i} appear in c and are of the corresponding types specified for arguments of predicate q_i.
  4. $f_j(Y_1, Y_2, \ldots, Y_{n_j}, z)$ where Y_1,Y_2, $\ldots$, Y_{n_j} appear in c and are of the corresponding types specified for input arguments of function f_j and $z \in \mathcal{O}_j$.

As stated in the previous chapters, LINUS transforms the ILP problem to a propositional form. Roughly speaking, each of the literals listed above corresponds to a propositional feature generated by LINUS during the transformation process. To guide the induction process, LINUS uses meta-level knowledge, which can exclude any of the above four cases, thus reducing the search space. If we retain only the first case, the hypothesis language is reduced to an attribute-value language.

The branching factor of node c in the refinement graph for LINUS can be calculated by summing the numbers of literals that can be added to the body of clause c for each of the above four cases:[6]

1. $b_1 = \sum_{i=1}^{n} |\mathcal{T}_i|$, where $\mathcal{T}_i$ is the type of the i-th argument of the target predicate p; for argument X_i of type $\mathcal{T}_i$, $|\mathcal{T}_i|$ literals of the form $X_i = v$ are possible.
2. $b_2 = 2 \times \sum_{s=1}^{u} \binom{k_{ArgT_s}}{2}$, where u is the number of distinct types of arguments of target predicate p and k_{ArgT_s} is the number of arguments of p of type $\mathcal{T}_s$; from the arguments of type $\mathcal{T}_s$, $\binom{k_{ArgT_s}}{2}$ different literals of the form $X_i = X_j$ can be constructed.
3. $b_{3,i} = 2 \times \prod_{s=1}^{u_i} (k_{ArgT_s})^{n_{i,s}}$, where u_i is the number of distinct types of arguments of predicate q_i, $n_{i,s}$ is the number of arguments of q_i that are of type $\mathcal{T}_s$ and k_{ArgT_s} is the number of arguments of target predicate p that are of type $\mathcal{T}_s$; we can fill $n_{i,s}$ places for arguments of type $\mathcal{T}_s$ in $(k_{ArgT_s})^{n_{i,s}}$ ways, independently from choosing the arguments of q_i which are of different types.
4. $b_{4,j} = |\mathcal{O}_j| \times \prod_{s=1}^{u_j} (k_{ArgT_s})^{n_{j,s}}$, where $|\mathcal{O}_j|$ is the number of values of the output type for function f_j, u_j is the number of distinct types of input arguments of function f_j, $n_{j,s}$ is the number of input arguments of f_j that are of type $\mathcal{T}_s$ and k_{ArgT_s} is the number of arguments of target predicate p that are of type $\mathcal{T}_s$; this formula is obtained by combining the lines of reasoning used in cases 1 and 3: z can be chosen in $|\mathcal{O}_j|$ ways, while the input arguments of f_j can be chosen in $\prod_{s=1}^{u_j} (k_{ArgT_s})^{n_{j,s}}$ ways.

For all nodes in the graph, the branching factor is then given by the following formula:

$$BF_{LINUS}(c) = BF_{LINUS} = b_1 + b_2 + \sum_{i=1}^{l_q} b_{3,i} + \sum_{j=1}^{l_f} b_{4,j} \tag{7.2}$$

---

[6] Note that the computation of the branching factor is basically the same as the computation of the number of attributes to be used for learning in LINUS, carried out in Section 5.5, which gives the complexity of the LINUS learning task.

where l_q stands for the number of background knowledge predicates q_i and l_f for the number of functions f_j. Since no new variables are introduced in the body of a clause, the branching factor is constant throughout the refinement graph, i.e., $BF(c) = BF$ for all c.

BF is, in fact, equal to the number of propositional features generated by LINUS during the transformation stage. Given a constant upper bound on the arity (number of arguments) of background knowledge predicates, this number is polynomial in n, the arity of the target predicate, and l, the number of predicates (and functions) in the background knowledge. This fact has been used to prove positive PAC-learnability results for constrained logic programs in Džeroski et al. [1992b].

For the chess endgame example, $b_1 = 6 \times 8 = 48$, $b_2 = 2 \times (\binom{3}{2} + \binom{3}{2}) = 12$, $b_{3,1} = b_{3,2} = b_{3,3} = b_{3,4} = 2 \times 3^2 = 18$, the last case contributes no new literals, and the branching factor for LINUS is:

$$BF_{LINUS} = 48 + 12 + 4 \times 18 + 0 = 132$$

7.5 Costs of searching refinement graphs

This section estimates the search costs of FOIL and LINUS. Search costs depend on the complexity inherent to the selected language $\mathcal{L}$ (the branching factor of the nodes in the refinement graph), the costs of evaluating the heuristics (the node evaluation costs), and the search strategy. More specifically, assuming a constant node evaluation cost NEC, the total search cost is given by the product of NEC and the number of nodes searched. For a selected search strategy, the number of nodes searched depends on the length LOC of the clause generated and the branching factor $BF(c)$ of the nodes in the refinement graph.

Assuming a constant branching factor BF, we obtain the following expressions for search costs of the hill-climbing and beam search strategies:

$$HillCost = LOC \times BF \times NEC \tag{7.3}$$

$$BeamCost = BeamWidth \times LOC \times BF \times NEC \tag{7.4}$$

The node evaluation cost NEC includes the cost of evaluating the search heuristics and the stopping criteria. If BF and NEC are not constant, the above expressions change accordingly.

For FOIL, equation (7.3) applies. BF and NEC are not constant. Therefore, the branching factors $BF(c)$ of clauses c along the search path (computed by equation (7.1)) and the corresponding node evaluation costs $NEC(c)$ have to be considered. Therefore, equation (7.3) is modified as follows:

$$HillCost = \sum_{c} BF(c) \times NEC(c) \tag{7.5}$$

In FOIL, the cost $NEC(c)$ of evaluating the search heuristics and the stopping criteria depends on the numbers of positive and negative tuples covered by the clause c. Thus, we have $NEC_{FOIL}(c) = \mathcal{O}(n(c))$, where $n(c) = |\mathcal{E}_c|$ and $\mathcal{E}_c$ is the local training set (the set of tuples covered by c.) Unfortunately, in FOIL the size of $\mathcal{E}_c$ can increase when a literal with new variables is added to the clause (see Section 4.1), which makes it difficult to give an exact bound for $NEC_{FOIL}(c)$. For simplicity, we assume $NEC_{FOIL}(c) = \mathcal{O}(n)$, where $n = |\mathcal{E}|$ is the number of examples in the initial training set $\mathcal{E}$. This simplification leads to the following (crude) estimate of the total cost of inducing a clause of length LOC with FOIL:

$$HillCost_{FOIL} = \mathcal{O}(LOC \times BF_{FOIL} \times n)$$

In LINUS using CN2, where the search strategy is beam search, equation (7.4) applies. The branching factor BF is constant while the node evaluation cost $NEC(c)$ varies, depending on the numbers of positive and negative examples covered by a clause. Thus, we have $NEC_{LINUS}(c) = \mathcal{O}(n(c))$, where $n(c) = |\mathcal{E}_c|$ stands for the number of training examples covered by the clause. Since in LINUS the size of the set $\mathcal{E}_c$ decreases as a clause is refined, we have $n(c) < n$ for $n = |\mathcal{E}|$, and $NEC_{LINUS}(c) = \mathcal{O}(n)$.

An additional cost in LINUS is imposed by the transformation from the logic program form into the attribute-value form and vice versa. The total cost of using LINUS with CN2 is then the sum of the transformation costs and the cost of the search for clauses. The cost of transforming training examples from the DHDB form to the attribute-value form is $TransCost_{DHDB \to AV} = \mathcal{O}(BF_{LINUS} \times n)$, while the cost of transforming an induced if-then rule to a DHDB clause is $TransCost_{AV \rightarrow DHDB} = \mathcal{O}(LOC)$, where LOC is the number of conditions (literals) in the rule (clause). An estimate of the total cost of inducing a clause with LINUS is then $BeamCost_{LINUS} = \mathcal{O}(BeamWidth \times LOC \times BF_{LINUS} \times n + BF_{LINUS} \times n + LOC)$.

To conclude, as the number of variables in clause c grows, $BF_{FOIL}(c)$ grows accordingly. This leads to a much larger search space in FOIL and $BF_{FOIL} > BeamWidth \times BF_{LINUS}$. Thus, the more expressive language in FOIL causes $HillCost_{FOIL}$ to be greater than $BeamCost_{LINUS}$.

7.6 Comparing FOIL and LINUS

The hypothesis language used in LINUS is more restricted than the one used in FOIL. Notably, no new variables are introduced and no recursive clauses are allowed in the induced clauses. Consequently, the class of problems that can be solved by LINUS is smaller than the class of problems that can be solved by FOIL.

The branching factor of a node in the refinement graph indicates the search complexity which is inherent to the concept description language $\mathcal{L}$. Since LINUS uses a more restricted hypothesis language, the branching factor of the nodes in its refinement graph is smaller; therefore, search can be more efficient. LINUS also uses meta-level knowledge to guide the induction process which can further reduce the branching factor. In the context of the expressiveness/efficiency tradeoff, it uses a less expressive language to gain efficiency.

In order to actually compare the branching factors of the two systems, the refinement operator for LINUS should be restricted to cases 2 and 3, i.e., to using equality and background predicates only. In this case, $b_{2,LINUS} < b_{1,FOIL}$ and $b_{3,i,LINUS} < b_{2,i,FOIL}$. This is due to the fact that the information about the types of variables is used to restrict the choices of the possible predicate arguments and to the fact that no new variables are introduced.[7] The latter causes BF_{LINUS} to be constant, while $BF_{FOIL}(c)$ increases with the number of variables in c. The limitation of LINUS to learning non-recursive clauses further increases the difference between the two branching factors. Thus, the branching factor for LINUS is much smaller than the one for FOIL, which allows

---

[7] FOIL2 [Quinlan 1991] already takes into account types, but our argument still holds.

LINUS to be more efficient. However, the actual performance greatly depends on the way the refinement graph is searched.

Total search cost depends also on the search heuristics and strategy. Since LINUS uses attribute-value learning algorithms for induction, the heuristics used in it depend on the mechanisms used in the attribute-value learners. In the estimation of search costs the use of the CN2 algorithm was assumed. In this case, both LINUS and FOIL use statistically based search heuristics and stopping criteria. The search strategies are hill-climbing in FOIL and beam search in LINUS using CN2. Given the big difference in the branching factors and the similar search strategies, the total cost of inducing a clause in LINUS is smaller than that in FOIL.

Part III

Handling Imperfect Data in ILP

8

Handling imperfect data in ILP

The aim of this chapter is to provide for a better conceptual understanding of different kinds of imperfect data, as well as the mechanisms and heuristics for handling imperfect data in attribute-value learning and ILP. The chapter gives an overview of the different kinds of imperfect data and the related noise-handling mechanisms. In noise-handling, heuristics play a central role. The roles of accuracy and information-based heuristics as well as the probability estimates used in the heuristics are discussed and illustrated on an example chess endgame problem.

8.1 Types of imperfect data

In real-life domains, learning systems often have to deal with imperfect data. There are various kinds of imperfections in data, including:

- noise, i.e., random errors in training examples and background knowledge,
- insufficiently covered example space, i.e., too sparse training examples from which it is difficult to reliably detect correlations,
- inexactness, i.e., an inappropriate description language which does not contain/facilitate an exact description of the target concept, and
- missing values in the training examples.

As pointed out in Brazdil and Clark [1990], noisy data may be inherent to the world being observed (e.g., due to the imprecision of some device) or to the transmission of observations to the learning system (e.g., imperfect measuring equipment, transcription errors). Furthermore, there may also be 'systematic errors' in data (e.g., caused by incorrect calibration of measuring equipment, so that it is reading consistently low).

When learning in an attribute-value language, the four kinds of data imperfections are as follows:

- noise in the training examples, which can take the form of erroneous attribute values and/or erroneous classifications (values of the class variable),
- too sparse training examples,
- an inappropriate (some attributes may not be relevant) or insufficient (important features may be missing) set of attributes, used to describe the examples, and
- missing attribute values in the training examples.

In ILP, when learning relational descriptions in a first-order language, the following forms of imperfect data are usually met:

- noise in the training examples (caused by erroneous argument values and/or erroneous classification of facts as true or false), as well as noise in the background knowledge,
- too sparse training examples,
- inappropriate (some predicates may not be relevant) or insufficient (essential predicates may be missing) background knowledge, and
- missing argument values in the training examples.

Learning systems usually have a single mechanism for dealing with the first three kinds of imperfect data, i.e., with noisy, incomplete and inexact data. Such mechanisms, often called *noise-handling mechanisms*, are typically designed to prevent the induced hypothesis from *overfitting* the data set, i.e., to avoid overly specific concept descriptions. Missing values are usually handled by a separate mechanism.

Successors of the ID3 and AQ attribute-value learners include various kinds of noise-handling techniques. Early ILP systems, on the other hand, did not address the issue of learning from imperfect data. Several recent ILP systems, however, include elaborate noise-handling mechanisms.

8.2 Handling missing values

Values of attributes/arguments of training examples can be missing, which is frequently the case in real-life databases. Not many learning systems are actually able to handle missing data. Usually, a knowledge engineer has to solve this problem before actually running a learning system. The most frequent way to solve the problem is by replacing a missing value (usually called a 'don't know' value) by the majority value of the attribute/argument where majority is considered within the class to which the training example with a missing value belongs to. It is important that the majority value within the class of the example is considered instead of the majority value in the whole training set, as the latter can lead to contradictory data. Let us take an example from medicine. Suppose that in the data set the majority of patients are male and that the value of attribute *sex* is missing in an example, describing a typical female illness; replacing the missing value with value *male* is certainly wrong.

A more sophisticated approach to handling missing values is implemented in the decision-tree learning system ASSISTANT [Cestnik et al. 1987], but even then a knowledge engineer must act with precaution. In ASSISTANT, an example with a missing value is actually replaced with several examples, one for each of the possible attribute values, weighted by the conditional (with regard to the class of the example) probabilities of the values. These probabilities can be estimated with relative frequency from the set of instances in the current node of a decision tree. CN2 [Clark and Boswell 1991] handles missing values using the same principle.

In the ILP system LINUS, three options can be used. In LINUS using ASSISTANT the above described option is handled by ASSISTANT itself. Otherwise, on request, the DHDB interface of LINUS either replaces the 'don't know' value with the majority value of the attribute for the given class, or treats it as a ‘don’t care’ value. An example with ‘don’t care’ value gets expanded into as many examples as there are values in the corresponding variable type, each with one value of the attribute. Handling of ‘don’t care’ values is frequently provided in attribute-value learners (for example, it is implemented in ASSISTANT and programs of the AQ family such as NEWGEM).

8.3 Handling noise in attribute-value learning

Several attribute-value learning programs contain mechanisms to cope with imperfect data. Some of them belong to the Top Down Induction of Decision Trees (TDIDT [Quinlan 1986]) family of programs which learn concept descriptions in the form of decision trees, such as CART [Breiman et al. 1984] and ASSISTANT [Cestnik et al. 1987]. The program C4.5 [Quinlan 1993] induces hypotheses in the form of decision trees or sets of if-then rules. The main mechanism for handling noise in TDIDT programs is *tree pruning.* The other type of programs which learn from imperfect data are rule induction programs, such as AQ15 [Michalski et al. 1986] and CN2 [Clark and Niblett 1989, Clark and Boswell 1991]. Their noise-handling mechanisms consist of, for example, *significance tests* in CN2 and *rule truncation* in AQ15.

In tree pruning and rule truncation, the unreliable parts of the hypothesis are eliminated in order to increase the predictive accuracy of the hypothesis. As a positive side-effect, the obtained descriptions are smaller, simpler and more comprehensible. When classifying a new object only a few important attributes are relied on, while the others are deliberately disregarded. The information carried by the less important attributes is not used for classifying new cases, which is sometimes a disadvantage. Nevertheless, a significant increase in classification accuracy can be achieved by tree pruning and rule truncation.

Noise-handling mechanisms in TDIDT programs include *tree pruning* [Cestnik et al. 1987] and *post-processing* of rules derived from decision trees [Quinlan 1987a]. There are two types of tree pruning: pre-pruning and post-pruning. *Pre-pruning* is performed during the construction of a decision tree and decides whether to stop or to continue tree construction at a certain node. *Post-pruning*, on the other hand, is employed after the decision tree is constructed. It estimates the classification errors in the nodes of the tree and then decides whether to prune the subtrees rooted at those nodes.

Post-processing of if-then rules derived from decision trees [Quinlan 1987a, 1987b] is a form of reduced error pruning. In reduced error pruning, operators from a predefined set are applied to the induced concept description in a greedy manner, until the operators, if applied, would result in a decrease in the classification accuracy on a given pruning set (either the set from which the description was induced or a separate testing set). In Quinlan [1987a], the post-processing of rules derived from a decision tree is performed by first eliminating irrelevant conditions from each of the rules and then by eliminating rules that do not contribute to the accuracy of the rule set. Recall the definition of the irrelevance of literals from Section 3.2.2 stating that a condition (literal) in a rule (clause) is *irrelevant* if it can be removed from the rule (clause) without decreasing its classification accuracy. If a rule covers $n^{\oplus}(c)$ examples of the predicted class and $n^{\ominus}(c)$ examples of incorrect classes, Quinlan proposes the following classification accuracy estimate:

$$A(c) = \frac{n^{\oplus}(c) - 0.5}{n^{\oplus}(c) + n^{\ominus}(c)}$$

Note that this same heuristic is used in clause post-processing in LINUS (see Section 5.4.2).

The rule induction system CN2 [Clark and Niblett 1989] uses an entropy-based search heuristic, which favors rules that cover examples of one class only. This means that, for example, a rule that covers two positive examples and no negative ones will be preferred to a rule that covers a thousand positive and two negative examples. Whereas the correlation expressed by the first rule is likely to be due to chance, the correlation detected by the second is likely to be genuine. To avoid selecting highly specific rules, CN2 uses a *significance test*, which ensures that the distribution of examples covered by the rule is significantly different from that which would occur by chance. The search strategy used is beam search and the search for rules is stopped when all examples are covered or no more significant clauses can be found.

While a significance test eliminates rules that are below a certain threshold of significance, there is still the problem of rules which just pass the significance threshold [Clark and Boswell 1991]. Such rules will tend to be preferred over more general and reliable, but apparently less accurate rules with higher entropy of the distribution of the examples covered. This actually means that the metrics of entropy has an undesirable *downward bias*, i.e., preference for rules low down in the general to specific (top-down) search space. Raising the significance threshold causes the level of specificity at which the search terminates to raise, but does not eliminate the downward bias itself.

A recent improvement of CN2 [Clark and Boswell 1991] estimates the expected accuracy of a rule using the Laplace error estimate. If a rule c which predicts class C_j in a domain with N classes covers $n(c)$ examples, $n^{C_j}(c)$ of which are positive (of class C_j), the Laplace estimate is computed by the following formula:

$$p(C_j|c) = \frac{n^{C_j}(c) + 1}{n(c) + N} \tag{8.1}$$

Using this estimate for directing the search instead of entropy eliminates the undesirable downward bias and achieves improved accuracy in practice. See Section 8.5 for a more detailed discussion on the Laplace estimate.

8.4 Handling noise in ILP

Most of the early ILP systems belong to the interactive ILP paradigm and assume correct data. Thus, they do not address the issue of learning from imperfect data. However, with the advent of empirical ILP which has the potential for practical applications, the interest in handling noise in ILP has increased. Several recent ILP systems include elaborate noise-handling mechanisms.

The mechanisms for dealing with the first three kinds of data imperfections can be based on appropriate search heuristics and stopping criteria used in hypothesis construction. An alternative approach is to generate the target predicate definition as if the data were completely correct, i.e., by employing the standard consistency and completeness stopping criteria. The induced hypothesis could then be subjected to some form of post-processing, such as reduced error pruning.

In LINUS, learning is actually performed by an attribute-value learner (see Chapter 5). As LINUS incorporates different attribute-value learners, various noise-handling mechanisms can be applied. Furthermore, the latest advances in noise handling, such as rule induction with CN2 using improved probability estimates, can easily be incorporated.

FOIL, described in Section 4.1, includes a post-processing component [Quinlan 1991], as well as stopping criteria which decide whether to stop adding literals to a clause and whether to stop adding clauses to a hypothesis. As a clause is completed, literals are eliminated from its body, if this does not cause the clause to cover new negative examples.

The stopping criteria in FOIL are based on the *encoding length restriction* [Quinlan 1990], which restricts the total length of an induced clause to the number of bits needed to explicitly enumerate the positive training examples it covers. The number of bits needed to explicitly indicate $n^{\oplus}(c)$ positive examples covered by clause c out of the n_{cur} examples in the current training set is

$$ExplicitBits(c, \mathcal{E}_{cur}) = log_2(n_{cur}) + log_2\binom{n_{cur}}{n^{\oplus}(c)}$$

The number of bits needed to encode a clause with m literals in the body is calculated as

$$ClauseBits(c) = \sum_{i=1}^{m}(1 + log_2(l) + log_2(V_{q_i})) - log_2(m!)$$

where l is the number of different predicates in the background knowledge and V_{q_i} is the number of possible variabilizations (choices of variables) of the predicate used in literal L_i [Quinlan 1990].

The construction of a clause is stopped (the necessity stopping criterion is satisfied) when

- no negative examples are covered by the clause, or
- adding any literal with positive gain would cause $ClauseBits(c)$ to exceed $ExplicitBits(c, \mathcal{E}_{cur})$.

If there are no more bits available for adding another literal, but the clause is still at least 85%[1] accurate, it is retained in the induced set of

---

[1] An ad-hoc chosen threshold.

clauses; otherwise, it is discarded.

The construction of a hypothesis (a set of clauses) is terminated (the sufficiency stopping criterion is satisfied) when

- all the positive examples are covered, or
- all the literals with positive gain require more than $ExplicitBits(c, \mathcal{E}_{cur})$ to encode.

The encoding length restriction has several deficiencies. First, in non-noisy exact domains it sometimes prevents FOIL from building complete descriptions (see Chapter 6). In noisy domains, it allows very specific clauses covering a small number of examples, which is not desirable. Suppose we have a training set with one positive and 1023 negative examples. Knowing that the domain is noisy, most systems that can handle noise would regard the single positive example as erroneous and would not even attempt to build a clause to cover it. Nevertheless, FOIL will have 20 ($20 = log_2(1024) + log_2\binom{1024}{1}$) bits available to build a clause covering the single positive example. The heuristic, thus, allows for building relatively long and specific clauses that cover a small number of examples, which is not desirable in noisy domains.

Finally, suppose several relations irrelevant to the target relation are added to the background knowledge. This would cause the number of bits needed to encode a relation to increase and thus decrease the effective number of bits available for building clauses. In practice, this would cause FOIL to fail to induce a definition of the target relation, even if it could build such a definition given only the relevant relations.

In addition, the encoding length restriction is responsible for the fact that an important feature, which exists in LINUS, is missing in FOIL, i.e., literals of the form $X = value$ may not appear in the body of a clause. The lack of this feature hinders the use of FOIL in problems which combine attribute-value and relation learning, such as the problem of early diagnosis of rheumatic diseases described in Chapter 11. Although this could be circumvented in FOIL by introducing additional predicates in the background knowledge of the form *is_a_value(X)* e.g., *is_a_2(X)*, such a modification would increase the number of bits needed to encode a clause. The encoding length restriction would then prevent

FOIL from learning a sufficient number of clauses, which would lead to incomplete hypotheses.

All the above problems with the encoding length restriction stem from the deficiences of the selected encoding scheme. A more sophisticated encoding scheme [Muggleton et al. 1992b] takes into account the information necessary to encode the background knowledge as well as the proofs which would generate the training examples covered. This encoding scheme is based on algorithmic information theory [Solomonoff 1964].

The initial implementation of GOLEM already shows awareness of the need to handle imperfect data. It allows a generated clause to cover a pre-determined number of negative examples. This is a rudimentary way of handling noise. Namely, suppose that clauses are allowed to cover at most two negative examples. If a clause covers 1000 positive and two negative examples, it probably reflects a genuine correlation in the training examples. However, if it covers two positive and two negative examples, the correlation it represents is very likely due to chance. GOLEM should take into account at least both the number of positive and negative examples covered by the clause and allow, for instance, clauses to be built which are at least 85% accurate.

Several other approaches to noise-handling can be applied with GOLEM. An alternative approach is not to allow clauses to cover negative examples, but to apply some form of reduced error pruning after the clauses are generated. Recently, GOLEM has been used within a non-monotonic learning framework [Bain 1991], employing in addition the above-mentioned encoding scheme of Muggleton et al. [1992b]. Good results in noise-handling in practical domains have been reported [Srinivasan et al. 1992] using this scheme.

Reduced error pruning is the mechanism applied to sets of clauses induced by FOCL, a learning system that integrates empirical ILP (FOIL) and explanation-based learning [Pazzani and Kibler 1992]. The set of training examples is split into a set for learning and a set for pruning. Two operators, *delete last literal* and *drop clause* are first independently applied to each clause of the induced hypothesis. The modification that yields the greatest improvement in accuracy is retained. The procedure is repeated until the application of any of the operators would decrease accuracy. The authors report that reduced error pruning performs better than FOIL's stopping criteria used in FOCL, except for small training sets (80 examples). They use training sets of 80, 160, 320 and 480 examples of illegal positions in the KRK chess endgame, split into 66.7% for training and 33.3% for pruning. They also synthetically introduce noise in arguments (called *tuple* noise) amounting to 5% and classification noise (20%) in the training examples.

8.5 Heuristics for noise-handling in empirical ILP

This section proposes several heuristics to be used in empirical ILP for directing the search for clauses and possibly for stopping criteria or post-processing [Lavrač et al. 1992a, 1992b]. It assumes a top-down empirical ILP algorithm, consisting of the covering and specialization loops as outlined in Algorithm 3.2 of Section 3.2.2. In the specialization loop, only one specialization operator is assumed: adding a literal L to the body of the current clause c. By adding a literal to the body of clause $T \leftarrow Q$ (where Q is a conjunction of literals), a new clause c' is constructed $T \leftarrow Q'$, where $Q' = Q, L$. Search typically starts with the clause with an empty body and continues by heuristically searching for literals to be added to the body of the clause.

In this section we consider the heuristics used to guide the search of literals. The same heuristics can also be used as stopping criteria, deciding whether to stop adding literals to the clause and whether to stop adding clauses to the hypothesis. Furthermore, we consider only the specialization of a single clause c of hypothesis $\mathcal{H}$, which in general consists of a set of clauses. Note that this is done in order to simplify the analysis; clauses already generated should also be taken into account. Recall the discussion of this problem in Section 3.2.2.

In addition to the entropy/information-based search heuristic used in the construction of decision trees (see Section 5.2.2), a variety of other heuristics have been proposed by other authors [Breiman et al. 1984, Mingers 1989b]. An empirical comparison of search heuristics [Mingers 1989b] shows that there is little difference between them and that their use reduces the size rather than improves the accuracy of induced hypotheses. However, later experiments have shown that this holds in some domains, whereas in others substantial differences in accuracy may also arise [Buntine and Niblett 1992]. Hence, the importance of search heuristics should not be underestimated. Furthermore, the accuracy depends also on the heuristics applied in post-processing of rules/decision trees/clauses [Brunk and Pazzani 1991, Mingers 1989a, Quinlan 1987a, 1987b].

Heuristics use different probability estimates. It was shown that the method of estimating probabilities used in the heuristics has a greater impact on the accuracy of the induced hypothesis than the actual form of the heuristics (accuracy, entropy, gini-index) [Cestnik 1991]. Therefore, in order to improve accuracy in ILP, we propose to use simple heuristics as defined in this section and a reliable probability estimation method as proposed in Section 8.6.

Recall the notational conventions from Section 3.2.2. Let n be the number of examples in the initial training set, $n^{\oplus}$ of which are positive and $n^{\ominus}$ negative. Let n_{cur} denote the number of examples in the current training set $\mathcal{E}_{cur}$, and $n(c)$ the number of examples in the local training set $\mathcal{E}_c$, i.e., the set of examples from the current training set $\mathcal{E}_{cur}$ which are covered by the clause $c = T \leftarrow Q$.

The *quality* of a clause c (with respect to the corresponding training set $\mathcal{E}_{cur}$) is computed by estimating the *goodness of split*, i.e., how 'good' is the distribution (split) $n^{\oplus}(c) - n^{\ominus}(c)$ of positive and negative examples covered by the clause. In principle, the more positive examples covered the better. Note, however, that things are not so simple in noisy domains.

In the following, we first propose two types of heuristics which can be used in empirical ILP to evaluate the quality of a clause. The simplest measure of the quality of a clause $c = T \leftarrow Q$ is its *expected classification accuracy*, defined as the probability that an example covered by clause c is positive[2].

Accuracy

$$A(c) = p(\oplus|c) \tag{8.2}$$

The *expected classification error* is defined as $1 - A(c)$.

---

[2] In ILP terms, $p(\oplus|c)$ means $p(T|Q)$, i.e., the probability that the head of clause $T \leftarrow Q$ is implied by its body. For correct clauses, $p(T|Q)$ is 1, but this is not necessarily true for clauses induced from noisy examples.

Another measure of the quality of a clause is its expected *informativity*, which is the amount of information necessary to specify that an example covered by the clause is positive.

Informativity

$$I(c) = -log_2\ p(\oplus|c) \tag{8.3}$$

To better suit the ILP framework, the proposed heuristic differs from the entropy-based informativity [Quinlan 1986], which could be used instead. The goal in ILP is, namely, to build clauses covering as many positive examples as possible (preferably excluding negative examples) and not to build descriptions that would maximally discriminate between classes $\oplus$ and $\ominus$. The usual entropy measure may be used if we want to generate a description of the class $\ominus$ as well.

Given the current clause $c = T \leftarrow Q$ and a new clause $c' = T \leftarrow Q'$, *accuracy gain* $AG(c', c)$ and *information gain* $IG(c', c)$ are defined as follows.

Accuracy gain

$$AG(c', c) = A(c') - A(c) = p(\oplus|c') - p(\oplus|c) \tag{8.4}$$

Information gain

$$IG(c', c) = I(c) - I(c') = log_2\ p(\oplus|c') - log_2\ p(\oplus|c) \tag{8.5}$$

The accuracy gain is the increase in the classification accuracy, while the information gain is the decrease in the information necessary to specify that an example covered by the clause is positive, achieved by specializing clause c to c'.

We next propose the use of weights in the above heuristics. Without taking into account the number of examples covered by a clause, the heuristics can namely favor very specific clauses with high gain [Smyth et al. 1990]. The weights are introduced in equations (8.6) and (8.7). If accuracy and information gain are weighted by the relative frequency of positive examples covered at an individual specialization step $\frac{n^{\oplus}(c')}{n^{\oplus}(c)}$, *weighted accuracy gain* $WAG(c', c)$ and *weighted information gain* $WIG(c', c)$ are obtained.

Weighted accuracy gain

$$WAG(c', c) = \frac{n^{\oplus}(c')}{n^{\oplus}(c)} \times (p(\oplus|c') - p(\oplus|c)) \tag{8.6}$$

Weighted information gain

$$WIG(c', c) = \frac{n^{\oplus}(c')}{n^{\oplus}(c)} \times (log_2\ p(\oplus|c') - log_2\ p(\oplus|c)) \tag{8.7}$$

The main purpose of introducing weighted gains is to find the balance between the gain and the number of examples covered by the clause. A similarly defined weighted information gain is used in FOIL (see Section 4.1).

8.6 Probability estimates

Heuristics for evaluating the quality of clause c use probabilities that have to be estimated from the current training set $\mathcal{E}_{cur}$. The probability estimates are based on the distribution (split) $n^{\oplus}(c) - n^{\ominus}(c)$ of positive and negative examples covered by the clause. Relative frequency $\frac{n^{\oplus}(c)}{n(c)}$ is often used to approximate the probability $p(\oplus|c)$. However, the reliability of this approximation decreases as the size of $\mathcal{E}_{cur}$ decreases. In the extreme case of only one positive example in $\mathcal{E}_{cur}$ the estimate of $p(\oplus|c)$ is 1. Even when the data are not noisy, this estimate is too optimistic. To avoid this problem, the Laplace law of succession can be used [Niblett and Bratko 1986]: if in the sample of n trials there were s successes, the probability of the next trial being successful is $\frac{s+1}{n+2}$, assuming a uniform initial distribution of successes and failures. To avoid this assumption, Cestnik [1990] proposes the use of the m-estimate derived by a Bayesian estimation procedure; after s successes in n trials, the probability of success in the next trial is estimated as $\frac{s+p_a m}{n+m}$, where p_a is the prior probability of success and m is a parameter of the method.

Relative frequency

$$p(\oplus|c) = \frac{n^{\oplus}(c)}{n(c)} \tag{8.8}$$

The use of relative frequency is appropriate when the number of covered examples is large. In that case, all the estimates considered give practically the same result. However, when a small number of examples is available, as is often the case in practice, the use of relative frequency is inappropriate. For instance, problems arise when $n(c) = 0$ (division by 0). If $n(c) > 0$ and $n^{\oplus}(c) = 0$ then $p(\oplus|c)$ becomes 0, even if $n(c)$ is small. As shown by Cestnik, this means that the estimate is not reliable [Cestnik 1990].

Laplace estimate

$$p(\oplus|c) = \frac{n^{\oplus}(c) + 1}{n(c) + 2} \tag{8.9}$$

Equation (8.9) applies, as there are only two classes in ILP: $\oplus$ and $\ominus$. In a domain with N classes C_j, the formula in equation (8.1) applies. The Laplace estimate is more reliable than relative frequency when dealing with a small number of examples; however, it relies on the assumption of a uniform prior probability distribution of the two (N) classes [Niblett and Bratko 1986] which is rarely met in practice.

m-estimate

$$p(\oplus|c) = \frac{n^{\oplus}(c) + m \times p_a(\oplus)}{n(c) + m} \tag{8.10}$$

The m-estimate [Cestnik 1990] avoids the problems of relative frequency and the Laplace estimate (in particular when the number of training examples is small) by taking into account the prior probabilities of the classes. The prior probability $p_a(\oplus)$ can be estimated by the relative frequency of positive examples in the initial training set: $\frac{n^{\oplus}}{n}$. The parameter m can be set subjectively; it expresses our confidence in the experimental evidence (training examples). The actual value of m should be set according to the amount of noise in the examples (larger m for more noise). As m grows toward infinity, the m-estimate approaches the prior probability of the corresponding class.

The goal of proposing the m-estimate to be used in ILP is to provide for a firm theoretical background based on the Bayesian analysis and to build hypotheses which are more accurate when classifying unknown cases [Cestnik 1990, 1991]. The m-estimate is a general probability estimate. By appropriately setting its two parameters m

and $p_a(\oplus)$, the other two estimates are obtained as special cases of the m-estimate:

- relative frequency $p(\oplus|c) = \frac{n^{\oplus}(c)}{n(c)}$ (for $m = 0$), and
- the Laplace estimate $p(\oplus|c) = \frac{n^{\oplus}(c)+1}{n(c)+2}$ (for $m = 2$, $p_a(\oplus) = \frac{1}{2}$).

8.7 Heuristics at work

In this section, we illustrate the use of the above defined heuristics and probability estimates on a simple example, based on the task of learning illegal chess endgame positions, as defined in Section 6.5. In addition to the background knowledge predicates defined for LINUS, we will assume the presence of two other predicates, $aeq\_file(X, Y)$ and $aeq\_rank(X, Y)$, where aeq stands for $adjacent\_or\_equal$. For notational convenience, the abbreviations adj_file and adj_rank are used instead of $adjacent\_file$ and $adjacent\_rank$, respectively.

8.7.1 The training set and its partitions

Suppose that, from an initial training set of 100 examples [Quinlan 1990], an ILP algorithm has constructed the following current clause c:

$$illegal(A, B, C, D, E, F) \leftarrow adj\_file(A, E) \qquad (8.11)$$

The local training set $\mathcal{E}_c$ of the 15 examples covered by the current clause c is given in Table 8.1. Out of 15 examples, six are positive and nine are negative. This situation is denoted by a 6-9 split of the local training set $\mathcal{E}_c$. Note that the first negative example is 'noisy': it is purposely incorrectly classified as a legal endgame position ($\ominus$) although it is, in fact, illegal ($\oplus$).

Adding literal L to the body of clause (8.11) leads to a new clause $c' = illegal(A, B, C, D, E, F) \leftarrow adj\_file(A, E), L$ and a split of the new local training set $\mathcal{E}_{c'}$. Theoretically, in $\mathcal{E}_{c'}$ there are 68 further splits of the 6-9 split of $\mathcal{E}_c$: 6-0, 5-0, ..., 1-0, 6-1, ... (7 ways of choosing among 6 positive examples, 10 ways of choosing among 9 negative examples, disregarding splits 0-0 and 6-9; $7 \times 10 - 2 = 68$). The splits are ordered according to the decreasing values of the classification accuracy

Positive examples		*Negative examples*	
$illegal(e,2,e,7,d,2)$	$\oplus$	$illegal(c,6,f,6,d,7)$ $(noisy)$	$\ominus$
$illegal(g,3,g,1,f,4)$	$\oplus$	$illegal(e,8,a,6,d,5)$	$\ominus$
$illegal(c,4,e,3,d,5)$	$\oplus$	$illegal(e,6,g,6,f,3)$	$\ominus$
$illegal(f,1,b,7,e,2)$	$\oplus$	$illegal(d,3,b,4,e,8)$	$\ominus$
$illegal(f,4,b,1,e,4)$	$\oplus$	$illegal(c,8,g,3,d,5)$	$\ominus$
$illegal(f,5,c,8,g,5)$	$\oplus$	$illegal(a,2,g,1,b,8)$	$\ominus$
		$illegal(h,2,e,5,g,8)$	$\ominus$
		$illegal(c,4,e,6,b,7)$	$\ominus$
		$illegal(h,2,b,1,g,7)$	$\ominus$

Table 8.1: Examples covered by the current clause.

heuristic from equation (8.2) if computed using the relative frequency estimate from equation (8.8): $6-0_1$, $5-0_2$, $4-0_3$, $3-0_4$, $2-0_5$, $1-0_6$, $6-1_7$, $5-1_8$, $4-1_9$, $6-2_{10}$, $3-1_{11}$, ... Indexes denote the split number (as ranked by the accuracy heuristic). The splits on the x axes of Figures 8.1 and 8.2 correspond to this ordering.

8.7.2 Search heuristics at work

Which literal L is to be added next to the body of clause (8.11)? For the given argument types and due to the symmetry of predicate arguments there are 48 possible applications of the predicates from the background knowledge (including the applications of the equality predicate, and disregarding the applications of the target predicate itself). For each of the 48 applicable literals L, a new local training set $\mathcal{E}_{c'}$ of clause $c' = illegal(A,B,C,D,E,F) \leftarrow adj\_file(A,E), L$ is obtained. Out of 68 theoretically possible splits, only 24 splits can actually occur by adding one of the 48 literals (since different literals may cause the same splits). Eight out of the 48 possible literals are listed below, together with the resulting splits: $B = F$ $(3-0_4)$, $A = C$ $(2-0_5)$, $aeq\_file(A,C)$ $(2-0_5)$, $aeq\_rank(B,F)$ $(6-1_7)$, $adj\_rank(B,F)$ $(3-1_{11})$, $adj\_file(C,E)$ $(3-1_{11})$, *not* $less\_rank(B,D)$ $(3-6_{41})$, and *not* $aeq\_rank(B,F)$ $(0-8_{60})$.

For four of the above literals, Table 8.2 gives their corresponding splits and the estimates of the quality of clause c' after adding L to the

Literal	*Split*	AG_{rf}	AG_{Lap}	$AG_{m=2}$	WAG_{rf}
$B = F$	$3-0_4$	$1:0.600$	$4:0.388$	$5:0.340$	$11:0.120$
$A = C$	$2-0_5$	$1:0.600$	$6:0.338$	$9:0.273$	$17:0.080$
$aeq\_rank(B,F)$	$6-1_7$	$7:0.457$	$5:0.366$	$4:0.348$	$2:0.213$
$adj\_rank(B,F)$	$3-1_{11}$	$10:0.350$	$10:0.255$	$11:0.218$	$15:0.093$
Literal	*Split*	IG_{rf}	IG_{Lap}	$IG_{m=2}$	WIG_{rf}
$B = F$	$3-0_4$	$1:1.322$	$4:0.958$	$5:0.902$	$15:0.264$
$A = C$	$2-0_5$	$1:1.322$	$6:0.865$	$9:0.763$	$22:0.176$
$aeq\_rank(B,F)$	$6-1_7$	$7:1.100$	$5:0.918$	$4:0.918$	$2:0.513$
$adj\_rank(B,F)$	$3-1_{11}$	$10:0.907$	$10:0.695$	$11:0.639$	$17:0.242$

Table 8.2: Clause quality measured by AG, WAG, IG and WIG using different probability estimates.

body of clause (8.11). The clause quality is computed using the accuracy gain AG and WAG, as well as the information gain heuristics IG and WIG with different probability estimates. The index rf denotes relative frequency, Lap the Laplace estimate, and $m = 2$ denotes the m-estimate with m set to 2 and probability $p_a(\oplus) = \frac{1}{3}$ (the relative frequency of positive examples in the initial training set of 100 examples). In each individual column, the numbers preceding ':' denote the ranking of splits within each individual heuristic. For example, $4 : 0.348$ (in row 3, column 5) means that the literal $aeq\_rank(B,F)$ with split $6-1$ has the fourth best rank when evaluated by $AG_{m=2}$. This same literal (this same split), evaluated by AG_{rf}, has only the seventh best rank $7 : 0.457$ (row 3, column 3).

Figures 8.1 and 8.2 show the values of the accuracy gain and information gain heuristics using different probability estimates on the 68 splits. Table 8.2 and Figures 8.1 and 8.2 show that the heuristics AG and IG using relative frequency evaluate the 'pure' splits 6-0 to 1-0 as best. This is questionable as small splits, such as 1-0, are unreliable. Knowing the chess endgame domain, the best literal to be added to clause (8.11) is $aeq\_rank(B,F)$. Its effect is a 6-1 split, covering the 'noisy' example which is misclassified as negative. By AG_{rf} this split is evaluated only as the seventh best $(6-1_7)$. On the other hand, this same split is evaluated as the best by $AG_{m=2}$ and WAG_{rf}

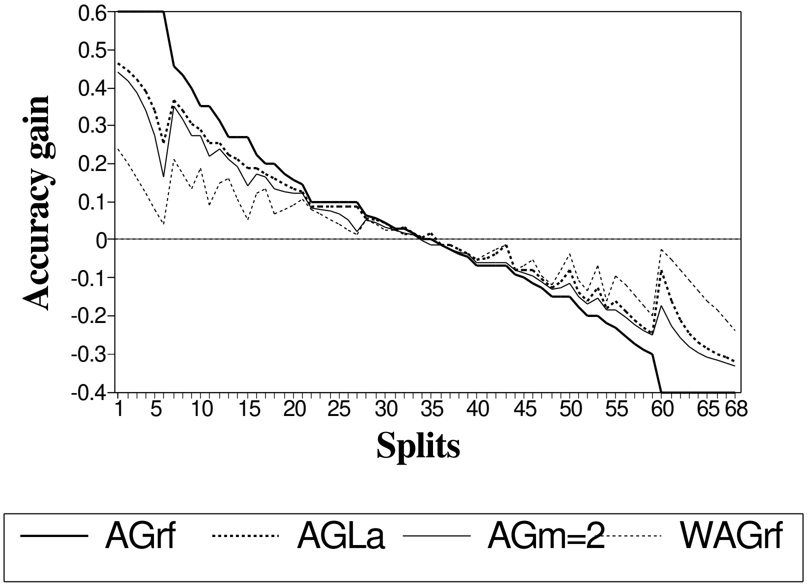


Figure 8.1: Values of heuristics AG_{rf}, AG_{Lap}, $AG_{m=2}$ and WAG_{rf} on 68 splits, ordered according to the decreasing AG_{rf}.

(see Figure 8.1). Apart from the $AG_{m=2}$ and WAG_{rf} heuristics, also the $IG_{m=2}$, and WIG_{rf} (as well as WAG_{Lap}, $WAG_{m=2}$, WIG_{Lap} and $WIG_{m=2}$, which are not in Table 8.2) heuristics would in fact select this literal.

Let us explain this in some more detail. The heuristic $AG_{m=2}$ selects the following best splits: 1: 6-0, 2: 5-0, 3: 4-0, 4: 6-1 (the value 0.348 for the literal $aeq\_rank(B, F)$ in row 3, column 5), 5: 3-0, etc. Since no literals that would cause 6-0, 5-0 or 4-0 splits exist, the fourth split 6-1 is selected as the best, leading to clause $illegal(A, B, C, D, E, F) \leftarrow adj\_file(A, E), aeq\_rank(B, F)$. Having set $m = 2$, the covered negative example is correctly judged to be noisy.

Figures 8.1 and 8.2 show that the heuristics AG_{rf} and IG_{rf} are substantially different from the other three heuristics which are, on the other hand, qualitatively similar. The heuristics AG_{Lap} and $AG_{m=2}$ are the 'closest' (the most similar) according to the similarity measure

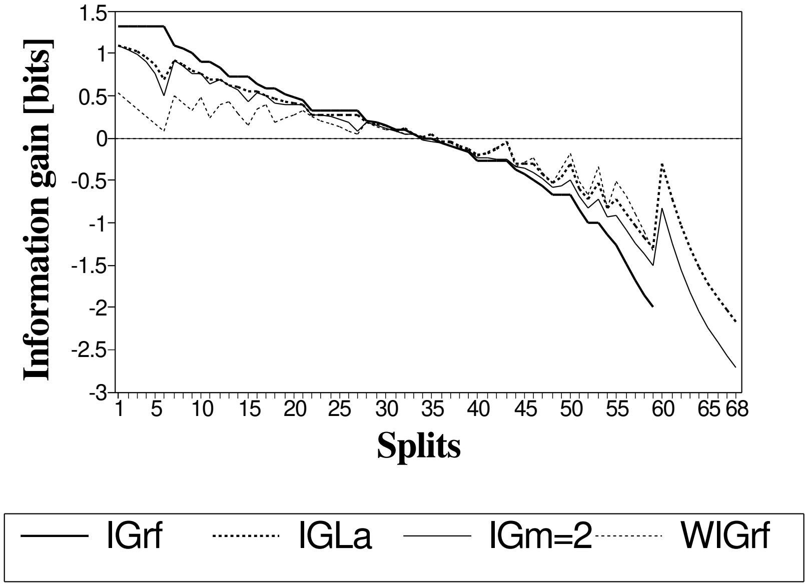


Figure 8.2: Values of heuristics IG_{rf}, IG_{Lap}, $IG_{m=2}$ and WIG_{rf} on 68 splits, ordered according to the decreasing AG_{rf}.

which counts the differences between the split ranks produced by the two heuristics. In our domain this is understandable since the Laplace assumption of a uniform prior distribution of classes $\oplus$ and $\ominus$ is not too strongly violated, having a 6-9 distribution of the $\oplus$ and $\ominus$ examples.

Using the same similarity measure, it was shown that all WAG (and WIG) heuristics are 'closer' to $AG_{m=2}$ (and $IG_{m=2}$) than other heuristics are. Since empirical evidence strongly suggests to use the m-estimate in order to achieve better classification accuracy [Cestnik 1990, Cestnik and Bratko 1991, Džeroski 1991], this leads also to the hypothesis that heuristics using weights perform better than unweighted ones, the best being WAG and WIG using the m-estimate. As a rule of thumb, if AG is to be used then AG_m is recommended. Else, WAG should be used.

To conclude, the analysis of the proposed search heuristics on the simple chess endgame example suggests that weighted accuracy and information gain WAG and WIG exhibit a similar performance to unweighted AG and IG using the m-estimate. This leads also to the hypothesis that weighted heuristics perform better than unweighted ones, the best being WIG and WAG using the m-estimate. This claim needs to be reinforced with further experimental work in real-life domains.

9

mFOIL: Extending noise-handling in FOIL

The ILP system mFOIL [Džeroski 1991, Džeroski and Bratko 1992a] is largely based on the FOIL approach. The main difference is that mFOIL uses search heuristics and stopping criteria that improve noise-handling. Another major difference is the beam search strategy used in mFOIL as opposed to the hill-climbing search used in FOIL. To reduce its search space, mFOIL uses some additional information, such as the symmetry and different variables (rectified literals) constraints, described in the next section.

In this chapter, we first describe the search space of mFOIL. We then describe the search heuristics, the search strategy and stopping criteria used to handle noisy data. Finally, we discuss some important implementation details.

9.1 Search space

The search space of mFOIL is determined by the language of program clauses, the predicates from the background knowledge, and their corresponding declarations. As illustrated in Figure 9.1, the background knowledge may take the form of a logic program, i.e., a set of program clauses. In this logic program, all the background predicates have to be defined in such a way that each call to a predicate always instantiates (binds) all of its unbound arguments. The background knowledge may contain non-ground and non-unit clauses, as in LINUS. FOIL and GOLEM, on the other hand, can only use ground facts as background knowledge.

The training examples have the form of ground facts. Structured terms may appear in training examples and background knowledge, but are treated as constants by mFOIL. This in fact means that $f(a, b)$ and $f(c, d)$ may only be generalized by mFOIL to a variable A, and not to a structured term $f(X, Y)$. To enable generalizations of the latter form, the transformations of flattening and unflattening [Rouveirol 1990] may be used, which transform function symbols to predicate symbols and vice versa.

A sample input file for mFOIL, containing training examples, background knowledge and appropriate declarations for the task of learning illegal positions in the KRK chess endgame (described in Section 6.5) is given in Figure 9.1. For notational convenience, $adjacent\_file$ and $adjacent\_rank$ are abbreviated to adj_file and adj_rank, respectively. The declaration $learn(illegal(file, rank, file, rank, file, rank))$ identifies the target relation and the types of its arguments. Similarly, declarations, such as $base(adj\_file(file, file))$ identify the background knowledge predicates and the corresponding types of arguments. In order to learn recursive clauses, the target predicate has to be declared as a background predicate as well.

The variables chosen as arguments of a literal have to be of the types appropriate for the corresponding predicate. In order to further reduce the search space of possible literals, additional information from the declarations is used, which is described below and includes input/output modes (designation of arguments as input/output), possible symmetries of predicate arguments and rectification of literals.

- *Input/output modes.* The declaration $modes(adj\_file(+, +))$ states that both arguments of the predicate adj_file have to be bound, i.e., the variables in a literal with this predicate symbol have to appear in the clause to which the literal is added. The declaration $modes(neighbor(+, -))$ states that the first argument of the predicate *neighbor* has to be bound, while the second may be either bound or unbound. This means that in a literal $neighbor(X, Y)$, X will have to be an old variable, while Y may be either an old or a new variable. Finally, the declaration $modes(neighbor(-, -))$ means that any of the arguments may be input or output.

  In general, a '+' argument is processed in such a way that a new variable cannot be introduced in its place. Either a new or an old

```
% type declarations
type(file, [a, b, c, d, e, f, g, h]).
type(rank, [1, 2, 3, 4, 5, 6, 7, 8]).

% predicate declarations
learn(illegal(file, rank, file, rank, file, rank)).

base(X = Y).
base(adj_file(file, file)).             base(less_file(file, file)).
base(adj_rank(rank, rank)).             base(less_rank(rank, rank)).
modes(adj_file(+, +)).                  modes(less_file(+, +)).
modes(adj_rank(+, +)).                  modes(less_rank(+, +)).
symmetric(adj_file, [(1, 2)]).          symmetric(less_file, [ ]).
symmetric(adj_rank, [(1, 2)]).          symmetric(less_rank, [ ]).

% background knowledge
adj_file(a, b). adj_file(b, a).         adj_rank(1, 2). adj_rank(2, 1).
adj_file(b, c). adj_file(c, b).         adj_rank(2, 3). adj_rank(3, 2).
adj_file(c, d). adj_file(d, c).         adj_rank(3, 4). adj_rank(4, 3).
adj_file(d, e). adj_file(e, d).         adj_rank(4, 5). adj_rank(5, 4).
adj_file(e, f). adj_file(f, e).         adj_rank(5, 6). adj_rank(6, 5).
adj_file(f, g). adj_file(g, f).         adj_rank(6, 7). adj_rank(7, 6).
adj_file(g, h). adj_file(h, g).         adj_rank(7, 8). adj_rank(8, 7).

                                        % background knowledge
                                        less_rank(X, Y) : -X < Y.

% training examples                     % background knowledge
positive(illegal(g, 6, c, 7, c, 8)).    less_file(a, b).
positive(illegal(d, 8, c, 2, d, 7)).    less_file(b, c).
positive(illegal(b, 6, c, 1, h, 1)).    less_file(c, d).
                                        less_file(d, e).
negative(illegal(b, 6, g, 4, b, 3)).    less_file(e, f).
negative(illegal(c, 6, a, 2, f, 3)).    less_file(f, g).
negative(illegal(h, 8, e, 4, h, 2)).    less_file(g, h).
```

Figure 9.1: mFOIL input for the problem of learning illegal positions in the KRK chess endgame.

variable may be substituted for a '–' argument. If no mode declaration is given for a predicate, mFOIL assumes that all of the arguments may be either input or ouput. However, at least one of the variables in a variabilization [Pazzani et al. 1991] of a predicate (new literal) must be old, regardless of mode declarations. This constraint is enforced in FOIL as well, and is equivalent to the connectedness constraint of Rouveirol [1992].

- *Symmetric predicates.* A binary predicate p is symmetric if the truth values of $p(X,Y)$ and $p(Y,X)$ are equal for any binding of X and Y. In general, a predicate can be symmetric in a pair of arguments. The declaration $symmetric(adj\_file,[(1,2)])$ states that the predicate adj_file is symmetric in its first and second argument. This means that only one of the literals $adj\_file(X,Y)$ and $adj\_file(Y,X)$ will be considered when adding literals to the body of a clause. Information about predicate symmetry is also used in LINUS (see Section 5.4.4).

- *Rectified literals.* A constraint *variables(different)* is applied in mFOIL, thus further reducing the number of variabilizations of a predicate. It usually turns out that literals like $adj\_file(X,X)$ or $less\_rank(Y,Y)$ have truth value $false$ for all possible values of X and Y. Thus, they impose an unnecessary computational overhead in the learning process. Literals that do not contain identical variables as arguments are named *rectified* literals. As a relation where two arguments are identical can be represented by a relation of a smaller arity, only rectified literals are allowed in mFOIL by default (parameter *variables* has value *different*). This constraint may be eliminated by setting the parameter *variables* to the value *identical* (see Section 9.4).

9.2 Search heuristics

The main difference between mFOIL and FOIL is that, instead of the entropy-based weighted information gain heuristic in FOIL, an accuracy estimate is used as a search heuristic in mFOIL. This may be the Laplace estimate or the more sophisticated m-estimate [Cestnik 1990, 1991]. Both estimates have been used in tree pruning

[Cestnik and Bratko 1991] and as search heuristics in rule induction [Clark and Boswell 1991, Džeroski et al. 1992a], greatly improving noise-handling in attribute-value learning systems. Both estimates are described in Section 8.6.

If a clause c covers $n(c)$ training examples, out of which $n^{\oplus}(c)$ are positive, its expected accuracy can be estimated by the Laplace estimate

$$A(c) = p(\oplus|c) = \frac{n^{\oplus}(c) + 1}{n(c) + 2}$$

or by the m-estimate

$$A(c) = p(\oplus|c) = \frac{n^{\oplus}(c) + m \times p_a(\oplus)}{n(c) + m}$$

where $p_a(\oplus)$ is the prior probability of the class $\oplus$ and is estimated by the relative frequency of positive examples in the whole training set.

It should be noted that these estimates are computed directly from the examples in the original training set in mFOIL, in contrast to the use of a local training set of extended tuples for this purpose in FOIL. This decision is justified by the following argument. Suppose a single noisy training example, erroneously classified as positive, is covered by a clause. In FOIL, this example may be extended to, for instance, ten tuples in the local training set. Estimating the expected accuracy from ten positive tuples would yield a score high enough for the clause to be accepted. Thus a clause might be built that covers a single erroneous example.

The Laplace estimate and the m-estimate used as search heuristics allow for a kind of pruning similar to the gain pruning in FOIL (see Sections 4.1 and 8.4). This kind of pruning is described here, although it is not actually implemented in mFOIL. Suppose a literal L with new variables is to be added to a clause. If the clause c, obtained with this specialization, covers $n^{\oplus}(c)$ positive examples, the best we can achieve with further refinements is a clause that covers $n^{\oplus}(c)$ positive and no negative examples. The heuristic value $A(c)$ of such a clause would be $\frac{n^{\oplus}(c)+1}{n^{\oplus}(c)+2}$ or $\frac{n^{\oplus}(c)+m\times p_a(\oplus)}{n^{\oplus}(c)+m}$, if estimated by the Laplace and the m-estimate, respectively. If this value is less than the heuristic value of the best partial clause found so far, no replacement of new variables in L

with old will produce a better clause. Consequently, such replacements need not be considered, thus saving considerable computational effort.

In fact, any well behaved heuristic function facilitates this kind of pruning. Suppose that the heuristic value of a clause covering $n^{\oplus}(c)$ positive and $n^{\ominus}(c)$ negative examples is estimated by the value of the function $\mathcal{Q}(n^{\oplus}(c), n^{\ominus}(c))$. The pruning process described above is then justified if $\frac{\partial \mathcal{Q}(s,t)}{\partial s} > 0$ and $\frac{\partial \mathcal{Q}(s,t)}{\partial t} < 0$, for all non-negative s and t.

9.3 Search strategy and stopping criteria

A covering approach, similar to the one in FOIL, is used in mFOIL. This means that clauses are built repetitively, until one of the stopping criteria is satisfied. After a clause has been built, the positive examples covered by it are removed from the training set. The search strategy in mFOIL is beam search, which at least partially alleviates the problem of getting into undesirable local maxima, which is typical of greedy hill-climbing search.

The search for a clause starts with the clause with an empty body. During the search, a small set of promising clauses found so far is maintained (the beam), as well as the best *significant* clause found so far. At each step of the search, the refinements of each clause in the beam (obtained by adding literals to their bodies) are evaluated using the search heuristic, and the best of their improvements[1] constitute the new beam. To enter the new beam a clause has to be *possibly significant*. The search for a clause terminates when the new beam becomes empty (no possibly significant improvements of the clauses in the current beam have been found at the current step). The best significant clause found so far is retained in the hypothesis if its expected accuracy is better than the default accuracy (the probability of the more frequent of the classes $\oplus$ or $\ominus$), estimated from the entire training set by the relative frequency estimate.

The significance test used in mFOIL is similar to the one used in CN2 [Clark and Niblett 1989] and is based on the likelihood ratio statistic [Kalbfleish 1979]. If a clause c covers $n(c)$ examples, $n^{\oplus}(c)$ of which are positive, the value of the statistic can be calculated as follows. Let $p_a(\oplus)$ and $p_a(\ominus)$ be the prior probabilities of classes $\oplus$ and $\ominus$,

---

[1] An improvement of clause c is a refinement c' of clause c, such that $A(c') > A(c)$.

estimated by the relative frequencies of positive and negative examples in the entire training set: $p_a(\oplus) = \frac{n^{\oplus}}{n}$ and $p_a(\ominus) = \frac{n^{\ominus}}{n}$. In addition, $p(\oplus|c) = \frac{n^{\oplus}(c)}{n(c)}$ is the expected accuracy defined as the probability that an example covered by clause c is positive, and $p(\ominus|c) = 1 - p(\oplus|c)$. Then we have

$$LikelihoodRatio(c) = 2n(c) \times (p(\oplus|c)log(\frac{p(\oplus|c)}{p_a(\oplus)}) + p(\ominus|c)log(\frac{p(\ominus|c)}{p_a(\ominus)}))$$

This statistic is distributed approximately as χ^2 with one degree of freedom. If its value is above a specified significance threshold, the clause is deemed significant.

A kind of pruning, implemented in mFOIL, is based on the following argument. Suppose a partial clause c covers $n^{\oplus}(c)$ positive examples. The best we can hope to achieve by refining this clause is a clause that covers $n^{\oplus}(c)$ positive and no negative examples. The likelihood ratio statistic would in this case have the value $-2n^{\oplus}(c) \times log(p_a(\oplus))$. If this value is less than the significance threshold, no significant refinement of this clause can be found. Otherwise, the clause is $possibly\ significant$.

The search for clauses is terminated when too few positive examples (possibly zero) are left for a generated clause to be significant or when no significant clause can be found with expected accuracy greater than the default. Demanding the expected accuracy (estimated by the heuristic value) to be higher than the default accuracy may cause constructing no clauses at all in domains where negative examples prevail. In such cases, the criterion of accuracy might be ommited and the expected accuracy of the constructed clauses disregarded, provided that they are significant. The search would then terminate when no further significant clause could be constructed. Afterwards, the clauses with very low accuracy might be discarded. This approach was applied in the finite element mesh design domain, where the default accuracy of the class $\ominus$ is above 90%.

9.4 Implementation details

Several parameters are in use in mFOIL, which determine the search heuristic used, the width of the beam in the beam search, the level of significance applied to the induced clauses and the use of rectified or

Parameter	Values	Default value
heuristic	Laplace m	Laplace
m	non-negative real	0
beam	positive integer	5
significance	non-negative real	6.64
variables	different identical	different

Table 9.1: mFOIL parameters and their values.

non-rectified literals. The value of a parameter, say p, is set by using the command $set(p, value)$. For example, to override the default constraint of using rectified literals only, the command $set(variables, identical)$ is used. To set the search heuristic to the m-estimate with $m = 2$, the following two commands are used: $set(heuristic, m), set(m, 2)$. A list of the parameters used in mFOIL with the possible values, as well as the default values used, is given in Table 9.1.

The parameter *heuristic* determines the search heuristic used and can be either the Laplace or the m-estimate, the default being the Laplace estimate. The parameter m, with a default value zero, determines the value of m to be used in the m-estimate. The default beam width (stated by the parameter *beam*) is five, i.e., five clauses are retained in the beam at each search step. The default $significance$ threshold 6.64 corresponds to a significance level of 99%.

mFOIL is implemented in Quintus Prolog and runs on SUN SPARC-Station 1. The current implementation is fairly inefficient. The runing time for the KRK endgame domain (see Chapter 10) is in the order of two minutes, while the running time for the finite element mesh design domain (Chapter 12) is in the order of two hours. The corresponding times for FOIL, implemented in C, are in the order of seconds and minutes, respectively.

Several simplifications have been made as compared to FOIL. Unlike FOIL2.1 [Quinlan 1991], mFOIL makes no distinction between determinate and non-determinate literals. Also, instead of using partial orderings to prevent infinitely recursive clauses, a very simple constraint is used. Namely, a clause $T \leftarrow Q$ must not contain a literal identical to T in its body, which does not always prevent infinite recursion.

10

Experiments in learning relations from noisy examples

The ultimate performance test for ILP algorithms should be their application to practical domains involving imperfect data. However, it is difficult to measure the level of noise (imperfectness) in such domains. Furthermore, while several standard data sets obtained from practical settings are now available for attribute-value learning systems [Clark and Boswell 1991], not many datasets of this kind are available for the ILP case. The problem of learning illegal positions in the KRK chess endgame, on the other hand, has been used as a testbed for most empirical ILP systems (FOIL, FOCL, GOLEM, LINUS). By introducing a controlled amount of noise in the training examples artificially, it is possible to assess the dependence of the learning performance on the noise level more accurately.

This chapter presents the experimental results of applying LINUS, FOIL and mFOIL to the problem of learning illegal positions in the KRK chess endgame, introduced in Section 6.5. To examine the effects of noise, various amounts of noise were added to the examples. Two groups of experiments were performed. In the first group, noise was added to the examples non-incrementally. In this case, we compared the performance of LINUS using ASSISTANT and NEWGEM to FOIL (FOIL0). In the second group of experiments, noise was incrementally added to the set of training examples. The performance of mFOIL is compared to FOIL2.1 [Džeroski 1991, Džeroski and Bratko 1992a, 1992b]. Finally, a comparison of LINUS, mFOIL and FOIL2.1 on the non-incremental noise version of the data is presented.

10.1 Introducing noise in the examples

To examine the effects of noise in the examples on the accuracy of the induced hypotheses, several series of experiments were conducted in which the examples were corrupted with different amounts of noise. In the experiments, the small sets of 100 examples each (see Section 6.5) were used as training sets, and the five large sets were merged into one testing set of 5000 examples.

In the following, $x\%$ of noise in argument A means that in $x\%$ of examples, the value of argument A was replaced by a random value of the same type from a uniform distribution [Quinlan 1986]. For example, 5% of noise in argument A means that its value was changed to a random value in 5 out of 100 examples, independently of noise in other arguments. The class variable was treated as an additional argument [Clark and Niblett 1989] when introducing noise. The percentage of introduced noise varied from 5% to 80% as follows: 5%, 10%, 15%, 20%, 30%, 50%, and 80%. Background knowledge was not corrupted with noise.

In the experiments described in Section 10.2, noise was added to the training examples independently for each percentage, i.e., in a non-incremental fashion. Suppose we had a training set of 100 examples. To add 10% of noise in the values of argument A, all correct values were taken and 10 of them were randomly selected and corrupted. In the experiments described in Section 10.3, noise was added incrementally. This means that 5 values were first corrupted out of the 100 correct values. Next, out of the 95 remaining non-noisy values, additional 5 were corrupted to achieve 10% of noisy values. Five more examples (out of the uncorrupted 90) were replaced with noisy examples to achieve 15% of noise in argument A.

It should be noted that the effects of noise in the arguments are combined when applying background predicates to them. For instance, consider the predicate $(WRf = BKf)$, which can take values $true$ and $false$, denoting that the White Rook is, or is not, on the same file as the Black King. When 30% of values of arguments WRf and BKf are *noisy*, then 51% ($0.51 = 0.3 + 0.3 - 0.3 \times 0.3$) of the values of predicate $(WRf = BKf)$ are *noisy*. Furthermore, these values are corrupted in a different way from the way the two arguments are corrupted. If we corrupted the value of predicate $(WRf = BKf)$ in the same way as the values of arguments WRf and BKf, the distribution of the corrupted values would be uniform, i.e., $true$ (1/2) and $false$ (1/2). However, if either of WRf and BKf is corrupted by assigning random values to it, the resulting distribution for the corrupted values of $(WRf = BKf)$ is $true$ (1/8) and $false$ (7/8).

Noisy values are not necessarily incorrect. Since they are chosen randomly, it may happen that the correct value is selected. Furthermore, even if an argument value is incorrect, the resulting training example may still be correctly classified. This contributes to the difficulty of analyzing the effects of noise on ILP [Džeroski and Lavrač 1991b].

10.2 Experiments with LINUS

To examine the effects of noise, three series of experiments were conducted with LINUS and FOIL, introducing noise in the training examples in three different ways. First, noise was added in the values of the arguments of the target relation, then in the values of the class variable, and finally, in both the argument and the class variable values. Noise was added non-incrementally in all three cases.

Each experiment was performed using one of the three ways of introducing noise and a chosen noise level. Noise was first introduced in the training sets. A set of clauses was then induced from each of the training sets using LINUS and FOIL. Finally, the classification accuracy of the sets of clauses was tested on the testing set, and the average of the five results was computed.

In the experiments, ASSISTANT and NEWGEM were used as attribute-value learning algorithms in LINUS. Within LINUS, ASSISTANT was run with its default parameters, which set the noise-handling mechanism to tree pre-pruning only. The NEWGEM parameters were set to handle perfect data, i.e., to minimize the number of literals in a clause and to maximize the number of examples uniquely covered by a clause. This was done in order to compare the performance of algorithms which can handle imperfect data to the performance of those that can not. An early version of FOIL (FOIL0) was used in the experiments. The background knowledge used in FOIL and LINUS was the same as described in Section 6.5.

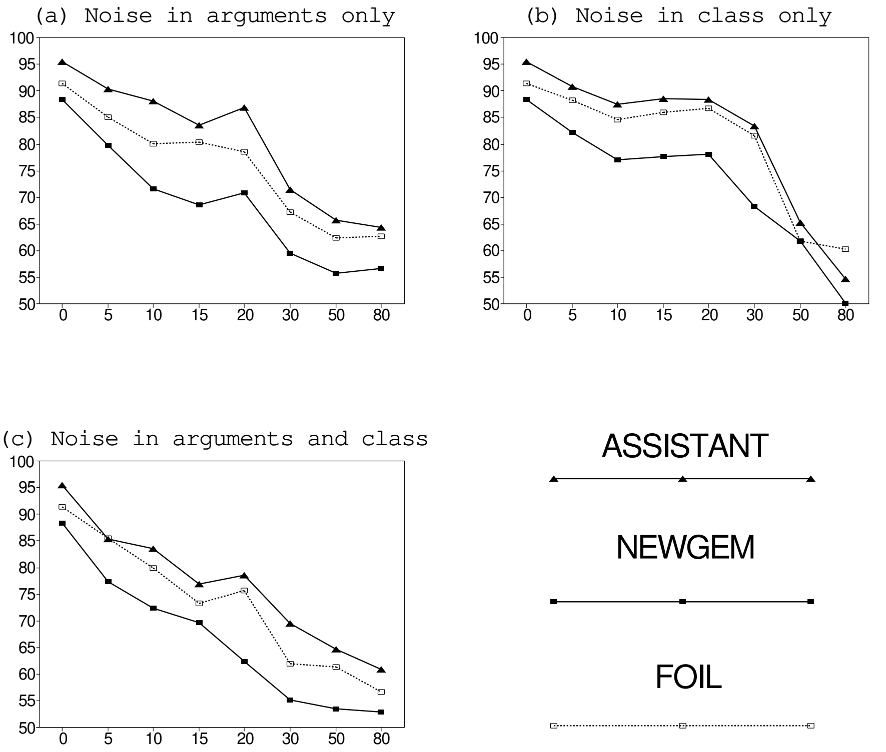


Figure 10.1: Accuracy vs. noise in training examples.

Figure 10.1(a) gives the results of the experiments with noise introduced only in the arguments, Figure 10.1(b) with noise in the class variable only, and Figure 10.1(c) with noise introduced in the arguments and in the class variable at the same time.

Noise affected adversely the classification accuracy of both systems. The classification accuracy decreased as the percentage of noise increased: the most when it was introduced in both the arguments and the class variable, and the least when introduced in the class variable only. LINUS using ASSISTANT achieved a better classification accuracy than FOIL. As NEWGEM used no mechanism for handling imperfect data, LINUS using NEWGEM performed worse than FOIL.

Analysis of the clauses induced by LINUS using ASSISTANT and

FOIL shows that with the increase of the percentage of noise in the training examples, the following can be observed:

- The average clause length increases in FOIL and decreases in LINUS using ASSISTANT.
- The average number of examples covered by a clause slowly decreases in FOIL and increases in LINUS using ASSISTANT.
- For both systems, the number of positive examples not covered by the hypothesis rises, faster for LINUS using ASSISTANT than for FOIL.

All these facts indicate that FOIL slightly overfits the training examples, which results in a lower performance on unseen cases. This is mainly due to the defficiency of the encoding length restriction discussed in Section 8.4.

Two further remarks about the results have to be made. First, there is an increase in the classification accuracy at 20% noise level. This is due to the non-incremental nature of increasing noise in the training data. This phenomenon disappeared in the experiments where noise was added incrementally [Džeroski 1991]. Second, in the training data, 1/3 of examples are positive (illegal positions) and 2/3 are negative (legal positions). A learning system with a perfect noise-handling mechanism should achieve a classification accuracy which must not be lower than the accuracy achieved by a default classification rule that classifies each example into the majority class (in our case into class *legal*).

We can notice that in Figure 10.1 the classification accuracy drops under the 66% majority class accuracy. This is understandable when noise is introduced in the class variable (Figure 10.1(b)) and both in the arguments and the class variable (Figure 10.1(c)), since the noise changes the class distribution, but not in the case where only arguments are corrupted (Figure 10.1(a)). This indicates that the training examples are being overfitted. To explain this phenomenon, we adapt an argument from Clark and Boswell [1991]. At 100% noise in the arguments, the arguments and the class are completely independent. The maximally overfitted rules (one per example) would then be correct with probability 1/3 if the class is *illegal* and with probability

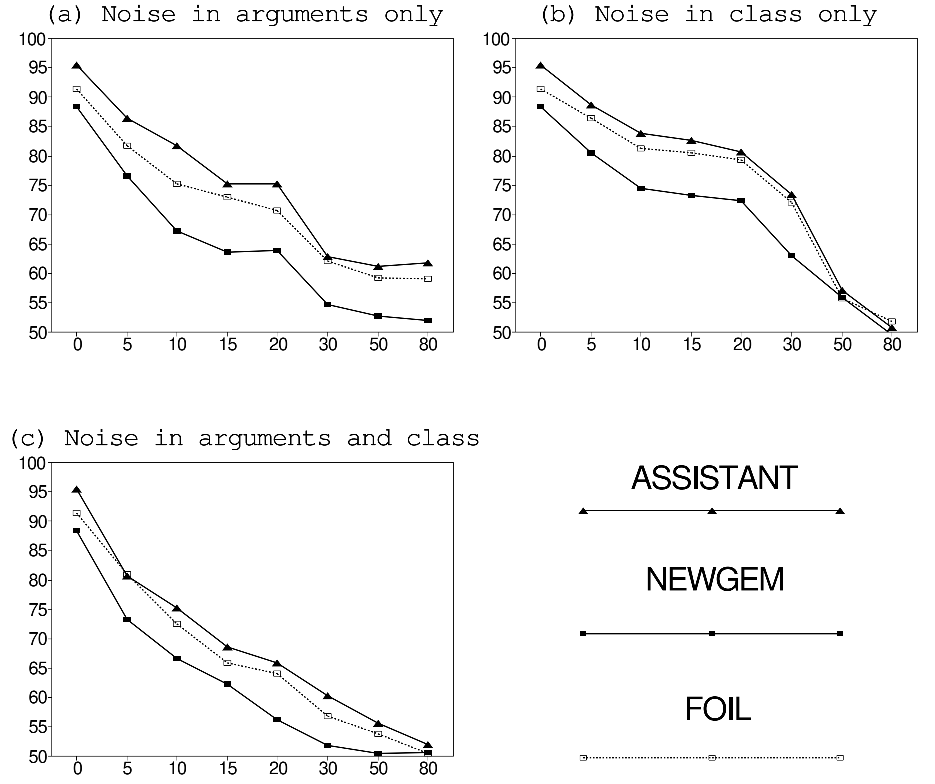


Figure 10.2: Accuracy vs. noise in training and testing examples.

2/3 if the class is *legal* (the default probabilities of positive and negative examples). The probability of a correct answer would then be $\frac{2}{3} \times \frac{2}{3} + \frac{1}{3} \times \frac{1}{3} = \frac{5}{9}$, or 55%, which is lower than the 66% default accuracy. The overfitting observed in our experiments reflects behavior between these two extremes.

All experiments described in this section were repeated with noisy testing sets [Džeroski and Lavrač 1991a]. Noise was introduced in the testing sets in exactly the same way as in the training sets (same amounts of noise in the arguments, the class variable or both). The classification accuracy of both LINUS and FOIL dropped significantly, but their relative performance remained unchanged, i.e., LINUS using ASSISTANT achieved better and LINUS using NEWGEM worse classification accuracy than FOIL.

Figure 10.2 shows the results for the case where noise was introduced in both the training and the testing examples. Figure 10.2(a) gives the results of the experiments with noise introduced only in the arguments, Figure 10.2(b) with noise in the class variable only, and Figure 10.2(c) with noise introduced in the arguments and in the class variable at the same time.

To summarize, we compared the performance of LINUS and FOIL on the problem of learning illegal positions in a chess endgame. Several series of experiments were conducted, where various amounts of non-incremental noise were added to the training examples. Both LINUS using ASSISTANT and FOIL performed well, compared to LINUS using NEWGEM, which used no mechanism for handling imperfect data. As a result of the successful tree-pruning mechanism used in ASSISTANT and the defficiency of the encoding length restriction used in FOIL, LINUS using ASSISTANT achieved better classification accuracy than FOIL.

10.3 Experiments with mFOIL

This section discusses the performance of mFOIL and FOIL on the same task of learning illegal chess endgame positions from noisy examples. In contrast with the experiments in Section 10.2, noise was artificially added to the correct examples in an incremental fashion. mFOIL using the m-estimate as a search heuristic achieved better classification accuracy than FOIL. We also make a comparison of FOIL and mFOIL with LINUS using ASSISTANT on the training sets with non-incremental noise.

A recent version of FOIL (FOIL2.1), described in Quinlan [1991], was used in the experiments. It was run with its default parameters. Unlike FOIL0, FOIL2.1 can use the information about the types of predicate arguments. Thus, all the systems use the following background predicates: $adjacent\_file(X, Y)$ and $less\_file(X, Y)$ with arguments of type $file$ (with values a to h), $adjacent\_rank(X, Y)$ and $less\_rank(X, Y)$ with arguments of type $rank$ (with values 1 to 8), and equality $X = Y$, used for both types of arguments.

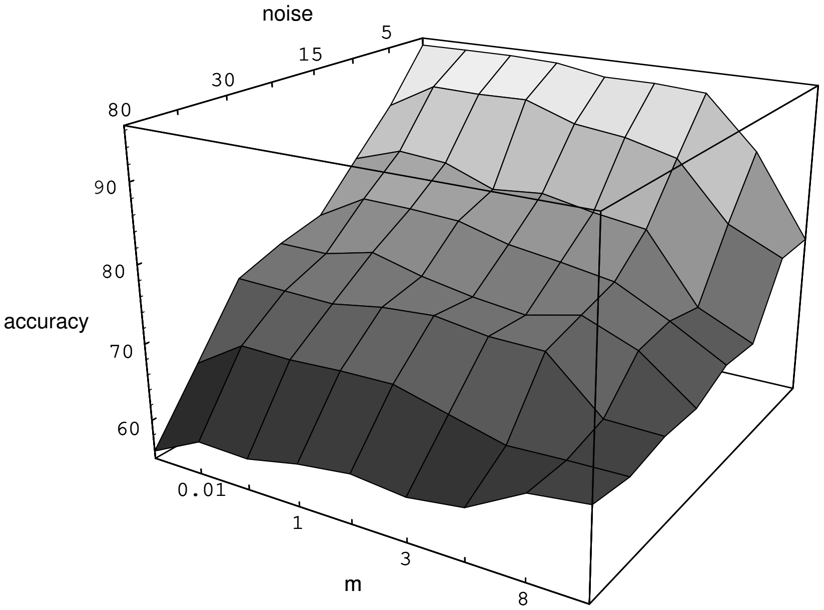
The parameters in mFOIL were set as follows. The beam width was set to five (default) and the significance threshold was set to zero (no significance was tested). This corresponds to the parameter settings used in Clark and Boswell [1991], which turned out to work well. The construction of clauses was stopped when a clause with expected accuracy (estimated by the search heuristic) lower than the default accuracy was constructed. The Laplace estimate and the m-estimate, with several different values for m, were used as search heuristics. The values for m were as follows: 0, 0.01, 0.5, 1, 2, 3, 4, 8 and 16, a subset of the values used by Cestnik and Bratko [1991].

The three series of experiments with mFOIL were repeated with a significance threshold of 99%. Finally, some experiments were performed on training examples with non-incremental noise. LINUS using ASSISTANT, FOIL and mFOIL with $m = 0.01$ were applied in this case to examples corrupted with all three kinds of noise.

Figure 10.3 gives the results of the experiments with noise introduced only in the values of the arguments, Figure 10.4 with noise in the values of the class variable only, and Figure 10.5 with noise introduced in the values of the arguments and the class variable at the same time. The results are the average over five experiments.

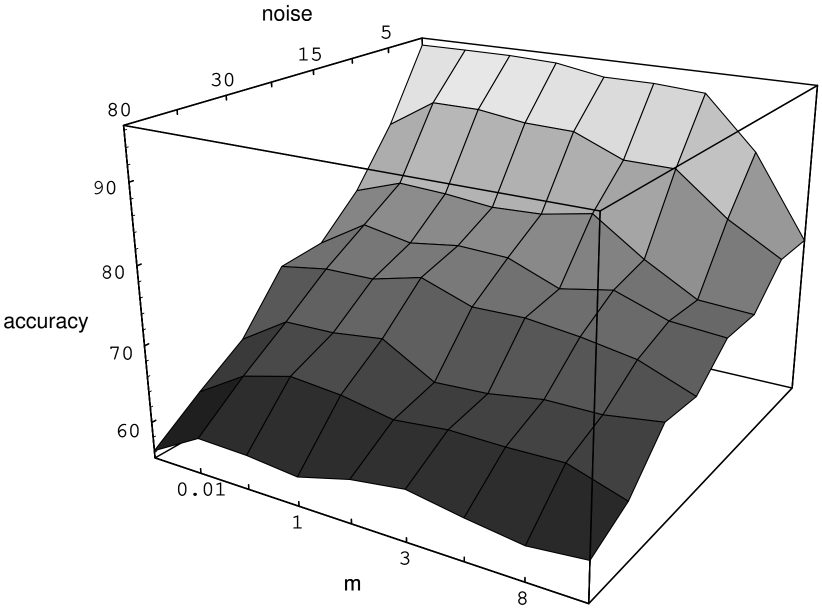
Noise affected adversely the classification accuracy of both FOIL and mFOIL. The classification accuracy decreased as the percentage of noise increased: the most when it was introduced in both the values of the arguments and the class variable, and the least when introduced in the values of the class variable only. As noise was introduced incrementally, accuracy dropped monotonically when the amount of noise increased. In earlier experiments with non-incremental noise [Džeroski and Lavrač 1991a], described in Section 10.2, an increase in accuracy was observed when noise increased from 15% to 20%. As such an increase is not present in the results of experiments with incremental noise, we may conclude that the increase was due to chance in the non-incrementally added noise, and not to some specific properties of the learning systems involved.

If the best m is chosen for each percentage of noise, the results achieved by mFOIL are better than the ones achieved by FOIL (see Figure 10.6) or mFOIL with the Laplace estimate. Moreover, if mFOIL uses the m-estimate with $m = 0.01$, for which best performance is



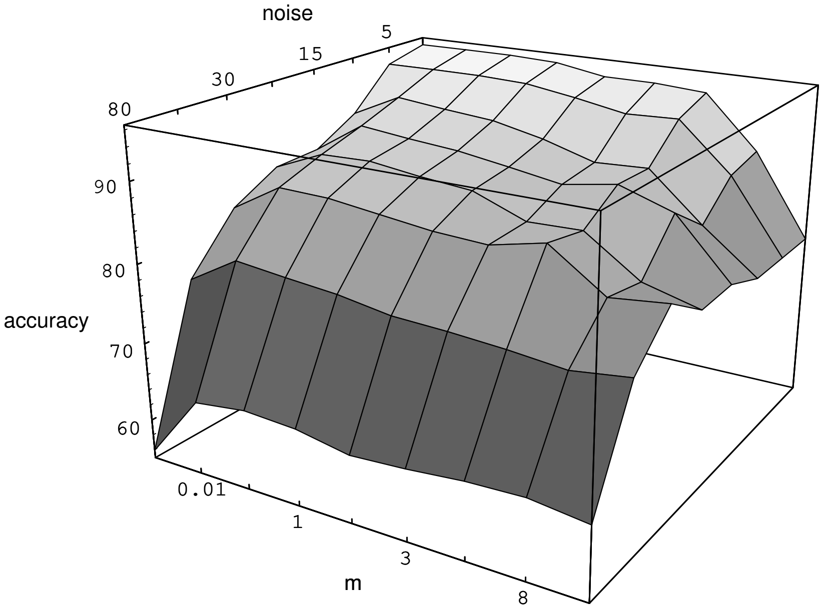
	Noise							
Estimate	0	5	10	15	20	30	50	80
Laplace	95.14	90.11	80.41	78.32	73.66	72.57	67.91	59.20
$m = 0.0$	95.53	88.27	82.45	76.50	74.98	72.80	64.27	55.86
$m = 0.01$	95.57	91.87	84.59	80.06	74.95	72.61	68.32	59.10
$m = 0.5$	95.57	91.71	84.04	79.92	76.49	72.42	68.20	58.91
$m = 1.0$	95.33	91.86	81.52	79.54	74.74	73.53	68.54	60.34
$m = 2.0$	94.21	89.53	82.16	77.72	73.52	74.04	68.64	61.21
$m = 3.0$	94.21	88.71	80.84	76.78	72.72	73.06	66.79	60.50
$m = 4.0$	93.78	86.98	79.88	75.66	74.30	72.91	64.96	61.47
$m = 8.0$	86.92	79.44	70.98	72.14	71.50	66.45	65.23	65.40
$m = 16.0$	76.22	76.22	67.72	68.16	66.07	66.30	65.38	66.30
Best m	95.57	91.87	84.59	80.06	76.49	74.04	68.64	66.30

Figure 10.3: mFOIL: Accuracy vs. noise in the arguments.



	Noise							
Estimate	0	5	10	15	20	30	50	80
Laplace	95.14	92.71	88.90	86.40	84.58	83.75	73.74	60.25
$m = 0.0$	95.53	94.19	88.86	85.74	85.39	82.19	75.37	55.89
$m = 0.01$	95.57	94.26	92.02	89.96	88.14	86.01	79.20	64.17
$m = 0.5$	95.57	94.26	91.34	89.28	88.37	85.88	78.54	65.04
$m = 1.0$	95.33	94.02	90.78	88.74	87.55	85.12	77.97	64.67
$m = 2.0$	94.21	93.01	89.36	87.76	86.96	84.37	76.94	63.44
$m = 3.0$	94.21	91.80	87.38	86.70	84.39	84.07	76.69	63.87
$m = 4.0$	93.78	92.09	87.73	87.88	84.61	85.66	76.26	64.54
$m = 8.0$	86.92	85.81	81.65	85.60	79.84	80.84	75.61	64.81
$m = 16.0$	76.22	76.22	76.22	78.22	78.02	81.67	76.56	63.96
Best m	95.57	94.26	92.02	89.96	88.37	86.01	79.20	65.04

Figure 10.4: mFOIL: Accuracy vs. noise in the class.

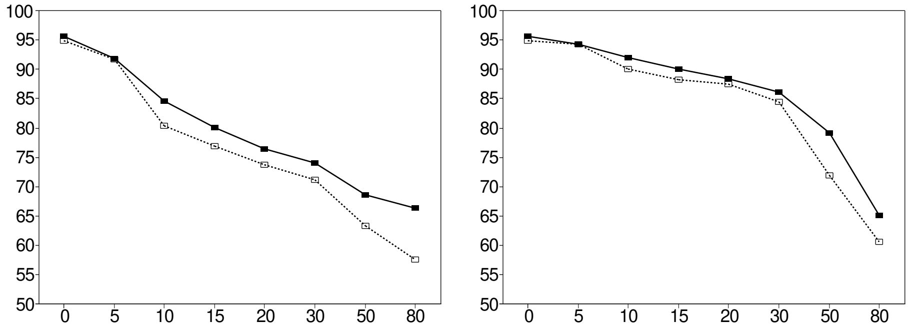


	Noise							
Estimate	0	5	10	15	20	30	50	80
Laplace	95.14	85.36	75.76	75.08	71.69	64.22	61.38	57.58
$m = 0.0$	95.53	85.56	77.92	72.68	71.95	64.64	60.62	56.22
$m = 0.01$	95.57	89.64	80.16	76.58	73.01	68.64	64.53	59.84
$m = 0.5$	95.57	89.62	79.83	75.38	73.09	68.80	66.27	59.65
$m = 1.0$	95.33	88.72	79.17	76.36	74.74	69.65	65.68	58.98
$m = 2.0$	94.21	88.53	79.48	76.10	72.38	65.73	64.42	60.79
$m = 3.0$	94.21	85.74	80.79	73.44	72.53	66.09	65.11	61.73
$m = 4.0$	93.78	85.66	76.05	74.75	71.74	67.18	65.00	60.52
$m = 8.0$	86.92	80.21	72.16	72.33	70.57	67.19	65.14	59.54
$m = 16.0$	76.22	76.22	71.77	71.79	67.79	68.21	62.76	60.24
Best m	95.57	89.64	80.79	76.58	74.74	69.65	66.27	61.73

Figure 10.5: mFOIL: Accuracy vs. noise in the arguments and the class.

(a) Noise in arguments only

(b) Noise in class only



(c) Noise in arguments and class

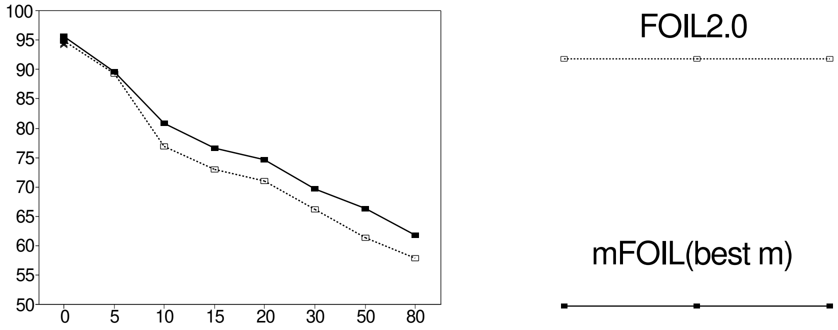


Figure 10.6: Accuracy vs. noise: mFOIL with best m and FOIL.

achieved, it still performs better than with the Laplace estimate, as well as better than FOIL. The differences between mFOIL with $m = 0.01$ and FOIL at 10% and 15% of noise of all kinds are significant at least at the 90% level, according to the t-test for dependent samples.

For a fixed percentage of noise, the behavior of the classification accuracy of mFOIL for different values of m is similar to the behavior of accuracy of CN2 with the m estimate for different values of m. In most cases it grows from $m = 0$ to $m = 0.01$, grows until a maximum is reached and then falls. This behavior is illustrated in Figure 10.7, where the accuracies are given for training examples with noise in both arguments and class.

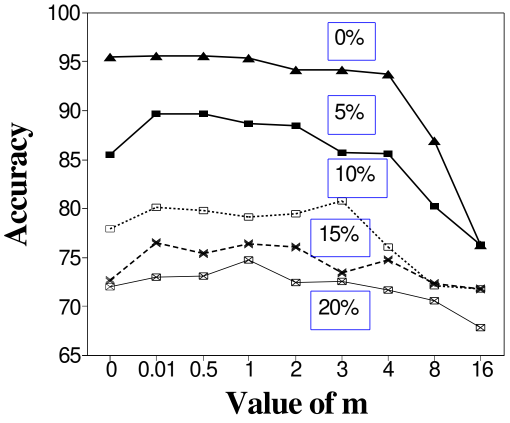


Figure 10.7: Behavior of mFOIL accuracy for different values of m.

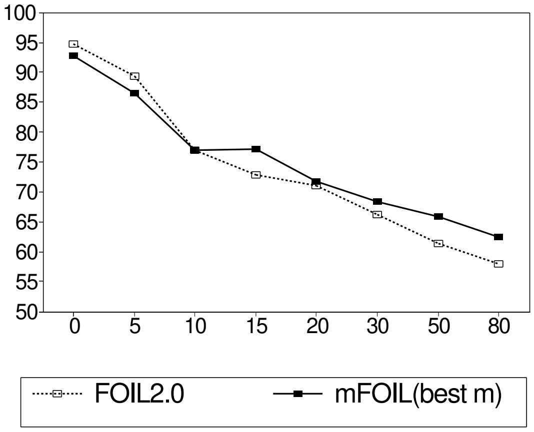


Figure 10.8: Accuracy of mFOIL with best m and 99% significance threshold.

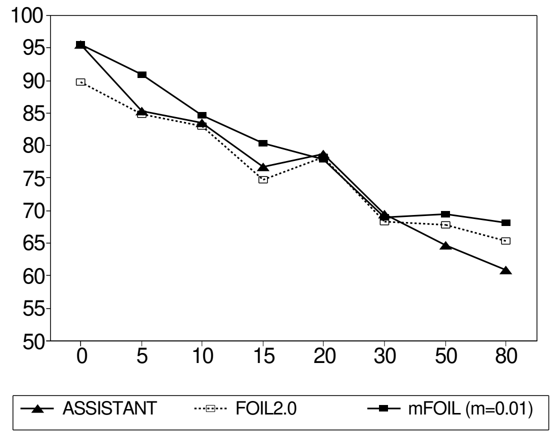


Figure 10.9: LINUS, FOIL and mFOIL accuracy: non-incremental noise.

If a 99% significance threshold is applied the performance of mFOIL with best m decreases. Similar results have been reported for CN2 [Clark and Boswell 1991]. However, it still performs better than FOIL for noise above 15% in arguments or both arguments and class. A comparison of accuracies of mFOIL with best m and FOIL, for noise in arguments and class is given in Figure 10.8.

Finally, we compare the performance of mFOIL (with $m = 0.01$, and a zero significance threshold), FOIL and LINUS (using ASSISTANT) on training examples with non-incremental noise. The results for noise in arguments and class are given in Figure 10.9. mFOIL achieved better classification accuracy than both FOIL and LINUS.

Part IV

Applications of ILP

11

Learning rules for early diagnosis of rheumatic diseases

Correct diagnosis in the early stage of a rheumatic disease is a difficult problem. Having passed all the investigations, many patients cannot be reliably diagnosed after their first visit to the specialist. The reason for this is that symptoms, clinical manifestations, laboratory and radiological findings of various rheumatic diseases are similar and not specific. Diagnosis can also be incorrect because of the subjective interpretation of anamnestic, clinical, laboratory and radiological data [Pirnat et al. 1989].

This diagnostic domain was used earlier [Pirnat et al. 1989, Karalič and Pirnat 1990] in experiments with the decision-tree learning system ASSISTANT [Cestnik et al. 1987] (see Section 5.2.2). In this real-life diagnostic domain, the use of ASSISTANT was very appropriate since it can deal with incomplete (missing) and erroneous (noisy) data.

This chapter describes the application of LINUS to the problem of learning rules for early diagnosis of rheumatic diseases. In addition to the attribute-value descriptions of patient data, LINUS was also given background knowledge provided by a medical specialist in the form of typical co-occurences of symptoms. The experiments described in this chapter were done with LINUS using CN2 [Clark and Boswell 1991] (see Section 5.2.4). The CN2 algorithm also has the ability to cope with noisy data, but induces descriptions in the form of if-then rules. It turns out that CN2 is better suited for the use of the particular medical background knowledge in the induction of diagnostic rules.

Our study shows how the noise-handling mechanisms of CN2 and the ability of LINUS to use background knowledge affect the performance (i.e., classification accuracy and information content) and the complexity of the induced diagnostic rules. In addition, a medical evaluation of the rules shows that the use of background knowledge in LINUS improves the quality of rules. The performance of LINUS using CN2 is also compared to the performance LINUS using ASSISTANT, measured in earlier experiments in this domain [Lavrač et al. 1991b].

First, we present the diagnostic problem and give the background knowledge to be used in the induced diagnostic rules. Next, the results of the experiments and the medical evaluation of induced rules are presented. The chapter concludes with a discussion and directions for further work.

11.1 Diagnostic problem and experimental data

Data about 462 patients were collected at the University Medical Center in Ljubljana. There are over 200 different rheumatic diseases which can be grouped into three, six, eight or twelve diagnostic classes. As suggested by a specialist, eight diagnostic classes were considered. Table 11.1 shows the names of the diagnostic classes and the numbers of patients belonging to each class.

The experiments were performed on anamnestic data, without taking into account data about patients' clinical manifestations, laboratory and radiological findings. The sixteen anamnestic attributes are as follows: sex, age, family anamnesis, duration of present symptoms, duration of rheumatic diseases, joint pain (arthrotic, arthritic), number of painful joints, number of swollen joints, spinal pain (spondylotic, spondylitic), other pain (headache, pain in muscles, thorax, abdomen, heels), duration of morning stiffness, skin manifestations, mucosal manifestations, eye manifestations, other manifestations and therapy.

Out of 462 patient records only 8 were incomplete; 12 attribute values were missing (for attributes sex and age). This was not problematic since LINUS using CN2 can handle missing data.

11.2 Medical background knowledge

The available patient data were augmented with additional diagnostic knowledge. A specialist for rheumatic diseases has provided his knowledge about the typical co-occurrences of symptoms. Six typical

Class	Name	Number of patients
$A1$	degenerative spine diseases	158
$A2$	degenerative joint diseases	128
$B1$	inflammatory spine diseases	16
$B234$	other inflammatory diseases	29
C	extra-articular rheumatism	21
D	crystal-induced synovitis	24
E	non-specific rheumatic manifestations	32
F	non-rheumatic diseases	54

Table 11.1: The eight diagnostic classes and the corresponding numbers of patients.

groupings of symptoms were suggested by the specialist.

- The first grouping relates the joint pain and the duration of morning stiffness. The characteristic combinations are given in the table below; all other combinations are insignificant or irrelevant:

Joint pain	Morning stiffness
no pain	≤ 1 hour
arthrotic	≤ 1 hour
arthritic	> 1 hour

- The second grouping relates the spinal pain and the duration of morning stiffness. The following are the characteristic combinations:

Spinal pain	Morning stiffness
no pain	≤ 1 hour
spondylotic	≤ 1 hour
spondylitic	> 1 hour

- The third grouping relates sex and other pain. Indicative is the pain in the thorax or in the heels for male patients, all other combinations are non-specific:

Sex	Other pain
male	thorax
male	heels

- The fourth grouping relates joint pain and spinal pain. The following are the characteristic co-occurences:

Joint pain	Spinal pain
no pain	spondylotic
arthrotic	no pain
no pain	spondylitic
arthritic	spondylitic
arthritic	no pain
no pain	no pain

- The fifth grouping relates joint pain, spinal pain and the number of painful joints:

Joint pain	Spinal pain	Painful joints
no pain	spondylotic	0
arthrotic	no pain	$1 \leq \text{joints} \leq 30$
no pain	spondylitic	0
arthritic	spondylitic	$1 \leq \text{joints} \leq 5$
arthritic	no pain	$1 \leq \text{joints} \leq 30$
no pain	no pain	0

- The sixth grouping relates the number of swollen joints and the number of painful joints:

Swollen joints	Painful joints
0	0
0	$1 \leq \text{joints} \leq 30$
$1 \leq \text{joints} \leq 10$	$0 \leq \text{joints} \leq 30$

The above background knowledge is encoded in the form of utility functions, introducing specific function values for each characteristic combination of symptoms. For example, the third grouping of characteristic symptoms is implemented by the utility function given below:

```
% grouping3( sex/input, location/input, grouping3_value/output)
grouping3( male, thorax, male_thorax ) :- !.
grouping3( male, heels, male_heels ) :- !.
grouping3( _, _, irrelevant ).
```

The characteristic combinations of input attribute values yield utility function values which are mnemonic and understandable, e.g., *male_thorax* and *male_heels* in the above utility function. These names are 'artificial' (not used by specialists), but they represent meaningful co-occurrences of symptoms which have their role in expert diagnosis. All the other combinations yield the same function value *irrelevant*.

The choice of functions instead of utility predicates is based on the same argument. It is important for a specialist to know the exact value of the function (i.e., the exact combination of symptoms), and not only to know that some combination of values of individual attributes has occurred (which would be the case if this knowledge were encoded in the form of utility predicates having only values *true* and *false*).

11.3 Experiments and results

In the induction of medical diagnostic rules, LINUS was used in the class learning mode (see Section 5.4.4). CN2 [Clark and Boswell 1991] was used within LINUS to induce rules for each of the eight diagnostic classes, taking into account the background knowledge in the form of utility functions.

In order to evaluate the effects of background knowledge and noise-handling mechanisms on the classification accuracy and the complexity of the induced rules, two groups of experiments were performed: one group on the whole set of patient data, and the other on ten different partitions of the data set into training and testing examples. In each group of experiments, the experiments were further designed along another two dimensions. Namely, in one half of the experiments the background knowledge described in Section 11.2 was used in the learning process. Each of the groupings contributed an additional attribute, giving a total of six new attributes. The set of attributes used for learning with CN2 thus consisted of 16 initial and 6 new attributes. In the other half of the experiments no background knowledge was used, and the set of attributes consisted of the 16 initial attributes only. Along the other dimension, the significance test noise-handling mechanism in CN2 (with a significance level of 99%) was used in one half of the experiments and not in the other half.

The classification accuracy was measured as the percentage of examples correctly classified by a rule set. The complexity was measured by the number of rules in the set, as well as the total number of literals (conditions) appearing in all rules in the set. Finally, the information content, i.e., relative information score [Kononenko and Bratko 1991] was measured, which takes into account the difficulty of the classification problem. This measure of classifier performance is briefly described below. To calculate the relative information score, a suitably modified implementation of CN2 [Džeroski et al. 1992a] was used.

The most general form of the answer of a classifier, given a testing example, is a probability distribution over the classes of the domain. Let the correct class of example e_k be C, its prior probability $p(C)$ and the probability returned by the classifier $p'(C)$. The *information score* of this answer is

$$IS(e_k) = \begin{cases} -log(p(C)) + log(p'(C)), & p'(C) \geq p(C) \\ log(1 - p(C)) - log(1 - p'(C)), & p'(C) < p(C) \end{cases}$$

As $IS(e_k)$ indicates the amount of information about the correct classification of e_k gained by the classifier's answer, it is positive if $p'(C) > p(C)$, negative if the answer is misleading ($p'(C) < p(C)$) and zero if $p'(C) = p(C)$.

The *relative information score* $IS_r(\mathcal{E})$ of the answers of a classifier on a testing set consisting of examples $\mathcal{E} = \{e_1, e_2, ..., e_n\}$ belonging to one of classes $C_1, C_2, ..., C_N$ can be calculated as the ratio of the average information score of the answers and the entropy of the prior distribution of classes.

$$IS_r(\mathcal{E}) = \frac{\frac{1}{n} \times \sum_{k=1}^{n} IS(e_k)}{-\sum_{i=1}^{N} p(C_i) \times log(p(C_i))}$$

The relative information score of the 'default' classifier, which always returns the prior probability distribution, is zero. If the classifier always guesses the right class and is absolutely sure about it ($p'(C) = 1$), then $IS_r(\mathcal{E}) = 1$, provided the class distributions of the training and testing sets are the same.

To scale the evaluation of a classifier to the difficulty of the problem, the relative information score takes into account the prior probability of classes (diagnoses). Namely, a correct classification into a more probable diagnosis provides less information than a correct classification into a rare diagnosis, which is only represented by a few training examples. For example, in domains where one of the diagnostic classes is highly likely, it is easy to achieve high classification accuracy. The 'default' classifier that assigns the most common diagnosis to all patients would in that case have undeservedly high classification accuracy. However, its relative information score is zero, indicating that the classifier provides no information at all.

11.3.1 Learning from the entire training set

In the first group of experiments, the data about all 462 patients were used. Four experiments were performed in which the use/non-use of background knowledge and the use/non-use of the significance test were varied. The classification accuracy and the relative information score (both calculated on the training set), as well as the complexity of the induced rule sets, were measured. Table 11.2 gives the results of these experiments.

The results show that the use of background knowledge increased the accuracy of the induced rules on the training set. More importantly, it also increased the information content of the rules. The total number of literals appearing in all rules increased with the use of background knowledge when the significance test was used, and remained unchanged when the significance test was not used.

Background knowledge	Significance test	Accuracy	Relative inf. score	Number of rules	Number of literals
no	no	62.8%	49%	96	302
no	yes	51.7%	22%	30	102
yes	no	72.9%	59%	96	301
yes	yes	52.4%	30%	38	120

Table 11.2: Classification accuracy, relative information score (both measured on the training set itself) and complexity of rules derived from all 462 examples.

The use of the significance test greatly decreased the size of the rule sets, both with and without using background knowledge. However, it also caused a decrease in the classification accuracy on the training set. This is natural, since the main function of this noise-handling mechanism is to prevent rules from overfitting the training set. Although this decreases the classification accuracy on the training set, it usually increases the classification accuracy on unseen cases (see Table 11.5).

All groupings appear in the induced rules. In the rules induced with no significance test, the most common groupings are $grouping5$ with thirteen and $grouping4$ with twelve occurences; $grouping2$ occurs five times, $grouping1$ and $grouping6$ four times each, and $grouping3$ occurs twice. When the significance test is used, the number of occurences decreases, as the number of rules/literals decreases drastically. In this case, $grouping1$ occurs twice, $grouping2$ three times, $grouping3$ and $grouping6$ once each, $grouping5$ seven times and $grouping4$ nine times. The most common groupings (4 and 5) combine spinal pain, the most informative attribute, to other relevant features.

To summarize, all utility functions from the background knowledge appeared in the induced rules. The use of background knowledge improved both the classification accuracy and the relative information score of the induced rules at the cost of a slight increase of rule complexity. In the following section, a medical evaluation of the effects of background knowledge on the induced rules is presented.

11.3.2 Medical evaluation of diagnostic rules

The induced diagnostic rules were shown to a specialist for rheumatic diseases who found most of the rules meaningful and understandable. The specialist evaluated the entire set of induced rules according to the following procedure. For each of the conditions in a rule, one point was given to the rule if the condition was in favour of the diagnosis made by the rule, minus one point if the condition was against the diagnosis and zero points if the condition was irrelevant to the diagnosis. The mark of a rule was established as the sum of the points for all conditions in the rule.

The actual marks ranged from minus one to three. Intuitively, mark 3 was given to rules which are very characteristic for a disease and could be even published in a medical book. Mark 2 was given to good, correct

Class	Number of rules with mark					Total no. of rules	Average mark
	3	2	1	0	-1		
A1			4	1	2	7	0.29
A2			3	2	1	6	0.33
B1		1	2			3	1.33
B2			3	1		4	0.75
C			1	2		3	0.33
D		1	2			3	1.33
E				3		3	0.00
F				1		1	0.00
Overall	0	2	15	10	3	30	0.53

Table 11.3: Medical expert evaluation of rules induced without background knowledge.

rules. Mark 1 was given to rules which are not wrong, but are not too characteristic for the diagnosis. Mark 0 was given to rules which are possible according to the specialist's knowledge; however, they reflect a coincidential combination of features rather than a characteristic one. A negative mark -1 was given to misleading rules, which actually indicate that the diagnosis is not likely.

Two sets of rules were evaluated, the ones induced with and without the use of background knowledge, both induced using the significance test. Figure 11.1 shows some rules which were evaluated as best by the specialist and their marks. They were induced by LINUS using CN2 with the use of background knowledge and the significance test. All rules for the diagnosis E (non-specific rheumatic manifestations) were given mark 0. This is due to the fact that the only characteristic of this class is that all patients with reumatic diseases who cannot be diagnosed otherwise are assigned this diagnosis.

Tables 11.3 and 11.4 summarize the medical expert evaluation of the induced rules. Out of 30 rules induced without background knowledge no rules were given mark 3, two rules were given mark 2, fifteen mark 1, ten mark 0 and three rules mark -1 (misleading rules). The average rule mark is 0.53 and the three misleading rules cover 34 examples.

Class = degenerative_spine_diseases **if** **Mark: 2**
Duration_of_present_symptoms > 6.5_months,
Duration_of_rheumatic_diseases < 5.5_years,
Number_of_painful_joints > 16,
grouping2(Spinal_pain, Duration_of_morning_stiffness, Value),
Value = 'spondylotic & dms =< 1_hour'.

Class = degenerative_spine_diseases **if** **Mark: 2**
Age < 66.5,
Other_pain = no,
Skin_manifestations = no,
grouping5(Joint_pain, Spinal_pain, Number_of_painful_joints, Value),
Value = 'no_pain & spondylotic & pj = 0'.

Class = degenerative_joint_diseases **if** **Mark: 3**
Age > 46.5,
Duration_of_present_symptoms < 30_months,
Number_of_painful_joints < 19,
Duration_of_morning_stiffness < 0.75_hours,
grouping3(Sex, Other_pain, Value1),
Value1 = irrelevant,
grouping4(Joint_pain, Spinal_pain, Value2),
Value2 = 'arthrotic & no_pain'.

Class = inflammatory_spine_diseases **if** **Mark: 2**
Sex = male,
Duration_of_rheumatic_diseases > 9_years,
Other_manifestations = no,
grouping4(Joint_pain, Spinal_pain, Value),
Value = 'no_pain & spondylitic'.

Class = other_inflammatory_diseases **if** **Mark: 3**
Age < 78.5,
Number_of_painful_joints > 18,
Number_of_swollen_joints > 2.5,
grouping1(Joint_pain, Duration_of_morning_stiffness, Value),
Value = 'arthritic & dms > 1_hour'.

Figure 11.1: Rules for early diagnosis of rheumatic diseases induced with the use of background knowledge.

Class = extraarticular_rheumatism **if** **Mark: 0**
Duration_of_rheumatic_diseases < 1.5_year,
Number_of_painful_joints < 0.5,
Other_pain = other,
Other_manifestations = no.

Class = crystal_induced_synovitis **if** **Mark: 2**
Sex = male,
Age > 24,
Number_of_painful_joints < 14.5,
Other_pain = no,
Duration_of_morning_stiffness < 0.25_hours,
Eye_manifestations = no,
grouping4(Joint_pain, Spinal_pain, Value),
Value = 'arthritic & no_pain'.

Class = crystal_induced_synovitis **if** **Mark: 1**
Sex = male,
Age > 46.5,
Number_of_painful_joints > 3.5,
Skin_manifestations = psoriasis.

Class = nonspecific_rheumatic_diseases **if** **Mark: 0**
Age > 29.5, Age < 33.5,
Number_of_painful_joints > 10.5,
Spinal_pain = no_pain.

Class = nonrheumatic_diseases **if** **Mark: 3**
Age < 53.5,
Duration_of_present_symptoms > 2_months,
Number_of_swollen_joints < 0.5,
Other_pain = no,
Eye_manifestations = no,
grouping5(Joint_pain, Spinal_pain, Number_of_painful_joints, Value),
Value = 'no_pain & no_pain & pj = 0'.

Figure 11.1: (continued)

Class	Number of rules with mark					Total no. of rules	Average mark
	3	2	1	0	-1		
A1		3	2	2		7	1.14
A2	1	1	3	1	1	7	1.00
B1		1	2			3	1.33
B2	1	2	4			7	1.57
C				3		3	0.00
D		1	2			3	1.33
E				4		4	0.00
F	1	1	1	1		4	1.50
Overall	3	9	14	11	1	38	1.05

Table 11.4: Medical expert evaluation of rules induced with background knowledge.

On the other hand, out of 38 rules induced with background knowledge, three rules were given mark 3, nine mark 2, fourteen mark 1, eleven mark 0 and only one rule was misleading. The average rule mark is in this case 1.05 and the misleading rule covers five examples only. All these figures indicate that the background knowledge substantially improves the induced rules from the medical expert point of view.

11.3.3 Performance on unseen examples

Since the ultimate test of the quality of induced rules is their performance on unseen examples, the second group of experiments was performed on ten different partitions of the data set into training and testing examples. Thus, corresponding to each of the experiments on the entire training set, four series of ten experiments each were performed, where 70% of the entire data set were used for learning and 30% for testing. A different partition into training and testing examples was used in each of the experiments and the same partitions were used for all four series.

The experiments were aimed at investigating the effects of background knowledge and noise-handling mechanisms on the performance

Background knowledge	Significance test	Accuracy	Relative inf. score	Number of rules	Number of literals
no	no	42.9%	23%	72	222
no	yes	45.3%	19%	30	76
yes	no	43.9%	24%	74	226
yes	yes	48.6%	26%	24	88

Table 11.5: Average classification accuracy, relative information score (measured on testing data) and complexity of rules induced by LINUS using CN2.

of induced rules on unseen cases. The performance of the rules was measured in terms of the classification accuracy and the relative information score. A summary of the results of the experiments is given in Table 11.5. The classification accuracy and the relative information score for all experiments are given in Table 11.6 and Table 11.7, respectively.

As expected, the average classification accuracy is much lower than the classification accuracy on the training data in Table 11.2. The relative information score also drops substantially. Finally, the rule set size is also reduced, as the rules have to explain a smaller number of examples (70% of the entire data set).

The use of background knowledge improved the classification accuracy and the relative information scores achieved. According to the statistical *t*-test for dependent samples, the difference in accuracy is not statistically significant when the CN2 significance test is not used. However, it is very significant (at the *t*-test 99% level) when the significance test is used in CN2. The relative information score difference is statistically significant in both cases (at the *t*-test 95% and 99.99% levels, respectively, with and without the significance test in CN2).

The significance test noise-handling mechanism in CN2 greatly reduces the complexity of the induced rules. It also improves the classification accuracy significantly. When background knowledge is used, it significantly improves both the classification accuracy (*t*-test significance level 99.99%) and the relative information score (*t*-test significance level 95%).

Partition	No significance test		Significance level 99%	
	Without BK	With BK	Without BK	With BK
1	38.1%	45.3%	47.5%	48.9%
2	44.6%	44.6%	45.3%	51.8%
3	45.3%	42.4%	51.1%	48.9%
4	43.9%	40.3%	44.6%	48.2%
5	40.3%	43.2%	46.0%	46.8%
6	48.2%	48.2%	49.6%	48.2%
7	42.4%	44.6%	44.6%	48.9%
8	38.8%	43.2%	41.0%	48.2%
9	45.3%	41.0%	43.9%	48.2%
10	41.7%	46.0%	39.6%	48.2%
Average	42.9%	43.9%	45.3%	48.6%

Table 11.6: Classification accuracy of rules, derived by CN2 with and without background knowledge (BK), on the testing set for each of the ten partitions of the 462 examples.

All groupings appear in the rule sets induced with using background knowledge. In the rules induced with no CN2 significance test, the most common groupings are *grouping*4 with 96 and *grouping*5 with 80 occurences in the 10 rule sets; *grouping*6 occurs 38 times, *grouping*2 occurs 31 times, *grouping*1 occurs 23 times and *grouping*3 occurs 12 times. When the CN2 significance test is used, the number of occurences decreases, as the number of rules (literals) decreases drastically. In this case, *grouping*1 occurs ten times, *grouping*2 seven times, *grouping*3 five times, *grouping*4 occurs 73 times, *grouping*5 occurs 71 time and *grouping*6 occurs 24 times.

Experiments were also conducted with LINUS using ASSISTANT on the same partitions of the data set as used by LINUS using CN2. ASSISTANT induces decision trees, where noise is handled either by tree pre-pruning or post-pruning [Cestnik et al. 1987]. A summary of the results of these experiments is given in Table 11.8.

The background knowledge did not significantly influence the performance of the trees induced by LINUS using ASSISTANT. This is

Partition	No significance test		Significance level 99%	
	Without BK	With BK	Without BK	With BK
1	21%	23%	17%	26%
2	23%	24%	20%	28%
3	19%	22%	17%	24%
4	24%	21%	17%	22%
5	22%	25%	21%	26%
6	26%	24%	15%	24%
7	27%	30%	21%	27%
8	19%	24%	21%	26%
9	23%	23%	16%	25%
10	23%	27%	23%	31%
Average	23%	24%	19%	26%

Table 11.7: Relative information scores of rules, derived by CN2 with and without background knowledge (BK), on the testing set for each of the ten partitions of the 462 examples.

probably due to the fact that the background knowledge is in a form more suitable for rule induction, i.e., the groupings typically distinguish between one diagnosis and all the others. In decision trees, on the other hand, features that distinguish best among all possible diagnoses are favoured.

The noise-handing mechanisms of pre-pruning and post-pruning in ASSISTANT increased the accuracy of the induced trees. The increases are statistically significant (at the 98% level according to the t-test for dependent samples) in the case of pre-pruning and post-pruning, when no background knowledge is used, and in the case of post-pruning when background knowledge is used. It is interesting to note that pre-pruning increased both the classification accuracy and the relative information score, while post-pruning increased the accuracy at the expense of decreasing the relative information score.

Finally, let us mention that the accuracies and relative information scores achieved by LINUS using CN2 with background knowledge and with LINUS using ASSISTANT (both with and without background

Background knowledge	Pruning	Accuracy	Relative information score
no	no	41.80%	28.68%
no	pre	44.42%	31.03%
no	post	48.92%	25.31%
yes	no	42.34%	28.90%
yes	pre	43.17%	30.32%
yes	post	48.99%	25.94%

Table 11.8: Average classification accuracy and relative information score (measured on testing data) of trees generated by LINUS using ASSISTANT.

knowledge) are almost the same. However, the rules induced by CN2 proved to be easier to understand by the medical expert than the decision trees induced by ASSISTANT. In that context, the background knowledge substantially improved the quality of the rules induced by CN2 as evaluated by the medical expert (see Tables 11.3 and 11.4).

11.4 Discussion

LINUS, an inductive learning system, was used to learn diagnostic rules from anamnestic data of patients with rheumatic diseases. In addition to the available patient data, described by values of a fixed set of attributes, LINUS was given domain specific (background) knowledge, which specified some of the characteristic co-occurences of symptoms.

Medical evaluation of rules induced by LINUS using CN2 shows that the use of background knowledge substantially improves the quality of the induced rules from a medical point of view. However, even when background knowledge is used, the average rule is not wrong, but not too characteristic (average mark 1.05). The analysis by a specialist for rheumatic diseases indicated several reasons:

- Anamnestic data are by nature very noisy since they are, in fact, patients' own description of the disease, only interpreted by a specialist for rheumatic diseases. Interpretation of this data is subjective and therefore extremely unreliable. As such, in many cases the data even contradicts the expert's background knowledge.

- The grouping of about 200 different diagnoses into only eight diagnostic classes is problematic. For degenerative diseases (classes $A1$ and $A2$) many examples are available. Nearly 74% of all the data set consists of patient records for these two diagnostis classes, together with the class of non-rheumatic diseases (see Table 11.1). Furthermore, some diagnostic classes are relatively non-homogenous, having little common characteristics. Consequently, the specific background knowledge, which is usually used in differential diagnosis, can not be used effectively when dealing with such large groups of diagnoses.
- Some of the patients had more than one diagnosis, but only one diagnosis was included in the example set.
- Data were collected by different medical doctors without achieving their collective consensus. This could be the reason that sometimes the data contradict the expert's background knowledge.

This analysis shows the problems present in the data set. Nevertheless, useful information can still be extracted even from such data. The relative information score of classifications of training examples (without using the significance test) is 49% (accuracy 62.8%) when no background knowledge was used, and 59% (accuracy 72.9%) when background knowledge was used (Table 11.2). The relative information score of a classifier that always returns the prior probability distribution of diagnostic classes is zero (accuracy 34%). The information score of the induced rules thus indicates that useful information is contained in the data set.

The main goal of the study was to analyze how the use of background knowledge and the noise-handling mechanism of LINUS using CN2 affect the performance and the complexity of induced diagnostic rules. The use of noise-handling mechanisms improved the classification accuracy of induced rules. We have also shown that the use of additional background knowledge can improve the classification accuracy and the relative information score of induced rules. We expect that still better results could be achieved by restricting the problem to differential diagnosis in a smaller subset of diagnoses where specific (background) knowledge could have a more substantial role. Furthermore, the selection of typical patients only is also expected to contribute to better results.

12

Finite element mesh design

The finite element (FE) method is frequently used to analyze stresses in physical structures. These structures are represented quantitatively as finite collections (meshes) of elements. Dolšak applied the ILP system GOLEM, described in Section 4.2, to construct rules deciding on appropriate mesh resolution [Dolšak 1991, Dolšak and Muggleton 1992, Dolšak et al. 1992]. In this chapter we describe the application of FOIL and mFOIL to the same problem [Džeroski and Bratko 1992a, Džeroski and Dolšak 1992]. We first define the learning problem more precisely, then proceed with a description of the experimental setup and finally present and discuss the results.

12.1 The finite element method

The finite element (FE) method is used extensively by engineers and scientists to analyze stresses in physical structures. These structures are represented quantitatively as finite collections (meshes) of elements. The deformation of each element is computed using linear algebraic equations. Figure 12.1 shows a typical instance of such a structure, with a corresponding finite element mesh. The structure shown is a pipe connector [Dolšak 1991].

In order to design a numerical model of a physical structure it is necessary to determine the appropriate resolution for modeling each component part. The basic demand for a FE mesh is that it should represent the shape of a structure accurately. In addition, a fine mesh should be used in places where high deformations are expected, in order to minimize the errors in the calculated deformation values. On the other hand, a coarser mesh is appropriate where small deformations are expected. Too fine a mesh leads to unnecessary computational

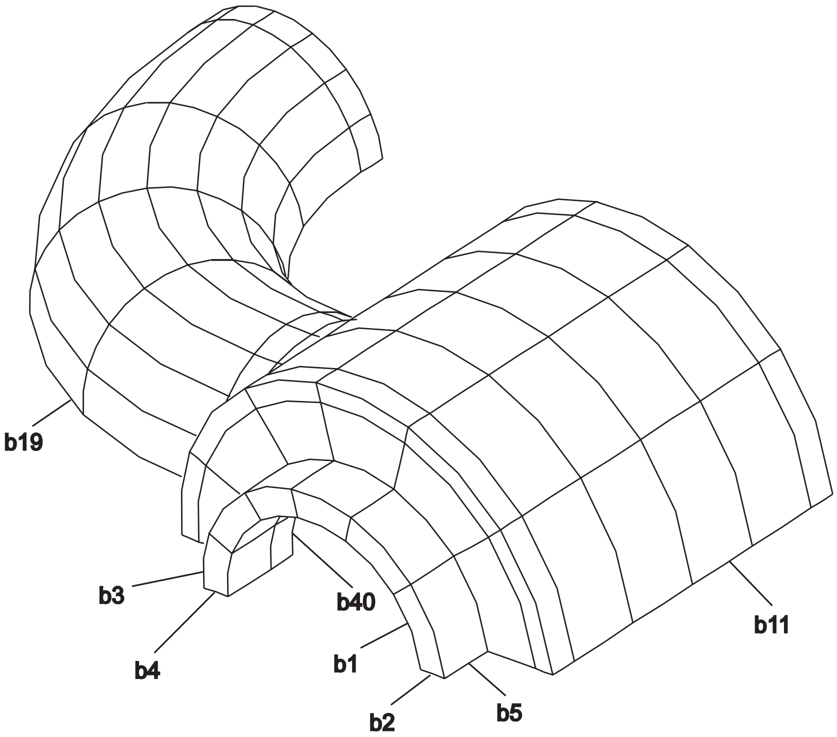


Figure 12.1: A typical structure with its corresponding FE mesh. From Dolšak [1991].

overheads when executing the model, while too coarse a mesh produces intolerable approximation errors. In general, the coarsest mesh which gives rise to sufficiently low errors should be employed.

Considerable expertise is required in choosing appropriate resolution values, since they are affected by several factors, including the shape of the structure, the stresses applied to it and the boundary conditions. As the FE method has been used extensively for the last thirty years, various FE mesh designs are described in numerous published reports. The descriptions typically include the shape of the objects analyzed, the stresses applied to them, the final FE mesh (chosen after several trials) and the results of the analysis. By applying appropriate machine learning techniques, the data from these reports can be used to construct a knowledge base for a FE mesh design expert system.

The first step towards using machine learning techniques is the choice of an appropriate representation for the problem at hand. In the case of FE mesh design this includes the representation of the structure analyzed, the stresses applied to it, the boundary conditions and the final design of the mesh. An attribute-value representation, used in most of the commercially available machine learning systems, is essentially inappropriate for representing this problem. Namely, a reasonable representation of the geometry of a structure must include the relations between its primitive components, which can not be represented naturally in an attribute-value language. Instead, ILP techniques should be applied.

12.2 The learning problem

The resolution of a FE mesh is determined by the number of elements on each of its edges. The problem of learning rules for determining the resolution of a FE mesh can, thus, be formulated as a problem of learning rules to determine the number of elements on an edge. The training examples have the form $mesh(E, N)$, where E is an edge label (unique for each edge) and N is the number of elements on the edge denoted by label E. In other words, the target relation *mesh* has arguments of type *edge* (which consists of all edge labels) and *elements*, where $elements = \{1, \ldots, 17\}$.

In preliminary experiments [Dolšak and Muggleton 1992], three objects (structures) with their corresponding meshes were used in the learning process: a hook, a hydraulic press cilinder and a paper mill. The preliminary results obtained by GOLEM were then compared to the performance of FOIL and LINUS, which were applied to the same problem [Dolšak and Muggleton 1992]. Dolšak [1991] then gathered data about three additional structures: a pipe connector, a roller and a bearing box.

Five different structures (A – a hydraulic press cylinder, B – a pipe connector, C – a paper mill, D – a roller and E – a bearing box), described in Dolšak [1991], were used for learning in our experiments. The positive training examples were derived from these structures. To each edge in a structure corresponds a positive training example stat-

	Number of elements													
Structure	1	2	3	4	5	6	7	8	9	10	11	12	17	$\sum$
A	21	9	3	2	3			1			2	13	1	55
B	9	9		1		11	4	4						42
C	6	6	2					14						28
D	14	13		2								28		57
E	23	36	11	5	9	4			6	1		1		96

Table 12.1: Distribution of the number of edges with respect to the number of elements on them.

ing the number of elements on it. For instance, the positive example $mesh(a15, 4)$ states that four is an appropriate number of elements for the edge $a15$.

The negative training examples were generated under a modified (smoothed) closed world assumption. They included most facts of the form $mesh(E, N)$, which were not among the positive examples. However, relatively small deviations (differences in the number of elements) were not considered as counter examples. For example, given the positive example $mesh(c15, 8)$, the facts $mesh(c15, 7), mesh(c15, 9)$ and $mesh(c15, 10)$ are not among the negative examples. A detailed description of the training data is given in [Dolšak 1991]. There were alltogether 278 positive examples and 2840 negative examples. The distributions of the number of edges according to the number of elements is given in Table 12.1.

The background knowledge describes some of the factors that influence the resolution of a FE mesh, such as the type of edges, boundary conditions and loadings, as well as the shape of the structure (relations of neighborhood and oppositeness). According to its importance and geometric shape, an edge can belong to one of the following types: $important\_long$, $important$, $important\_short$, $not\_important$, $circuit$, $half\_circuit$, $quarter\_circuit$, $short\_for\_hole$, $long\_for\_hole$, $circuit\_hole$, $half\_circuit\_hole$ and $quarter\_circuit\_hole$. With respect to the boundary conditions, an edge can be $free$, one_side_fixed, two_side_fixed or $fixed$. Finally, according to loadings an edge is not_loaded, one_side_loaded, two_side_loaded or $continuously\_loaded$.

Training examples	
Positive examples	*Negative examples*
$mesh(b1, 6).$	$mesh(b1, 1).$
$mesh(b2, 1).$	$mesh(b2, 2).$
$mesh(b3, 6).$	$mesh(b3, 2).$
$mesh(b4, 1).$	$mesh(b4, 3).$
$mesh(b10, 2).$	$mesh(b10, 8).$
$mesh(b11, 6).$	$mesh(b11, 12).$

Table 12.2: Training examples for the FE mesh design problem.

Background knowledge about the shape of a structure includes the symmetric relations *neighbor* and *opposite*, as well as the relation *equal*. The latter states that two edges are not only opposite, but are also of the same shape, such as concentric circles. Unlike the unary relations that express the properties of the edges, the relations *neighbor*, *opposite* and *equal* are binary. The arguments of all background predicates are of type *edge*. For each predicate, the first argument is an input argument and the second is an ouput one. Excerpts from the training examples and the background knowledge describing the labeled edges from Figure 12.1 are given in Tables 12.2 and 12.3, respectively.

12.3 Experimental setup

Learning and evaluation was done by the cross-validation leave-one-out strategy. For each of the five structures, a set of clauses was derived from the background knowledge and examples for the remaining four structures and then tested on the structure (leave-one-out). In the classification process, the induced rules have to assign the correct number of elements to each of the edges. More precisely, to determine the number of elements on edge E, the goal $mesh(E, N)$ is called, where E is a bound variable and N is not. If N is assigned the correct number, the rules score one. If it is assigned an incorrect number, or is not assigned a value at all (is uncovered), the rules score zero.

As the recursive clauses that were built in preliminary experiments

Background knowledge		
Edge type	*Boundary condition*	*Loading*
$long(b19)$.	$fixed(b1)$.	$not\_loaded(b1)$.
$short(b10)$.	$fixed(b2)$.	$not\_loaded(b2)$.
$not\_important(b2)$.	$fixed(b3)$.	$not\_loaded(b3)$.
$short\_for\_hole(b28)$.	$fixed(b4)$.	$not\_loaded(b4)$.
$half\_circuit(b3)$.	$two\_side\_fixed(b6)$.	$cont\_loaded(b22)$.
$half\_circuit\_hole(b1)$.	$two\_side\_fixed(b9)$.	$cont\_loaded(b25)$.
Geometry of the structure		
$neighbor(b1, b2)$.	$opposite(b1, b3)$.	$same(b1, b3)$.
$neighbor(b2, b3)$.	$opposite(b10, b33)$.	$same(b3, b6)$.
$neighbor(b6, b10)$.	$opposite(b11, b32)$.	$same(b9, b12)$.
$neighbor(b10, b11)$.	$opposite(b19, b39)$.	$same(b41, b42)$.

Table 12.3: Background knowledge for the FE mesh design problem.

caused infinite loops, no recursive clauses were allowed in mFOIL and GOLEM, and recursive clauses built by FOIL were discarded afterwards. For the classification process, the rules were ordered according to the Laplace estimate of their expected classification accuracy (described in Section 9.2). Clauses with accuracy less than 80% were discarded for comparison with FOIL, where this is done automatically.

mFOIL used the background knowledge as described above. In addition, literals of the form $X = value$, where $value$ is a constant, were allowed for variables of type *element*. The Laplace estimate was used as the search heuristic. Clauses were constructed until no more significant clauses could be found. The default beam size (five) and significance threshold (99%) were employed.

As FOIL does not have literals of the type $X = value$, relations of the form $is\_a\_value(X)$ were added to the background knowledge for each constant of type *element*. The default parameters were used in FOIL2.1. A few of the induced rules were recursive (as no partial orders exist in this domain, the recursive literals came into the clauses through the determinate literal facility of FOIL2.1). However, they were discarded as they caused infinite loops.

In the experiments with GOLEM, some pre-processing of the background relations was necessary. Due to the restrictions of introducing new variables and mode declarations in GOLEM, it was necessary to split each of the relations *neighbor*, *opposite* and *equal* to several subrelations. The transformation is described in detail in Dolšak [1991] and Dolšak and Muggleton [1992]. The noise parameter (number of negative examples that may be covered by a clause) was set to five. The *rlggsample* parameter was set to 50, i.e., the search for each of the clauses starts by randomly picking 50 pairs of positive examples and constructing their *rlgg*s (see Section 4.2). These parameter values were suggested by Feng, one of the developers of GOLEM.

The run time of FOIL on each training set amounted to five minutes on a SUN SPARC 1. mFOIL took about two hours for the same task. GOLEM, with settings as described above, took a little more than one hour. It should be noted that FOIL and GOLEM are implemented in C and mFOIL in Prolog.

12.4 Results and discussion

Some rules for FE mesh design, induced by mFOIL, are given in Figure 12.2. Similar clauses were induced by FOIL and GOLEM.

Both FOIL and mFOIL had problems with the relation *neighbor*. This predicate does not discriminate between positive and negative examples. In addition, it is never determinate, since each edge has usually two neighbors. Therefore, the gain (information gain in FOIL or accuracy increase in mFOIL) is small or zero. Consequently, this predicate is almost never used in the induced clauses. An additional problem in FOIL, which is the cause of its relatively poor performance,

$$
\begin{array}{rl}
mesh(A,B) \leftarrow & B = 1, not\_important(A). \\
mesh(A,B) \leftarrow & quarter\_circuit(A), B = 9. \\
mesh(A,B) \leftarrow & B = 2, short(A), \\
 & opposite(A,C), not\_important(C). \\
mesh(A,B) \leftarrow & two\_side\_fixed(A), B = 6, \\
 & opposite(A,C), cont\_loaded(C), half\_circuit(C).
\end{array}
$$

Figure 12.2: Rules for FE mesh design induced by mFOIL.

Structure	*FOIL*	*mFOIL*	*GOLEM*
A	17	22	17
B	5	12	9
C	7	9	5
D	0	6	11
E	5	10	10
$\sum$	34	59	52
%	12	21	19

Table 12.4: The number and percentage of examples correctly classified by FOIL, mFOIL and GOLEM in the FE mesh design domain.

is that the large number of background relations increases the number of bits needed to encode a clause. Thus, less clauses are allowed to be built.

The number of testing examples correctly classified by FOIL, mFOIL and GOLEM is given in Table 12.4. Both mFOIL and GOLEM performed better than FOIL. However, the achieved classification accuracies are not very high. This may be due to at least two reasons. First, a close inspection of Table 12.1 reveals that each of the objects has some unique characteristics. These can not be captured when learning from the other structures. Second, the neighbor relation is not used in the induced clauses, which means that essential information is not taken into account.

To determine the possible gain of the proper use of the *neighbor* relation, we conducted a simple experiment with mFOIL. The starting clause $mesh(E, N) \leftarrow$ which is refined by specialization, was replaced by the clause $mesh(E, N) \leftarrow neighbor(E, F)$. This enabled mFOIL to use the properties of the neighboring edges in the induced clauses. While the number of correctly classified edges for objects A to D remained approximately the same, the number of correctly classified edges of object E increased from 10 to 37. This suggests that a significant improvement is possible if the information about neighboring edges is taken into account properly.

To improve the above results, a larger set of structures should be used for learning, which would provide for some redundancy. Information about the neighbors of an edge should also be taken into account in an appropriate manner. To encourage the use of the *neighbor* relation, a lookahead is needed, which would allow for non-determinate literals with zero or small gain. Such lookahead is already implemented in FOIL for determinate literals with zero or small gain. In a similar way, non-determinate literals with non-negative gain might be added to a clause. In the mesh design domain, the cost imposed by this extension should not be prohibitive.

13

Learning qualitative models of dynamic systems

Qualitative models can be used instead of traditional numerical models in a wide range of tasks, including diagnosis, generating explanations of the system's behavior and designing novel devices from first principles. It is conjectured that qualitative models are sufficient for the synthesis of control rules for dynamic systems. It is, therefore, important to be able to generate qualitative models automatically from example behaviors of dynamic systems.

As a qualitative model consists of relations among the system variables/components of the modeled system, relation learning systems (ILP systems) can be used to induce qualitative models from example behaviors. Bratko et al. [1992] describe the application of the ILP system GOLEM to the problem of learning a qualitative model of the dynamic system of connected containers, usually referred to as the U-tube system. There have been, however, several problems with the application of GOLEM to this task, stemming from the inability of GOLEM to use non-ground and non-determinate background knowledge.

In this chapter, we describe the application of the inductive logic programming system mFOIL [Džeroski and Bratko 1992a] to the same task. We first give a brief introduction to the QSIM (Qualitative SIMulation) [Kuipers 1986] formalism and illustrate its use on the connected-containers (U-tube) system. The experiments and results of learning a qualitative model of the U-tube system with mFOIL are next presented, followed by a discussion of related work. Finally, we conclude with some directions for further work.

13.1 Qualitative modeling

Qualitative models can be used instead of traditional numerical models in a wide range of tasks [Bratko 1991]. These tasks include diagnosis (e.g., Bratko et al. [1989]), generating explanations of the system's behavior (e.g., Falkenheiner and Forbus [1990]) and designing novel devices from first principles (e.g., Williams [1990]). Bratko [1991] conjectures that qualitative models are sufficient for the synthesis of control rules for dynamic systems, and supports this conjecture with an example.

Among several established formalisms for defining qualitative models of dynamic systems, the most widely known are qualitative differential equations called confluences [De Kleer and Brown 1984], Qualitative Process Theory [Forbus 1984] and QSIM [Kuipers 1986]. We will adopt the QSIM (Qualitative SIMulation) formalism, as it has been already used for learning qualitative models.

In this section, we first introduce the QSIM [Kuipers 1986] formalism and then illustrate it on the U-tube system. We also describe how the QSIM theory can be formulated in logic [Bratko et al. 1992], so that it can be used as background knowledge in the process of learning qualitative models from examples.

13.1.1 The QSIM formalism

In the theory of dynamic systems, a physical system is represented by a set of continuous variables, which may change over time. Sets of differential equations, relating the system variables, are typically used to model dynamic systems numerically. Given a model (a set of differential equations) of the system and its initial state, the behavior of the system can be predicted by applying a numerical solver to the set of differential equations.

A similar approach is taken in qualitative simulation [Kuipers 1986]. In QSIM, a physical system is described by a set of variables representing the *physical parameters* of the system (continuously differentiable real-valued functions) and a set of *constraint equations* describing how those parameters are related to each other. In this case, a (qualitative) model is a set of constraint equations. Given a qualitative model and a qualitative initial state of the system, the QSIM simulation algorithm

[Kuipers 1986] produces a directed graph consisting of possible future states and the immediate successor relation between the states. Paths in this graph starting from the initial state correspond to behaviors of the system.

The value of a physical parameter is specified qualitatively in terms of its relationship with a totally ordered set of *landmark values*. The *qualitative state* of a parameter consists of its value and direction of change. The direction of change can be *inc* (increasing), *std* (steady) and *dec* (decreasing). Time is represented as a totally ordered set of symbolic distinguished time points. The current time is either at or between distinguished time-points. At a distinguished time-point, if several physical parameters linked by a single constraint are equal to landmark values, they are said to have *corresponding values*.

The constraints used in QSIM are designed to permit a large class of differential equations to be mapped straightforwardly into qualitative constraint equations. They include mathematical relationships, such as $deriv(Velocity, Acceleration)$ and $mult(Mass, Acceleration, Force)$. Constraints like $M^+(Price, Power)$ and $M^-(Current, Resistance)$, on the other hand, state that there is a monotonically increasing/decreasing functional relationship between two physical parameters, but do not specify the relationship completely.

13.1.2 The U-tube system

Let us illustrate the above notions on the connected-containers (U-tube) example, adapted from Bratko et al. [1992]. The U-tube system, illustrated in Figure 13.1, consists of two containers, A and B, connected with a pipe and filled with water to the corresponding levels La and Lb. Let the flow from A to B be denoted by Fab, the flow from B to A by Fba. The variables La, Lb, Fab and Fba are the parameters of the system.

The flows Fab and Fba are the time derivatives of the water levels Lb and La, respectively, and run in opposite directions. Let the difference in the levels of the containers A and B be $Diff = La - Lb$. The pressure $Press$ along the pipe influences the flow Fab: the higher the pressure, the greater the flow. A similar dependence exists between the level difference and the pressure. The above constraints can be formulated in QSIM as follows:

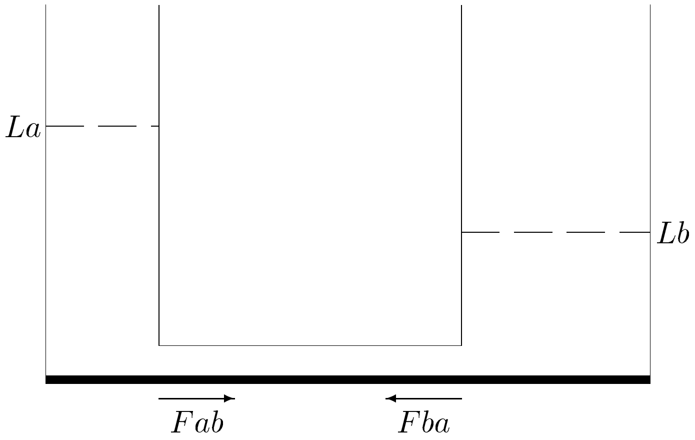


Figure 13.1: The U-tube system.

$$\frac{d}{dt}La = Fba$$

$$\frac{d}{dt}Lb = Fab$$

$$Fab = -Fba$$

$$Diff = La - Lb$$

$$Press = M^{+}(Diff)$$

$$Fab = M^{+}(Press)$$

If we are not explicitly interested in the pressure, the last two qualitative constraint equations can be simplified into one:

$$Fab = M^{+}(Diff)$$

For comparison, in a numerical model, the last two equations might have the form

$$Press = c_1 \cdot Diff$$

$$Fab = c_2 \cdot Press$$

$Time$	La	Lb	Fab	Fba
$t0$	$la0/dec$	$lb0/inc$	$fab0/dec$	$fba0/inc$
$(t0, t1)$	$0..la0/dec$	$lb0..inf/inc$	$0..fab0/dec$	$fba0..0/inc$
$t1$	$0..la0/std$	$lb0..inf/std$	$0/std$	$0/std$
$(t1, inf)$	$0..la0/std$	$lb0..inf/std$	$0/std$	$0/std$

Table 13.1: Qualitative behavior of the U-tube system.

or, when simplified

$$Fab = c \cdot Diff$$

where c, c_1 and c_2 are positive constants. In this case, the relationship between the variables Fab, $Press$ and $Diff$ is completely, and not only qualitatively, specified given the values of c, c_1 and c_2.

The landmark values for the variables of this model for the U-tube, ordered left to right, are as follows:

$$La : minf, 0, la0, inf$$
$$Lb : minf, 0, lb0, inf$$
$$Fab : minf, 0, fab0, inf$$
$$Fba : minf, fba0, 0, inf$$

These values are symbolic names corresponding to minus infinity, zero, infinity and the initial values of the four variables. The left-to-right ordering corresponds to the *less than* relation between the corresponding numerical values.

The QSIM simulation of the U-tube system produces the trace given in Table 13.1. From the trace we can see, for example, that in the initial state the value of the level La is equal to $la0$ and is decreasing (dec). This is represented as $La = la0/dec$. In the time interval that follows, La is between 0 and $la0$ and decreasing, which is written as $La = 0..la0/dec$.

13.1.3 Formulating QSIM in logic

Bratko et al. [1992] translate the QSIM approach to qualitative simulation into a logic programming formalism (pure Prolog). A sketch of the QSIM qualitative simulation algorithm, written as a logic program, is given below.

```
simulate(State) ←
   transition(State, NextState),
   simulate(NextState).

transition(state(V1, ...), state(NewV1, ...)) ←
   trans(V1, NewV1),                      %Model − independent
   ...,
   legalstate(NewV1, ...).                %Model − dependent
```

The simulation starts from the initial qualitative *State*, consisting of the qualitative values and directions of change of the system parameters. The simulator first finds a possible transition to a *NewState* and then continues the simulation from the new state. The relation *trans* is a non-deterministic relation that generates possible transitions of the system parameters, i.e., possible new values for them. It is defined as part of the QSIM theory [Kuipers 1986]. The model of a particular system is defined by the predicate *legalstate* which imposes constraints on the values of the system parameters. The definition of this predicate has the following form:

```
legalstate(...) ←
   constraint1(...),
   constraint2(...),
   ...
```

where the constraints are part of the QSIM theory. Under continuity assumptions, the problem of learning the legality of states is equivalent to the problem of learning the dynamics of the system.

The following Prolog predicates correspond to the QSIM constraints:

```
add(F1, F2, F3, Corr)      % F1 + F2 = F3
mult(F1, F2, F3, Corr)     % F1 * F2 = F3
minus(F1, F2, Corr)        % F1 = −F2
m_plus(F1, F2, Corr)       % F2 = M+(F1)
m_minus(F1, F2, Corr)      % F2 = M−(F1)
deriv(F1, F2)              % F2 = dF1/dt
```

In the above $F1, F2$ and $F3$ stand for system parameters and $Corr$ stands for a list of corresponding values.

The qualitative model for the U-tube system can be written in the logic programming notation as follows:

$legalstate(La, Lb, Fab, Fba) \leftarrow$
$\quad add(Lb, Diff, La, [c(lb0, d0, la0)]),$
$\quad m\_plus(Diff, Fab, [c(0, 0), c(d0, fab0)]),$
$\quad minus(Fab, Fba, [c(fab0, fba0)]),$
$\quad deriv(La, Fba),$
$\quad deriv(Lb, Fab).$

where $c(x, y, z)$ means that x, y and z are corresponding values for the constraint. For example, in the *add* constraint, $c(lb0, d0, la0)$ means that $lb0 + d0 = la0$.

If we are not interested in the $Diff$ parameter, the first two constraints can be replaced by $add(Lb, Fab, La, [c(lb0, fab0, la0)])$, or symmetrically, by the constraint $add(La, Fba, Lb, [c(la0, fba0, lb0)])$.

13.2 An experiment in learning qualitative models

A fundamental problem in the theory of dynamic systems is the identification problem, defined as follows [Bratko 1991]: given examples of the behavior of a dynamic system, find a model that explains these examples. Motivated by the hypothesis that it should be easier to learn qualitative than quantitative models, Bratko et al. [1992] have recently formulated the identification problem for QSIM models as a machine learning problem. Formulated in this framework, the task of learning QSIM-type qualitative models is as follows: given *QSIMtheory* and *ExamplesOfBehavior*, find a *QualitativeModel*, such that *QSIMtheory* and *QualitativeModel* explain the *ExamplesOfBehavior*, or formally,

$$QSIMtheory \cup QualitativeModel \models ExamplesOfBehavior$$

The identification task can be formulated as a machine learning task. Namely, the task of inductive machine learning is to find a hypothesis that explains a set of given examples. In some cases the learner can also make use of existing background knowledge about the given examples and the domain at hand. So, the learning task can be formulated as follows: given background knowledge $\mathcal{B}$ and examples $\mathcal{E}$, find a hypothesis $\mathcal{H}$, such that $\mathcal{B}$ and $\mathcal{H}$ explain $\mathcal{E}$, i.e., $\mathcal{B} \cup \mathcal{H} \models \mathcal{E}$. We can see that *ExamplesOfBehavior* correspond to $\mathcal{E}$, *QSIMtheory* corresponds to $\mathcal{B}$ and the target *QualitativeModel* to $\mathcal{H}$.

As a qualitative model consists of relations among the parameters of the modeled system, we have to use an inductive system for learning relations, i.e., an inductive logic programming system. Bratko et al. [1992] describe the application of the ILP system GOLEM to the problem of learning a qualitative model of the dynamic system of connected containers, usually referred to as the U-tube system. There have been, however, several problems with the application of GOLEM to this task, stemming from the inability of GOLEM to use non-ground and non-determinate background knowledge.

In this section, we describe the application of the ILP system mFOIL to the same task of learning a qualitative model of the U-tube system [Džeroski and Bratko 1992a]. We first describe in detail the experimental setup and then present the results generated by mFOIL, followed by a comparison with the results obtained by GOLEM and FOIL on the same problem.

13.2.1 Experimental setup

As mentioned earlier, when formulating the task of identification of models as a machine learning task, *ExamplesOfBehavior* become training examples and *QSIMtheory* constitutes the background knowledge. The *QualitativeModel* to be learned corresponds to the hypothesis to be induced by a machine learning system. In the following we describe in more detail the training examples and background knowledge used in our experiment. We also give the parameter settings for the inductive logic programming system mFOIL used in the experiment.

As the model of a system is defined by the predicate *legalstate*, the learning task is to induce a definition of this predicate. The behavior trace of the U-tube system (Table 13.1) provides three positive training examples for the predicate *legalstate* (the last two states of the trace are equal). In addition, a positive example which corresponds to the case where there is no water in the containers is considered. The set of

```
legalstate(la:la0/dec, lb:lb0/inc, fab:fab0/dec, fba:fba0/inc).
legalstate(la:0..la0/dec, lb:lb0..inf/inc, fab:0..fab0/dec, fba:fba0..0/inc).
legalstate(la:0..la0/std, lb:lb0..inf/std, fab:0/std, fba:0/std).
legalstate(la:0/std, lb:0/std, fab:0/std, fba:0/std).
```

Table 13.2: Positive examples for learning a qualitative model of the U-tube.

positive examples considered is given in Table 13.2.

Negative examples (illegal states) represent 'impossible' states which cannot appear in any behavior of the system. For instance, the state $(la : la0/inc, lb : lb0/inc, fab : f0/dec, fba : mf0/inc)$ is a negative example, because it cannot happen that the water levels in both containers increase. Negative examples can be either hand-generated by an expert, or can be generated under the closed-world assumption, when all positive examples are known.

Bratko et al. [1992] used six hand-crafted negative examples, called near misses, which differ only slightly from the positive ones. In a preliminary experiment, from the four positive examples in Table 13.2 and the six near misses, mFOIL generated an overly general, i.e., underconstrained model. We thus used a larger set of 543 negative examples, randomly chosen by Žitnik [1991] from the complete set of negative examples generated under the closed-world assumption.

The QSIM theory, formulated in logic, serves as background knowledge. To reduce the complexity of the learning problem, the corresponding values argument $Corr$ is omitted from the constraints. The background knowledge thus consists of the predicates $add(F1, F2, F3)$, $mult(F1, F2, F3)$, $minus(F1, F2)$, $m\_plus(F1, F2)$, $m\_minus(F1, F2)$ and $deriv(F1, F2)$, which correspond to the QSIM constraint primitives with empty lists of corresponding values. For comparison with GOLEM [Bratko et al. 1992], the *mult* relation was excluded from the background knowledge. All arguments of the background predicates are of the same type; they are compound terms of the form $FuncName : QualValue/DirOfChange$.

As mFOIL allows for the use of non-ground background knowledge, we used directly the Prolog definitions of the background predicates and did not tabulate them as ground facts. All of the background predicates, except *deriv*, are symmetric. For example, $add(X, Y, Z)$ is equivalent to $add(Y, X, Z)$. The same holds for the predicate *mult*. Similarly, $minus(X, Y)$ is equivalent to $minus(Y, X)$. This reduces the space of models to be considered.

All arguments of the background knowledge predicates were considered input, i.e., only relations between the given system parameters (La, Lb, Fab, Fba) were considered. This is reasonable, as the correct qualitative model can be formulated in terms of these parameters and without introducing new variables. In some cases, however, the introduction of new variables is necessary. For example, if the U-tube system were described only in terms of La and Lb, the new variables Fab and Fba (or at least one of them) would be necessary for the construction of a qualitative model of the system. As mFOIL allows for the introduction of new variables, such a case could be, in principle, handled if necessary.

Finally, let us mention that the default search heuristic and stopping criteria were used in mFOIL. The Laplace estimate was used as a search heuristic and a significance level of 99% was employed in the significance tests. The default beam width of 5 was increased to 20 in order to avoid getting stuck in local optima. Namely, if the beam width is one, beam search is actually hill-climbing search and is prone to getting stuck in local optima during the search for good models.

13.2.2 Results

Given the 4 positive, the 543 negative examples and the background knowledge as described above, mFOIL generated 20 different models, shown in Table 13.3. They are all evaluated as equally good by mFOIL as they correctly distinguish between the given positive and negative examples. In addition, they are of the same length, i.e., consist of four constraints each. However, not all of them are equivalent to the correct model.

Model # 6 is equivalent to the correct model, shown in Section 13.1.3, provided that corresponding values are ignored in the latter. The same holds for model # 16 [Žitnik, personal communication]. Out of the 194481 possible states, these models cover 130 states (among which 32 are physically possible, i.e., are positive examples). When all 32 positive examples and the same 543 negative examples were given to

#	Model: $legalstate(La, Lb, Fab, Fba) \leftarrow$
1	minus(Fab,Fba), add(Lb,Fab,La), m_minus(La,Fba), deriv(Fab,Fba)
2	minus(Fab,Fba), add(Lb,Fab,La), m_minus(La,Fba), deriv(Lb,Fab)
3	minus(Fab,Fba), add(Lb,Fab,La), m_minus(La,Fba), deriv(La,Fba)
4	minus(Fab,Fba), add(La,Fba,Lb), deriv(La,Fba), m_minus(Lb,Fab)
5	minus(Fab,Fba), add(La,Fba,Lb), deriv(La,Fba), m_plus(Lb,Fba)
6	*minus(Fab,Fba), add(La,Fba,Lb), deriv(Lb,Fab), deriv(La,Fba)*
7	minus(Fab,Fba), add(La,Fba,Lb), deriv(Fab,Fba), deriv(La,Fba)
8	minus(Fab,Fba), add(La,Fba,Lb), deriv(Fba,Fab), deriv(La,Fba)
9	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(Fba,Fab)
10	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(Fab,Fba)
11	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(Lb,Fab)
12	minus(Fab,Fba), add(La,Fba,Lb), m_plus(La,Fab), deriv(La,Fba)
13	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(Fba,Fab)
14	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(Fab,Fba)
15	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(Lb,Fab)
16	*minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Lb), deriv(La,Fba)*
17	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(Fba,Fab)
18	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(Fab,Fba)
19	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(Lb,Fab)
20	minus(Fab,Fba), add(La,Fba,Lb), m_minus(La,Fba), deriv(La,Fba)

Table 13.3: Qualitative models for the U-tube generated by mFOIL.

mFOIL, it was able to generate, among other models, the two models from Table 13.3 which are equivalent to the correct model, even with a beam of size 10 and the *mult* relation in the background knowledge.

An important issue arises from the above results, namely, the need for a criterion of quality for qualitative models other than the standard criteria used in machine learning. Considering the models from Table 13.3, all of which cover all positive and no negative examples, and all of which are of same length, this need is obvious. This problem could be reduced by imposing additional constraints on the models considered in the search process. However, additional semantic criteria may still be needed.

13.2.3 Comparison with other ILP systems

Bratko et al. [1992] applied GOLEM to the problem of learning of qualitative models in the QSIM formalism. They used the four positive examples from Table 13.2 and six hand-crafted negative examples (near misses). The model induced by GOLEM was shown to be dynamically equivalent to the correct model.

However, several modifications had to be done in order to apply GOLEM. As GOLEM accepts only ground facts as background knowledge, the Prolog definitions mentioned above had to be compiled into tables of ground facts. To reduce the complexity of the learning problem, the predicates were tabulated with empty lists of corresponding values (argument $Corr$). The *mult* constraint was not compiled at all. Finally, the *add* constraint had to be decomposed into several subconstraints in order to avoid the explosion of the number of generated ground facts.

Žitnik [1991] conducted further experiments in learning a qualitative model of the U-tube system, using both GOLEM and FOIL. She used several sets of examples, including the set of 4 positive examples and the set of 543 randomly generated negative examples used by mFOIL. Among the conclusions of her work, we would like to mention the following:

- The restriction to extensional background knowledge in FOIL and GOLEM causes an explosion in the complexity of the learning problem, which has to be handled by decomposing predicates into simpler primitives.

- The absence of type information causes problems in interpreting the generalizations produced by FOIL and GOLEM.

- Top-down systems, such as FOIL, need a larger number of negative examples in order to prevent over-generalization.

The following model [Žitnik 1991] was induced by GOLEM from the same examples as used by mFOIL and background knowledge as in Bratko et al. [1992].

$legalstate(la : A/B, lb : C/D, fab : E/B, fba : F/D) \leftarrow$
$\quad deriv\_simplified(D, E),$
$\quad legalstate(la : A/G, lb : C/H, fab : I/G, fba : J/H).$

The condition $deriv\_simplified(D, E)$ means $deriv(Lb, Fab)$. This example illustrates the problem that a model induced by GOLEM can have several interpretations. This is due to the fact that GOLEM may introduce new variables, such as the ones in the term $fba : J/H$ in the model below, the meaning of which may be difficult to grasp.

Similar problems appear when using FOIL [Žitnik 1991]. Some of these problems are absent in LINUS, as it has typed variables, and can use non-ground background knowledge. However, LINUS cannot introduce new variables, which can prevent it from learning an appropriate model if such variables are needed.

None of the above problems appear in mFOIL, as it can use non-ground background knowledge and the typing of variables prevents unclear generalizations, while still having the possibility to introduce new variables. It should be noted, however, that new variables that may be introduced by the background knowledge predicates are likely to be non-discriminating and thus some kind of lookahead would be needed to treat them properly.

13.3 Related work

An early system for learning qualitative models named QuMAS, using a restricted form of logic, is described in Mozetič [1987] and Bratko et al. [1989]. The idea of QuMAS to transform the problem of learning in logic to propositional form was further developed in LINUS. QuMAS was used, however, to learn about a *static* system. It used a completely different set of primitives (background knowledge predicates) and is thus incomparable to the work on learning qualitative models of *dynamic* systems [Bratko et al. 1992].

GENMODEL [Coiera 1989] also generates a QSIM model in logic, using a special kind of least general generalization. Similarly to mFOIL, it uses strong typing of variables, so that the value and direction of change always appear together and cannot be mixed (unlike in FOIL and GOLEM where they can get mixed). It shares the limitation of

LINUS, namely no new variables may be introduced. However, it takes into account the corresponding values in the constraints.

The MISQ approach [Kraan et al. 1991, Richards et al. 1992] is essentially identical to GENMODEL, sharing most of GENMODEL's limitations, including the inability to introduce new variables. The useful additions include a facility of extracting qualitative behaviors out of quantitative data and generation of alternative maximally consistent models when incomplete information is given about the example behaviors. Furthermore, dimensional analysis is used to reduce the set of constraints generated.

Varšek [1991] applied a genetic algorithm QME (Qualitative Model Evolution) to the problem of learning qualitative models from examples. The corresponding values are not taken into account. Models are represented as trees in the genetic algorithm. The genetic operator of *crossover* exchanges subtrees between trees, while the *mutation* genetic operator generates random subtrees on single trees. Using a different training set, QME obtained five models equivalent to the correct one. In addition, QME was applied to several other small domains, including a RC-circuit (resistor-capacitor-circuit). Similarly to GENMODEL, QME can not introduce new variables.

PAL [Morales 1992], originally designed to learn chess patterns, is also based on the idea of least general generalization. It can also use non-ground background knowledge. Using only the four positive examples and the same background knowledge as mFOIL, PAL induced a model equivalent to the correct one. The model contains several redundant constraints, as PAL uses no negative examples to reduce it. Morales also successfully induced models of the U-tube system described with three (La, Lb, Fab) and five $(La, Lb, Fab, Fba, Diff)$ parameters. However, adding new system parameters requires additional effort and has to be handled in a special way in PAL. The *rlgg*-based systems PAL, GENMODEL and MISQ do not need negative examples.

13.4 Discussion

We have successfully applied the inductive logic programming system mFOIL to the problem of learning a qualitative model of the connected-containers dynamic system. The ability of mFOIL to use non-ground background knowledge proved useful in this respect and advantageous as compared to GOLEM and FOIL. mFOIL produced many models which correctly distinguish between the given positive and negative examples. Two of these proved to be equivalent to the correct model. At the same time, however, this reveals the necessity for criteria other than completeness, consistency and length (complexity) to distinguish among different qualitative models. Although the number of different models may be reduced by using dimensional analysis, this does not avoid the need for suitable criteria (bias).

Another problem with learning qualitative models is the problem of introducing new variables. Although GOLEM, FOIL and mFOIL can introduce new variables, the literals that introduce these variables are typically both non-determinate (the new variables can have more than one value) and non-discriminating (the literals do not distinguish between positive and negative examples). For example, if variable X has a positive qualitative value and variable Y has a negative value, the literal $add(X, Y, Z)$, where Z is a new variable will be non-determinate, as Z can be either positive, zero or negative. However, if X, Y and Z are described numerically, then Z is uniquely determined given X and Y. This suggests that a LINUS-like approach [Lavrač and Džeroski 1992a] operating on real-valued variables, where new variables are introduced before learning, coupled with the approach of learning qualitative models from real-valued data [Kraan et al. 1991], might be effective. It might also be possible to generate qualitative models directly from numerical data, without extracting qualitative behaviors.

14

Other ILP applications

This chapter gives an overview of several recent applications of ILP to problems taken from real-life domains: inducing rules for predicting the secondary structure of proteins [Muggleton et al. 1992a], learning rules for predicting structure–activity relationships in drug design [King et al. 1992], and inducing temporal diagnostic rules for satelite component faults [Feng 1992]. These examples illustrate the potential of ILP for real-world applications.

The ILP system GOLEM, described in Section 4.2, was used in the three applications described here. When inducing diagnostic rules, the training data is perfect as it is generated from a qualitative model simulated under a variety of conditions. Thus, no noise handling mechanisms are needed. In the task of predicting structure–activity relations for drugs, GOLEM has been used in a non-monotonic learning setting, where the encoding/compression scheme, mentioned in Section 8.4, was employed to handle data imperfections [Muggleton et al. 1992a, Srinivasan et al. 1992].

14.1 Predicting protein secondary structure

This section briefly reviews the problem of learning rules for predicting the secondary structure of proteins, to which GOLEM has been applied [Muggleton et al. 1992a]. We first describe the problem itself, then the relevant background knowledge, and finally summarize the results [Muggleton et al. 1992a, Sternberg et al. 1992].

A protein is basically a string of amino acids (or *residues*). Predicting the three-dimensional shape of proteins from their amino acid sequence is widely believed to be one of the hardest unsolved problems in molecular biology. It is also of considerable interest to the phar-

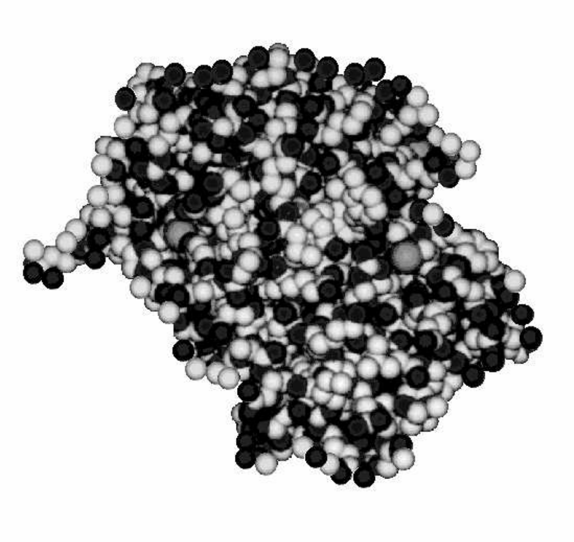


Figure 14.1: The 3D structure of a typical protein.

maceutical industry since shape generally determines the function of a protein. Figure 14.1 gives a model of the 3D structure of a typical protein.

The sequence of amino acids is called the *primary structure* of the protein. Spatially, the amino acids are arranged in different patterns (spirals, turns, flat sections, etc.). The three-dimensional spatial shape of a protein is called the *secondary structure*. When trying to predict the shape (*secondary structure*) it is easier to consider only one particular shape (pattern), instead of the multitude of possibilities; the α-helix (spiral) shape was considered by Muggleton et al. [1992a].

The target relation $alpha(Protein, Position)$ specifies that the residue at position $Position$ in protein $Protein$ belongs to an α-helix. Negative examples state all residue positions in particular proteins which are not in an α-helix secondary structure. For instance, the positive example $alpha(1HMQ, 104)$ means that the residue at position 104 in protein 1HMQ (hemerythrin met) is in an α-helix. The negative example $alpha(1HMQ, 105)$ states that the residue at position 106 in the same protein is not in an α-helix (it actually belongs to a β-coil secondary structure).

To learn rules to identify whether a position in a protein is in an

α-helix, Muggleton et al. [1992a] use the following information as background knowledge.

- The relation $position(A, B, C)$ states that the residue of protein A at position B is the amino-acid C. Each residue can be any one of the 20 possible amino acids and is denoted by a lower case character. For example, the fact $position(1HMQ, 111, g)$ says that the residue at position 111 in protein $1HMQ$ is glycine (using the standard one-character amino acid coding).

- The following arithmetic relations allow indexing of the protein sequence, relative to the residue for which secondary structure is being predicted. They have to be provided explicitly, as GOLEM has no built-in arithmetic relations. The relation $octf(A, B, C, D, E, F, G, H, I)$ specifies nine adjacent positions in a protein, i.e., it says that positions A to I occur in sequence. One fact in this relation may be, for instance, $octf(19, 20, 21, 22, 23, 24, 25, 26, 27)$. The $alpha\_triplet(A, B, C)$, $alpha\_pair(A, B)$ and $alpha\_pair4(A, B)$ relations allow a similar kind of indexing, which specifies residues that are likely to appear on the same face of an α-helix. They might be defined declaratively by the following three facts.

  $$alpha\_triplet(n, n+1, n+4).$$

  $$alpha\_pair(n, n+3).$$

  $$alpha\_pair4(n, n+4).$$

- Physical and chemical properties of individual residues are described by unary predicates. These properties include hydrophobicity, hydrophilicity, charge, size, polarity, whether a residue is aliphatic or aromatic, whether it is a hydrogen donor or acceptor, etc. Sizes, hydrophobicities, polarities etc., are represented by constants, such as $polar0, polar1, \ldots$. The use of different constant names for polarity zero $polar0$ and hydrophobicity zero $hydro0$ is necessary to prevent unjustified generalizations in GOLEM (these could be prevented by explicit statement of types instead). Relations between the constants, such as $less\_than(polar0, polar1)$, are also provided as background knowledge.

Let us now proceed with a very short description of the experimental setup and the results achieved by Muggleton et al. [1992a]. Sixteen proteins were used, twelve for training and four for testing. Without addressing in detail how GOLEM was run to induce rules from the training data, let us mention that each of the induced rules was allowed to misclassify up to 10 instances and some subjective statistical criteria were used to judge its significance before allowing it into the set of rules used for classification. The induced rules covered around 60% of the instances.

Level 0 rule

$$
\begin{aligned}
\mathit{alpha}0(A,B) \leftarrow\ & \mathit{octf}(D,E,F,G,B,H,I,J,K),\\
& \mathit{position}(A,D,Q), \mathit{not\_p}(Q), \mathit{not\_k}(Q),\\
& \mathit{position}(A,E,O), \mathit{not\_aromatic}(O),\\
& \mathit{small\_or\_polar}(O),\\
& \mathit{position}(A,F,R),\\
& \mathit{position}(A,G,P), \mathit{not\_aromatic}(P),\\
& \mathit{position}(A,B,C), \mathit{very\_hydrophobic}(C),\\
& \mathit{not\_aromatic}(C),\\
& \mathit{position}(A,H,M), \mathit{large}(M), \mathit{not\_aromatic}(M),\\
& \mathit{position}(A,I,L), \mathit{hydrophobic}(L),\\
& \mathit{position}(A,K,N), \mathit{large}(N), \mathit{ltv}(N,R).
\end{aligned}
$$

Level 1 rule

$$
\begin{aligned}
\mathit{alpha}1(A,B) \leftarrow\ & \mathit{octf}(D,E,F,G,B,H,I,J,K),\\
& \mathit{alpha}0(A,F), \mathit{alpha}0(A,G).
\end{aligned}
$$

Level 2 rule

$$
\begin{aligned}
\mathit{alpha}2(A,B) \leftarrow\ & \mathit{octf}(C,D,E,F,B,G,H,I,J),\\
& \mathit{alpha}1(A,B), \mathit{alpha}1(A,G), \mathit{alpha}1(A,H).
\end{aligned}
$$

Figure 14.2: Rules for predicting α-helix secondary structure.

To improve the coverage of these preliminary rules, the learning process was iterated. The predicted secondary structure positions found using the initial rules (called level 0 rules) were added to the background information. GOLEM was then run again to produce new (level 1) rules. This was necessary as the level 0 predictions were speckled, i.e. only short α-helix sequences were predicted. The level 1 rules in effect filtered the speckled predictions and joined together short sequences of α-helix predictions. The learning process was iterated once more with level 1 predictions added to the background knowledge and level 2 rules were induced. Finally, some of the level 1 and level 2 rules were generalized by hand by introducing their symmetric variants.

Applying GOLEM to the training set produced twenty-one level 0 rules, five symmetric level 1 rules and two symmetric level 2 rules. For illustration, Figure 14.2 gives one rule from each of level 0, level 1 and level 2.

The induced rules achieved accuracy of 78% on the training and 81% on the testing set. For comparison, the best previously reported result is 76%, achieved by using a neural network approach [Kneller et al. 1990]. The rules induced by GOLEM also have the advantage over the neural network method of being more understandable. The propositional learner PROMIS [King and Sternberg 1990] achieved 73% accuracy using the same data set as GOLEM. However, as the representation power of PROMIS is limited, it was not able to find some of the important relationships between residues that GOLEM found to be involved in α-helix formation.

14.2 Modeling structure–activity relationships

The majority of pharmaceutical research and development is based on finding slightly improved variants of patented active drugs. This involves laboratories of chemists synthesizing and testing hundreds of compounds. The ability to automatically discover rules that relate the activity of drugs to their chemical structure and chemical properties of their constituents (subcomponents) could provide great reduction in pharmaceutical research and development costs.

The problem of relating the structure of a chemical compound (drug) and the chemical properties of its constituents to a specific property (activity) of the compound is known as the problem of modeling quantitative structure–activity relationships (QSAR). In this section, a specific instance of this general problem is first described, namely, the

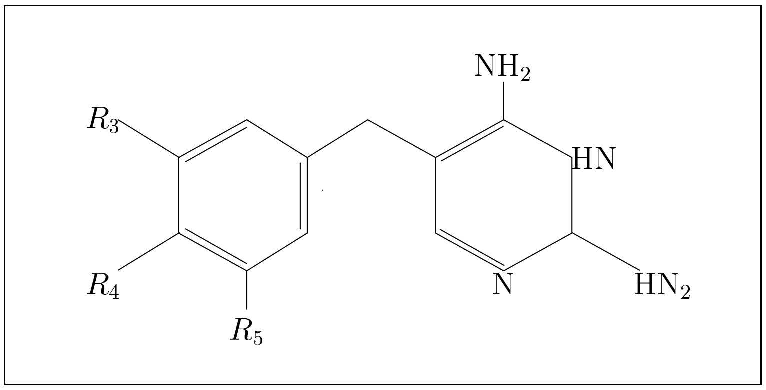


Figure 14.3: The structure of thrimethoprim analogues. From King et al. [1992] and Mladenić and Karalič [1992].

problem of modeling the QSAR of a series of drugs (trimethoprim analogues) inhibiting an enzyme (the *E. coli* dihydrofolate reductase) [King et al. 1992, Sternberg et al. 1992]. We next describe the formulation of this problem as a machine learning (classification) problem, including a description of the training examples and background knowledge as stated in [King et al. 1992]. We then summarize the results of the application of GOLEM to this problem [King et al. 1992] followed by the results of applying propositional learning systems to the same problem [Mladenić and Karalič 1992].

Trimethoprim analogues are compounds with a chemical structure as depicted in Figure 14.3. There are three alternate substitution positions (3, 4 and 5) on the phenyl ring. Drugs from this family are identified by the substituents R_3, R_4 and R_5.

King et al. [1992] used data on 44 trimethoprim analogues taken from Hansch et al. [1982] for training and 11 further cogeners taken from Roth et al. [1981, 1987] for testing. In these drugs, there are 25 possible choices for positions 3, 4 and 5. These include H (hydrogen, i.e., no substitution), Cl, F, Br, I, NH_2, NO_2, OH, $(OH)_2$, $CH_2O(CH_2)_3CH_3$, etc.

Each of the 24 non-hydrogen substituents is described with nine physico-chemical properties: polarity, size, flexibility, hydrogen-bond donor, hydrogen-bond acceptor, π donor, π acceptor, polarizability and σ effect. A list of the substituents together with their chemical properties can be found in King et al. [1992]. The chemical structures of the drugs and the substituents' properties listed above constitute the domain knowledge for the QSAR problem.

The biological activity of a drug (trimethoprim analogue) is measured as $-log(K_i)$, where K_i is the equilibrium constant for the association of the drug to dihydrofolate reductase [Hansch et al. 1982]. The activities of the training drugs range from 3.04 to 8.87, while the activities of the testing drugs are all high, ranging from 7.56 to 8.87. Note that the activity of a drug is a real-valued feature (variable) and not a discrete-valued one.

A standard approach to predict the biological activity of a drug is to formulate an empirical equation relating the activity of the drug to the physico-chemical properties of the corresponding substituents [Hansch et al. 1982]. The equation is typically formulated by using linear regression directly on the properties or on intermediate variables constructed from the properties by appropriate transformations. After finding the regression coefficients, the equation can be used to predict the activity of new compounds from the properties of their substituents.

To apply inductive logic programming to this domain, the QSAR problem is formulated in a slightly different way [King et al. 1992]. Instead of trying to predict directly the unknown activity of a compound, it is compared to the known activities of other compounds. A set of paired comparisons can then be converted into an overall ranking [David 1987].

The target relation $great(DrugX, DrugY)$ states that compound X exhibits higher biological activity than compound Y. Positive and negative examples are generated by comparing the activities of pairs of drugs. Paired examples of higher activity are positive examples (e.g., $great(d20, d15)$), and the corresponding pairs of lower activity are negative examples (e.g., $great(d15, d20)$). Pairs where the activities are equal (or within the margins of experimental measurement error) are discarded.

The background knowledge for GOLEM consists of the above mentioned domain knowledge and some explicitly stated arithmetic relations.

- The relation $struc(Drug, R_3, R_4, R_5)$ gives the chemical structure of trimethoprim analogues. For example, the background fact $struc(d40, CF_3, OCH_3, H)$ states that drug no. 40 has CF_3 substituted at position 3, OCH_3 substituted at position 4 and no substitution at position 5.

- A background predicate was introduced for each of the nine chemical properties of the substituents. E.g., $polar(CF_3, polar3)$ states that CF_3 has a polarity of 3, $polar(OCH_3, polar2)$ states that CF_3 has a polarity of 2, and $size(O(CH_2)_7CH_3, size5)$ states that $O(CH_2)_7CH_3$ is of size 5. Similar to the proteins work, described in the previous section, different constant names are used for each property.

- Some arithmetic knowledge was also explicitly included in the background knowledge, including relations between constants, such as $great\_polar(polar3, polar2)$ and relations between constants and fixed values, e.g., $great0\_polar(polar2)$.

GOLEM, used within the framework of non-monotonic learning [Srinivasan et al. 1992] and utilizing the encoding scheme of Muggleton et al. [1992b], induced nine rules comparing drug activities. An example rule is given in Figure 14.4. The rule states that drug A is better than drug B if drug B has no substitutions at positions 3 and 5 and the substitution D at position 3 of drug A has hydrogen donor 0, π-donor 1 and flexibility less than 4.

$$
\begin{aligned}
great(A, B) \leftarrow \quad & struc(A, D, E, F), \\
& struc(B, h, C, h), \\
& h\_donor(D, h\_don\_0), \\
& pi\_donor(D, pi\_don\_1), \\
& flex(D, G), \\
& less\_4\_flex(G).
\end{aligned}
$$

Figure 14.4: A rule for comparing drug activities.

Comparing the performance of GOLEM to the Hansch et al. [1982] approach, King et al. [1992] state that the Spearman rank correlation of the drug activity order predicted by GOLEM with the observed order is 0.92 on the training set of drugs and 0.46 for the testing set of drugs. For the predictions by the Hansch equation, the Spearman rank correlations are 0.79 and 0.42, respectively. Besides achieving slightly better accuracy than statistical methods, the induced rules also provide a description of the chemistry governing the problem.

Two propositional learners, ASSISTANT and RETIS, have recently been applied to the QSAR problem described above [Mladenić and Karalič 1992]. While ASSISTANT uses the paired comparison formulation, RETIS induces regression trees that predict activity in a similar way as the Hansch equation. Each drug is described with 22 attributes: the nine chemical properties of R_3, the nine chemical properties of R_4 and four additional boolean attributes ($R_3 = H$, $R_4 = H$, $R_5 = H$, $R_5 = R_3$).

The correlations achieved are better than the ones by GOLEM and the Hansch equation: 0.54 for ASSISTANT and 0.51 for RETIS on the testing drugs. These results reveal that for the particular QSAR problem at hand propositional learning systems might outperform ILP systems. One should not overlook, however, that ASSISTANT and RETIS have been given a small piece of relational knowledge ($R_3 = R_5$), borrowing an idea from LINUS [Lavrač et al. 1991a]. Also, Mladenić and Karalič [1992] do not address the problem of comprehensibility of the induced trees (classification and regression ones).

The main argument for the use of ILP lies in the generality of the background knowledge, training examples and concept description languages. Although it has been possible to formulate the QSAR problem for trimethoprim analogues in propositional form, this may be impossible or at least impractical for more general problems. Namely, the fixed structure and number of substitution positions make it easy to generate a propositional representation. In cases where the compounds under study are more complex, i.e., of less restricted structure, ILP may well be indispensable.

14.3 Learning diagnostic rules from qualitative models

The use of qualitative models to learn diagnostic rules was effectively demonstrated in the KARDIO expert system for diagnosing cardiac arrhythmias [Mozetič et al. 1984, Mozetič 1985a, Bratko et al. 1989]. The basic idea is to use a qualitative model of a system to generate behaviors of that system. If there is a fault in the system, it will be reflected in the behavior generated by the model. 'Faulty' behaviors generated by simulation can be used to provide examples for a learning program, which can then learn diagnostic rules relating the observable behavior of the system to the internal causes (diagnoses).

The KARDIO methodology is fairly general and can be applied to other domains besides the domain of diagnosing cardiac arrhythmias from ECGs (electrocardiograms). In fact, it has already been applied to generate diagnostic rules for the electrical power supply subsystem of a satellite [Pearce 1988a, 1988b] that relate the status of visible indicators to the diagnoses. It turns out, however, that some faults in this system cannot be diagnosed by simply looking at the visible indicators at some time point, but require knowledge of the indicator values at one or more previous time points. For example, the negative outcome of a simple check of power from the battery at time T_i is not sufficient to conclude that the battery is faulty; namely, if the battery were completely discharged at time T_{i-1}, it is only natural that it provides no power at time T_i, i.e., before it is recharged.

This section gives an overview of the KARDIO methodology [Bratko et al. 1989], illustrated on a simple electric circuit example. It then briefly summarizes the application of this methodology to the problem of diagnosing faults in the electrical power subsystem of a satellite [Pearce 1988a, 1988b], where a propositional learner was used to induce diagnostic rules from a qualitative model. Finally, it states the formulation of the task of learning temporal diagnostic rules for the same problem as an ILP task [Feng 1991, 1992].

14.3.1 The KARDIO methodology

In the KARDIO diagnostic system [Bratko et al. 1989], the prevailing type of knowledge is deep, capturing the underlying causal structure of the domain. In contrast, shallow knowledge states the relation between the diagnostic problem and its solution directly, without referring to the underlying principles. While a deep model of a device facilitates the task of explaining how the device works, surface knowledge may be sufficient or even preferrable when diagnosing faults in the device, since the use of surface knowledge is typically more efficient.

It is sensible to use different representations of the domain knowledge for different tasks. Bratko et al. [1989] argue that once some representation is given, it is most appropriate to automatically transform this representation into the one that is best suited for the particular task at hand. In KARDIO, surface knowledge was generated from the deep model by simulation, and was then compressed by inductive learning tools into efficient diagnostic and prediction rules.

To illustrate the notions introduced in the above two paragraphs, let us consider the simple electric circuit shown in Figure 14.5. A model of the circuit, represented as a Prolog program, is given in Figure 14.6. The circuit, as a system, is specified by its components (bulbs, battery) and the way they are interconnected. The behavior of the system can be derived from the behavior of the components, which is also defined in the model.

The predicate *circuit* defines the behavior of the circuit as a whole. The argument *BatteryState* denotes the functional state of the battery

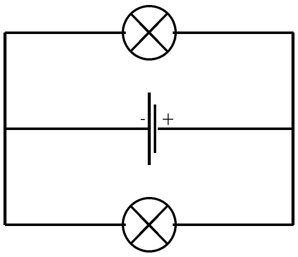


Figure 14.5: A simple electric circuit.

```
% Circuit definition
circuit(BatteryState,Bulb1State,Bulb2State,Light1,Light2) ←
    battery(BatteryState, Voltage),
    bulb(Bulb1State, Voltage,Light1),
    bulb(Bulb2State, Voltage,Light2).

% Battery definition: battery(BatteryState, Voltage)
battery(battery_ok,voltage_nonzero).
battery(battery_faulty,voltage_zero).

% Bulb definition: bulb(BulbState, Voltage,Light)
bulb(bulb_ok,voltage_nonzero,light).
bulb(bulb_ok,voltage_zero,dark).
bulb(bulb_blown,_,dark).
```

Figure 14.6: Qualitative model of a simple electric circuit.

(ok or faulty), *Bulb1State* and *Bulb2State* denote the functional states of the two bulbs (*ok* or *blown*), while *Light1* and *Light2* denote the lightness of the bulbs (*light, dark*). The behavior of both bulbs is defined with the predicate *bulb* and the behavior of the battery with the predicate *battery*. The argument *Voltage* denotes the voltage (non-zero, zero) on the battery and the bulbs. Note that the voltage is described by a qualitative value, rather than a quantitative (numerical) one.

Figure 14.7 gives an excerpt from a shallow, extensional representation of the circuit. The excerpt is equivalent to the deep model under the assumption that at most one component in the circuit is faulty at any single moment in time. Note that the lightness of the bulbs represents the observable behavior of the system. Thus the shallow knowledge directly relates the state (diagnosis) of the circuit to the observable behavior of the system.

Each of the facts in the figure can be derived by setting the functional states of the circuit components and then simulating the behavior of the circuit. For example, to derive the third fact in the figure, all we need to do is input the model from Figure 14.6 into a Prolog interpreter and then ask the question

```
?- circuit(battery_ok,bulb_blown,bulb_ok,Light1,Light2).
```

```
% Single fault assumption
% circuit(BatteryState,Bulb1State,Bulb2State,Light1,Light2)
% Circuit behavior description                                      % Diagnosis

circuit(battery_ok,bulb_ok,bulb_ok,light,light).                           % OK
circuit(battery_faulty,bulb_ok,bulb_ok,dark,dark).             % Faulty battery
circuit(battery_ok,bulb_blown,bulb_ok,dark,light).               % Bulb1 blown
circuit(battery_ok,bulb_ok,bulb_blown,light,dark).             % Bulb2 blown
```

Figure 14.7: Shallow representation of a simple electric circuit.

The Prolog interpreter would then reply

```
Light1 = dark
Light2 = light
```

In this case, we have used the Prolog interpreter to do the qualitative simulation. For efficiency or other reasons, it might be necessary to implement a special interpreter for the application domain at hand, as was done in KARDIO.

The task of diagnosis can be viewed as a classification task. In this case, the observable features are the attributes and the diagnosis (state of the system) is the class. Entries in the shallow representation can be considered examples for a propositional learning system. Diagnostic rules for the circuit under the single fault assumption are given in Figure 14.8.

In our example, to each entry in the shallow representation corresponds one diagnostic rule. However, this is a consequence of the single fault assumption and need not be true in general. Indeed, one particular fault can manifest itself in different ways, and a set of symptoms may be caused by different faults. In such cases, diagnostic rules induced from a shallow representation may be substantially shorter than the shallow representation itself [Bratko et al. 1989], or even the original deep model [Pearce 1988b]. In addition, if the induced rules correctly classify all training examples, they are logically equivalent to the deep model at the surface level.

Having elaborated the simple circuit example, let us now summarize the KARDIO project. An attempt to develop an expert system without

$Diagnosis = ok$ **if** $[Light1 = light] \wedge [Light2 = light]$
$Diagnosis = bulb1\_blown$ **if** $[Light1 = dark] \wedge [Light2 = light]$
$Diagnosis = bulb2\_blown$ **if** $[Light1 = light] \wedge [Light2 = dark]$
$Diagnosis = faulty\_battery$ **if** $[Light1 = dark] \wedge [Light2 = dark]$

Figure 14.8: Diagnostic rules for a simple electric circuit.

a deep model failed, due to the combinatorial complexity of the problem of diagnosing multiple cardiac arrhythmias. Consequently, a deep qualitative model of the electrical activity of the heart was constructed by hand [Mozetič 1984]. This model relates the functional state of components of the electrical system of the heart to electrocardiograms (ECGs), i.e., records of the electrical activity of the heart. A shallow representation was then generated by simulating all physiologically possible and medically interesting faults of the heart electrical system (arrhythmias), obtaining explicit relationships between arrhythmias and ECGs [Mozetič et al. 1984]. An expert system that uses the derived surface knowledge was then developed [Lavrač et al. 1985]. The surface knowledge was compressed by using propositional learning systems to obtain both diagnostic and prediction rules [Mozetič 1985a].

In fact, Mozetič went further by automating the synthesis of a deep model from surface knowledge [Mozetič 1987, Mozetič 1988]. While a straightforward application of propositional approaches sufficed for the compression of surface knowledge into diagnostic rules, it was insufficient for the harder task of learning deep models. Mozetič thus developed the special purpose learning system QuMAS [Mozetič 1987], which uses the idea of transforming the problem of learning relations to a propositional form. As we have seen earlier in the book, this idea is at the core of LINUS, which is a general purpose system for learning relations, i.e., an ILP system. In addition, an abstraction hierarchy was introduced into the heart model, which facilitates the learning process [Mozetič 1988].

Let us conclude this section by illustrating how the task of synthesising a deep model from surface knowledge can be formulated as an ILP task. Consider again the simple circuit of Figure 14.5. The target predicate is $circuit(Bat, Bulb1, Bulb2, Light1, Light2)$ and the training examples are of the form given in Figure 14.7. The background knowledge consists of the predicates $battery(State, Voltage)$ and $bulb(State, Voltage, Light)$. From 32 examples specifying legal and illegal behaviors of the circuit and the background knowledge described above, mFOIL was able to induce the correct model.

The training examples specify the functionality we require from our device, while the background knowledge specifies the functionality of the components at hand. The model to be induced would specify which components should be connected and in what way. In other words, when inducing a model from the desired behavior, we are actually designing a device from first principles. This formulation reveals that a wide range of design and modeling problems is amenable to the application of ILP techniques. A discusion of the potential applications of ILP in qualitative reasoning can be found in Bratko [1993].

14.3.2 Learning temporal diagnostic rules

The KARDIO methodology described in the previous section can be applied to other domains besides the domain of diagnosing cardiac arrhythmias from ECGs (electrocardiograms) where it was originally applied. In fact, it has already been applied to generate diagnostic rules for the electrical power supply subsystem of a satellite [Pearce 1988a, 1988b] that relate the status of indicators to the states of system components, i.e., to diagnoses. While Pearce describes the satellite power subsystem briefly, Feng [1992] gives a more detailed description. In the following, we first give a digest of the latter description. We then briefly summarize the application of the KARDIO methodology to the power subsystem done by Pearce [1988a], which ignores important time-dependent aspects. Finally, we describe the formulation of the problem of learning temporal diagnostic rules in the framework of ILP [Feng 1992].

The power subsystem of the Skynet satellite consists of several units, including an array of solar panels, an array switching regulator, an electrical integration unit, a battery pack and a battery control unit. The structure of the power subsystem is illustrated in Figure 14.9. During orbit, the work cycle of the power subsystem consists of three mission phases: *presolstice, solstice* and *eclipse.* In addition, the battery may be *reconditioned* during solstice. The system is configured differently for each mission phase by ground operator telecommands.

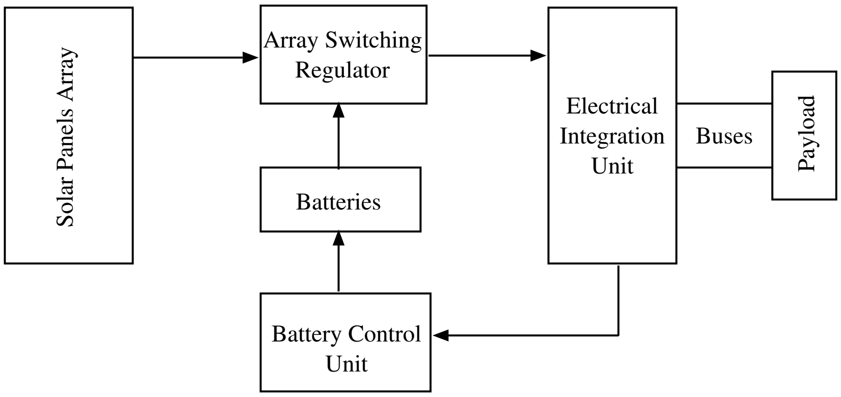


Figure 14.9: Block diagram of satellite power subsystem.

During battery charging (presolstice), solstice and reconditioning, there is constant sunlight, so the solar panels array provides the power needed for the satellite. In eclipse, there is no sunlight, so power is supplied from the battery pack, which is then charged during presolstice. To maintain the charging voltage untill the next eclipse, the battery is trickle-charged during solstice. Occasionally, the battery pack needs to be reconditioned (i.e., to discharge all electricity) in order to maintain battery health. It should then be recharged.

The charging and discharging of batteries is controlled by the battery control unit, which sends End of Charging (EoC) and End of Discharging (EoD) signals when appropriate. The EoC signal is triggered by either a battery over-temperature signal, a battery over-voltage signal or by telecommand from ground operators. The EoD signal is generated by either a battery under-voltage signal, on exiting eclipse or by telecommand from the ground.

The switch asr_switch_10 controls battery charging and is closed during charging. If it is accidentally opened, a fault will arise in battery charging. Similarly, the switch $relay\_a038$ short-cuts the battery to reconditioning and an accidental short-circuit involving this switch will result in loss of power of the battery pack.

Many components of the power subsystem are equipped with sensors and the sensor readings are transmitted regularly to the ground. Sensor readings include bus voltage, power supply from solar panels and battery voltage. Pearce [1988a, 1988b] applied the KARDIO methodology to induce diagnostic rules that infer faults from the sensor readings.

Pearce first constructed a qualitative model of the satelite power subsystem. Using heuristic knowledge about component behavior all possible component failures were simulated to generate a set of examples. As a result of component failures sensor readings changed. Following a simulated failure, the sensor readings were used as attributes and the diagnosis (fault) as the class. The examples were compressed by using the propositional learning system AQR [Clark and Niblett 1987], a variant of AQ [Michalski 1983], to obtain diagnostic rules.

It turns out, however, that some faults in this system cannot be diagnosed by simply looking at the visible indicators at some time point, but require a knowledge of the indicator values at one or more previous time points. For example, the negative outcome of a simple check of power from the battery at time T_i is not sufficient to conclude that the battery is faulty; namely, if the battery was reconditioned at time T_{i-1}, it is only natural that it provides no power at time T_i, i.e., before it is recharged. Other faults of this type can occur in the comparator component of the array switching regulator and in the operations from ground control.

Such faults, which can only be detected after a certain time delay, could not be diagnosed with the rules induced by Pearce [1988a], due to the propositional setup. These faults are relationally dependent on time in the sense that a fault at time T_i is caused by an operation or the failure of another component at earlier times T_j and T_k. Feng [1991, 1992] extended the qualitative model of Pearce [1988a] to incorporate temporal elements and then applied the KARDIO methodology to induce temporal diagnostic rules.

In the temporal qualitative model, each component is described by a predicate $component_i(State, T_j)$, which gives the state of $component_i$ at all particular time points T_j. The relation between successive time points is given by the predicate $succeed(T_i, T_j)$, which denotes that time point T_j follows T_i. The model also includes the predicate $mission\_phase(Phase, T_i)$, that states the mission phase (presolstice/charging, solstice, reconditioning, eclipse) at time T_i.

The simulator based on this model takes an initial state of the system and a component that is assumed faulty. It then moves from state to state according to transition rules, until the system reaches a stable state where there is no change of component states through several transitions. In each transition (at each time step) the states of the sensors are recorded. For each component fault, a separate ILP task was defined, to which GOLEM was applied [Feng 1992] in order to compress the simulated data and obtain temporal diagnostic rules. For illustration we will only consider the ILP formulation of the task of learning temporal diagnostic rules for battery faults.

In the ILP task of learning temporal diagnostic rules for battery faults, the target relation $fault(T_i)$ states that there was a battery fault at time point T_i. For example, the fact $fault(300)$ states that there is a battery fault at time point 300. The background knowledge contains the following information:

- Recordings of 29 sensors at each time point. A predicate $sensor_i(V, T_j)$ states that the sensor $sensor_i$ at time T_j has the value V. Figure 14.10 gives the sensor readings for all 29 sensors at time point 300.

- The predicate $mission\_phase(Phase, T_i)$ that states the mission phase at time T_i. The fact $mission\_phase(presolstice, 300)$ states that at time point 300 the satellite power subsystem is in the presolstice (charging) phase.

- The relation between successive time points given by the predicate $succeed(T_i, T_j)$, which denotes that time point T_j follows T_i. For example, the fact $succeed(300, 301)$ states that time point 301 immediately follows time point 300.

A typical rule induced from the above formulated ILP task would have the following form:

$$
\begin{aligned}
& fault(T_i) \leftarrow \\
& \quad sensor_{i,1}(V_{i,1}, T_i),\ sensor_{i,2}(V_{i,2}, T_i), \ldots \\
& \quad succeed(T_j, T_i), \\
& \quad sensor_{j,1}(V_{j,1}, T_j),\ sensor_{j,2}(V_{j,2}, T_j), \ldots \\
& \quad succeed(T_k, T_j), \\
& \quad sensor_{k,1}(V_{k,1}, T_k),\ sensor_{k,2}(V_{k,2}, T_k), \ldots
\end{aligned}
$$

```
tm040_switch(negative,300).
tm018_switch(negative,300).
tm031_switch(positive,300).
tm038_switch(positive,300).
tm022_switch(positive,300).
tm043_switch(negative,300).
tm013_switch(positive,300).
tm042_switch(positive,300).
tm007_switch(positive,300).
tm222_charging(positive,300).
tm071_asr_or_switch_20(positive,300).
tm070_supply_3c(positive,300).
tm058_asr_or_switch_10(positive,300).
tm057_supply_2c(negative,300).
tm257_battery_voltage(negative,300).
tm017_switch(negative,300).
tm009_switch(positive,300).
tm220_supply_1c(positive,300).
tm055_supply_1b(negative,300).
tm054_supply_1a(positive,300).
tm029_ovt_disabled(positive,300).
tm021_eoc_disabled(positive,300).
tm004_eoc_signaled(negative,300).
tm002_battov_temp(negative,300).
tm039_eod_disabled(positive,300).
tm015_eod_signaled(negative,300).
tm011_eod_override(positive,300).
tm001_eod_relay(positive,300).
tm211_bus_voltage(positive,300).
```

Figure 14.10: Sensor readings for the satellite power subsystem at time point 300.

Feng [1992] also applied GOLEM to learn diagnostic rules for comparator and operator faults. He also attempted to learn diagnostic rules for multiple faults. For more details, we refer the reader to Feng [1991, 1992].

14.4 Discussion

Until now, successful applications of machine learning used attribute-value learners. In this chapter, three real-world applications of inductive logic programming were outlined. We described the problem of learning rules for predicting the secondary structure of proteins, the problem of learning structure–activity rules and the problem of inducing temporal diagnostic rules from a qualitative model. Although we focused on the formulation of the tasks as ILP problems and the relevant background knowledge, we can state that the achieved results are encouraging.

In the applications described in this chapter, the ILP system GOLEM was used. In the previous chapters, two real-life applications of LINUS and mFOIL were presented, namely, learning medical rules for early diagnosis of rheumatic diseases, and learning rules for finite element mesh design. The application of mFOIL to learning qualitative models of dynamic systems was also described.

The above applications are some of the first attempts of ILP towards knowledge synthesis from real-life data. One should be aware that these tasks are specific instances of more general tasks, e.g., inducing temporal diagnostic rules for a device described by a temporal qualitative model, inducing rules for prediction of various properties of chemicals from their structure, and learning medical diagnostic rules by taking into account medical background knowledge. Consequently, the applications described in this book illustrate the potential of ILP for real-life applications.

Bibliography

[Adé and Bruynooghe 1992]
Adé, H. and Bruynooghe, M. (1992). A comparative study of declarative and dynamically adjustable language bias in concept learning. In *Proc. Workshop on Biases in Inductive Learning, Ninth International Conference on Machine Learning*. Aberdeen, Scotland.

[Bain 1991]
Bain, M. (1991). Experiments in non-monotonic learning. In *Proc. Eighth International Workshop on Machine Learning*, pages 380–384. Morgan Kaufmann, San Mateo, CA.

[Bergadano and Giordana 1988]
Bergadano, F. and Giordana, A. (1988). A knowledge intensive approach to concept induction. In *Proc. Fifth International Conference on Machine Learning*, pages 305–317. Morgan Kaufmann, San Mateo, CA.

[Bergadano and Giordana 1990]
Bergadano, F. and Giordana, A. (1990). Guiding induction with domain theories. In Kodratoff, Y. and Michalski, R., editors, *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 474–492. Morgan Kaufmann, San Mateo, CA.

[Bergadano et al. 1989]
Bergadano, F., Giordana, A., and Ponsero, S. (1989). Deduction in top-down inductive learning. In *Proc. Sixth International Workshop on Machine Learning*, pages 23–25. Morgan Kaufmann, San Mateo, CA.

[Bloedorn and Michalski 1991]
Bloedorn, E. and Michalski, R. (1991). Data-driven constructive

induction in AQ17-PRE: A method and experiments. In *Proc. Third International Conference on Tools for Artificial Intelligence*, pages 30–37. IEEE Computer Society Press, Los Alamitos, CA.

[Bratko 1989]
Bratko, I. (1989). Machine learning. In Gilhooly, K., editor, *Human and Machine Problem Solving.* Plenum Press, New York.

[Bratko 1990]
Bratko, I. (1990). *Prolog Programming for Artificial Intelligence.* Addison-Wesley, Wokingham, 2nd edition.

[Bratko 1991]
Bratko, I. (1991). Qualitative modelling: learning and control. In *Proc. International Conference on Artificial Intelligence.* Prague.

[Bratko 1992]
Bratko, I. (1992). Applications of machine learning: Towards knowledge synthesis. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 1207–1218. Tokyo.

[Bratko 1993]
Bratko, I. (1993). Machine learning and qualitative reasoning. *Machine Learning.* To appear.

[Bratko and Grobelnik 1993]
Bratko, I. and Grobelnik, M. (1993). Inductive learning applied to program construction and verification. In *Proc. Third International Workshop on Inductive Logic Programming*, pages 279–292. Bled, Slovenia.

[Bratko et al. 1989]
Bratko, I., Mozetič, I., and Lavrač, N. (1989). *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems.* MIT Press, Cambridge, MA.

[Bratko et al. 1991]
Bratko, I., Muggleton, S., and Varšek, A. (1991). Learning qualitative models of dynamic systems. In *Proc. Eighth International Workshop on Machine Learning*, pages 385–388. Morgan Kaufmann, San Mateo, CA.

[Bratko et al. 1992]
Bratko, I., Muggleton, S., and Varšek, A. (1992). Learning qualitative models of dynamic systems. In Muggleton, S., editor, *Inductive Logic Programming*, pages 437–452. Academic Press, London.

[Brazdil and Clark 1990]
Brazdil, P. and Clark, P. (1990). Learning from imperfect data. In Brazdil, P. and Konolige, K., editors, *Machine Learning, Meta-Reasoning and Logics*, pages 207–232. Kluwer, Boston.

[Breiman et al. 1984]
Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees.* Wadsworth, Belmont.

[Brunk and Pazzani 1991]
Brunk, C. and Pazzani, M. (1991). An investigation of noise-tolerant relational concept learning algorithms. In *Proc. Eighth International Workshop on Machine Learning*, pages 389–393. Morgan Kaufmann, San Mateo, CA.

[Bry 1990]
Bry, F. (1990). Query evaluation in recursive databases. *Data and Knowledge Engineering*, 5:289–312.

[Buntine 1988]
Buntine, W. (1988). Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176.

[Buntine and Niblett 1992]
Buntine, W. and Niblett, T. (1992). A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 8(1):75–85.

[Cameron-Jones and Quinlan 1993]
Cameron-Jones, R. and Quinlan, J. (1993). Avoiding pitfalls when learning recursive theories. In *Proc. Thirteenth International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, San Mateo, CA.

[Carbonell 1989]
Carbonell, J. (1989). Introduction: Paradigms for machine learning. *Artificial Intelligence*, 40(1–3):1–9.

[Cestnik 1990]
Cestnik, B. (1990). Estimating probabilities: A crucial task in machine learning. In *Proc. Ninth European Conference on Artificial Intelligence*, pages 147–149. Pitman, London.

[Cestnik 1991]
Cestnik, B. (1991). *Estimating probabilities in machine learning.* PhD thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia. In Slovenian.

[Cestnik and Bratko 1991]
Cestnik, B. and Bratko, I. (1991). On estimating probabilities in tree pruning. In Kodratoff, Y., editor, *Proc. Fifth European Working Session on Learning*, pages 151–163. Springer, Berlin.

[Cestnik et al. 1987]
Cestnik, B., Kononenko, I., and Bratko, I. (1987). ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In Bratko, I. and Lavrač, N., editors, *Progress in Machine Learning*, pages 31–45. Sigma Press, Wilmslow, UK.

[Clark 1978]
Clark, K. (1978). Negation as failure. In Gallaire, H. and Minker, J., editors, *Logic and Databases*, pages 293–322. Plenum Press, New York.

[Clark and Boswell 1991]
Clark, P. and Boswell, R. (1991). Rule induction with CN2: Some recent improvements. In *Proc. Fifth European Working Session on Learning*, pages 151–163. Springer, Berlin.

[Clark and Niblett 1987]
Clark, P. and Niblett, T. (1987). Induction in noisy domains. In Bratko, I. and Lavrač, N., editors, *Progress in Machine Learning*, pages 11–30. Sigma Press, Wilmslow, UK.

[Clark and Niblett 1989]
Clark, P. and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, 3(4):261–283.

[Coiera 1989]
Coiera, E. (1989). Learning qualitative models from example behaviours. In *Proc. Third International Workshop on Qualitative Physics*. Stanford, California.

[David 1987]
David, H. (1987). Ranking from unbalanced paired-comparison data. *Biometrica*, 74:432–436.

[De Kleer and Brown 1984]
De Kleer, J. and Brown, J. (1984). A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83.

[De Raedt 1992]
De Raedt, L. (1992). *Interactive Theory Revision: An Inductive Logic Programming Approach.* Academic Press, London.

[De Raedt 1993]
De Raedt, L. (1993). Inductive logic programming and scientific discovery. Technical report, Katholieke Universiteit Leuven, Leuven, Belgium.

[De Raedt and Bruynooghe 1989]
De Raedt, L. and Bruynooghe, M. (1989). Towards friendly concept-learners. In *Proc. Eleventh International Joint Conference on Artificial Intelligence*, pages 849–854. Morgan Kaufmann, San Mateo, CA.

[De Raedt and Bruynooghe 1992a]
De Raedt, L. and Bruynooghe, M. (1992a). Interactive concept learning and constructive induction by analogy. *Machine Learning*, 8(2):107–150.

[De Raedt and Bruynooghe 1992b]
De Raedt, L. and Bruynooghe, M. (1992b). A unifying framework concept-learning algorithms. *The Knowledge Engineering Review*, 7(3):251–269.

[De Raedt and Bruynooghe 1993]
De Raedt, L. and Bruynooghe, M. (1993). A theory of clausal discov-

ery. In *Proc. Thirteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA.

[De Raedt et al. 1991]
De Raedt, L., Bruynooghe, M., and Martens, B. (1991). Integrity constraints in interactive concept learning. In *Proc. Eighth International Workshop on Machine Learning*, pages 394–398. Morgan Kaufmann, San Mateo, CA.

[De Raedt et al. 1993a]
De Raedt, L., Lavrač, N., and Džeroski, S. (1993a). Multiple predicate learning. In *Proc. Thirteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA.

[De Raedt et al. 1993b]
De Raedt, L., Lavrač, N., and Džeroski, S. (1993b). Multiple predicate learning. In *Proc. Third International Workshop on Inductive Logic Programming*, pages 221–240. Bled, Slovenia.

[Dietterich and Michalski 1986]
Dietterich, T. and Michalski, R. (1986). Learning to predict sequences. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning: An Artificial Intelligence Approach*, volume II. Morgan Kaufmann, Los Altos, CA.

[Dolšak 1991]
Dolšak, B. (1991). Constructing finite element meshes using artificial intelligence methods. Master's thesis, Faculty of Technical Sciences, University of Maribor, Maribor, Slovenia. In Slovenian.

[Dolšak et al. 1992]
Dolšak, B., Jezernik, A., and Bratko, I. (1992). A knowledge base for finite element mesh design. In *Proc. Sixth ISSEK Workshop.* Jožef Stefan Institute, Ljubljana, Slovenia.

[Dolšak and Muggleton 1992]
Dolšak, B. and Muggleton, S. (1992). The application of inductive logic programming to finite element mesh design. In Muggleton, S., editor, *Inductive Logic Programming*, pages 453–472. Academic Press, London.

[Džeroski 1991]
Džeroski, S. (1991). Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia.

[Džeroski and Bratko 1992a]
Džeroski, S. and Bratko, I. (1992a). Handling noise in inductive logic programming. In *Proc. Second International Workshop on Inductive Logic Programming*. Tokyo, Japan. ICOT TM-1182.

[Džeroski and Bratko 1992b]
Džeroski, S. and Bratko, I. (1992b). Using the m-estimate in inductive logic programming. In *Proc. Workshop on Logical Approaches to Machine Learning, Tenth European Conference on Artificial Intelligence*. Vienna, Austria.

[Džeroski and Dolšak 1991]
Džeroski, S. and Dolšak, B. (1991). A comparison of relation learning algorithms on the problem of finite element mesh design. In *Proc. XXVI Yugoslav Conference of the Society for ETAN*. Ohrid, Yugoslavia. In Slovenian.

[Džeroski and Dolšak 1992]
Džeroski, S. and Dolšak, B. (1992). Comparison of ilp systems on the problem of finite element mesh design. In *Proc. Sixth ISSEK Workshop*. Jožef Stefan Institute, Ljubljana, Slovenia.

[Džeroski and Lavrač 1991a]
Džeroski, S. and Lavrač, N. (1991a). Learning relations from imperfect data. Technical Report IJS-DP-6163, Jožef Stefan Institute, Ljubljana, Slovenia.

[Džeroski and Lavrač 1991b]
Džeroski, S. and Lavrač, N. (1991b). Learning relations from noisy examples: An empirical comparison of LINUS and FOIL. In *Proc. Eighth International Workshop on Machine Learning*, pages 399–402. Morgan Kaufmann, San Mateo, CA.

[Džeroski and Lavrač 1992]
Džeroski, S. and Lavrač, N. (1992). Refinement graphs for FOIL

and LINUS. In Muggleton, S., editor, *Inductive Logic Programming*, pages 319–333. Academic Press, London.

[Džeroski et al. 1992a]
Džeroski, S., Cestnik, B., and Petrovski, I. (1992a). The use of Bayesian probability estimates in rule induction. In *Proc. First Slovenian Electrical Engineering and Computer Science Conference*, pages 155–158. Slovenia Section IEEE, Ljubljana.

[Džeroski et al. 1992b]
Džeroski, S., Muggleton, S., and Russell, S. (1992b). PAC-learnability of constrained nonrecursive logic programs. In *Proc. Third International Workshop on Computational Learning Theory and Natural Learning Systems.* Wisconsin, Madison.

[Džeroski et al. 1992c]
Džeroski, S., Muggleton, S., and Russell, S. (1992c). PAC-learnability of determinate logic programs. In *Proc. Fifth ACM Workshop on Computational Learning Theory*, pages 128–135. ACM Press, New York, NY.

[Falkenheiner and Forbus 1990]
Falkenheiner, B. and Forbus, K. (1990). Self-explanatory simulations: an integration of quantitative and qualitative knowledge. In *Proc. Fourth International Workshop on Qualitative Physics.* Lugano, Switzerland.

[Feng 1991]
Feng, C. (1991). Inducing temporal fault diagnostic rules from a qualitative model. In *Proc. Eighth International Workshop on Machine Learning*, pages 403–406. Morgan Kaufmann, San Mateo, CA.

[Feng 1992]
Feng, C. (1992). Inducing temporal fault diagnostic rules from a qualitative model. In Muggleton, S., editor, *Inductive Logic Programming*, pages 471–493. Academic Press, London.

[Flach 1992]
Flach, P. (1992). Logical approaches to machine learning – an overview. *THINK*, 1(2):25–36.

[Flach 1993]
Flach, P. (1993). Predicate invention in inductive data engineering. In Brazdil, P., editor, *Proc. Sixth European Conference on Machine Learning*, pages 83–94. Springer, Berlin.

[Forbus 1984]
Forbus, K. (1984). Qualitative process theory. *Artificial Intelligence*, 24:85–168.

[Frawley et al. 1991]
Frawley, W., Piatetsky-Shapiro, G., and Matheus, C. (1991). Knowledge discovery in databases: an overview. In Piatetsky-Shapiro, G. and Frawley, W., editors, *Knowledge discovery in databases*, pages 1–27. AAAI Press/MIT Press, Menlo Park, CA/Cambridge, MA.

[Grobelnik 1992]
Grobelnik, M. (1992). MARKUS: An optimized Model Inference System. In *Proc. Workshop on Logical Approaches to Machine Learning, Tenth European Conference on Artificial Intelligence*. Vienna, Austria.

[Hansch et al. 1982]
Hansch, C., Li, R., Blaney, J., and Langridge, R. (1982). Comparison of the inhibition of escherichia coli and lactobacillus casei dihydrofolate reductase by 2,4-diamino-5-(substituted-benzyl)pyrimidines: Quantitative structure-activity relationships, x-ray crystallography, and computer graphics in structure-activity analysis. *J. Med. Chem.*, 25:777–784.

[Harmon et al. 1988]
Harmon, P., Maus, R., and Morrissey, W. (1988). *Expert systems: Tools & Applications.* John Wiley, New York.

[Hinton 1989]
Hinton, G. E. (1989). Connectionist learning procedures. *Artificial Intelligence*, 40(1–3):185–234.

[Hunt et al. 1966]
Hunt, E. B., Marin, J., and Stone, P. J. (1966). *Experiments in Induction.* Academic Press, New York.

[Kalbfleish 1979]
Kalbfleish, J. (1979). *Probability and Statistical Inference*, volume II. Springer, New York.

[Karalič and Pirnat 1990]
Karalič, A. and Pirnat, V. (1990). Machine learning in rheumatology. *Sistemica*, 1(2):113–123.

[Kietz and Wrobel 1992]
Kietz, J. and Wrobel, S. (1992). Controlling the complexity of learning in logic through syntactic and task-oriented models. In Muggleton, S., editor, *Inductive Logic Programming*, pages 335–359. Academic Press, London.

[Kijsirikul et al. 1991]
Kijsirikul, B., Numao, M., and Shimura, M. (1991). Efficient learning of logic programs with non-determinate non-discriminating literals. In *Proc. Eighth International Workshop on Machine Learning*, pages 417–421. Morgan Kaufmann, San Mateo, CA.

[Kijsirikul et al. 1992]
Kijsirikul, B., Numao, M., and Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proc. Tenth National Conference on Artificial Intelligence.* Morgan Kaufmann, San Mateo, CA.

[King and Sternberg 1990]
King, R. and Sternberg, M. (1990). Machine learning approach for the prediction of protein secondary structure. *J. Mol. Biol.*, 216:441–457.

[King et al. 1992]
King, R., Muggleton, S., Lewis, R., and Sternberg, M. (1992). Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proc. National Academy of Sciences.*

[Kneller et al. 1990]
Kneller, D., Cohen, F., and Langridge, R. (1990). Improvements in

protein secondary structure prediction by an enhanced neural network. *J. Mol. Biol.*, 214:171–182.

[Kodratoff et al. 1992]
Kodratoff, Y., Sleeman, D., Uszynski, M., Causse, K., and Craw, S. (1992). Building a machine learning toolbox. In Le Pape, B. and Steels, L., editors, *Enhancing the Knowledge Engineering Process – Contributions from ESPRIT*. Elsevier.

[Kononenko and Bratko 1991]
Kononenko, I. and Bratko, I. (1991). Information-based evaluation criterion for classifier's performance. *Machine Learning*, 6(1):67–80.

[Kraan et al. 1991]
Kraan, I., Richards, B., and Kuipers, B. (1991). Automatic abduction of qualitative models. In *Proc. Fifth International Workshop on Qualitative Physics*. Austin, TX.

[Kuipers 1986]
Kuipers, B. (1986). Qualitative simulation. *Artificial Intelligence*, 29(3):289–338.

[Laird 1988]
Laird, P. (1988). *Learning from Good and Bad Data*. Kluwer, Boston, MA.

[Lavrač 1990]
Lavrač, N. (1990). *Principles of knowledge acquisition in expert systems*. PhD thesis, Faculty of Technical Sciences, University of Maribor, Maribor, Slovenia.

[Lavrač and Džeroski 1992a]
Lavrač, N. and Džeroski, S. (1992a). Background knowledge and declarative bias in inductive concept learning. In Jantke, K., editor, *Proc. Third International Workshop on Analogical and Inductive Inference*, pages 51–71. Springer, Berlin.

[Lavrač and Džeroski 1992b]
Lavrač, N. and Džeroski, S. (1992b). Inductive learning of relations from noisy examples. In Muggleton, S., editor, *Inductive Logic Programming*, pages 495–514. Academic Press, London.

[Lavrač et al. 1985]
Lavrač, N., Bratko, I., Mozetič, I., Čerček, B., Horvat, M., and Grad, A. (1985). KARDIO-E: An expert system for electrocardiographic diagnosis of cardiac arrhythmias. *Expert Systems*, 2(1):46–49.

[Lavrač et al. 1991a]
Lavrač, N., Džeroski, S., and Grobelnik, M. (1991a). Learning nonrecursive definitions of relations with LINUS. In *Proc. Fifth European Working Session on Learning*, pages 265–281. Springer, Berlin.

[Lavrač et al. 1991b]
Lavrač, N., Džeroski, S., Pirnat, V., and Križman, V. (1991b). Learning rules for early diagnosis of rheumatic diseases. In *Proc. Third Scandinavian Conference on Artificial Intelligence*, pages 138–149. IOS Press, Amsterdam.

[Lavrač et al. 1992a]
Lavrač, N., Cestnik, B., and Džeroski, S. (1992a). Search heuristics in empirical inductive logic programming. In *Proc. Workshop on Logical Approaches to Machine Learning, Tenth European Conference on Artificial Intelligence.* Vienna, Austria.

[Lavrač et al. 1992b]
Lavrač, N., Cestnik, B., and Džeroski, S. (1992b). Use of heuristics in empirical inductive logic programming. In *Proc. Second International Workshop on Inductive Logic Programming.* Tokyo, Japan. ICOT TM-1182.

[Lavrač et al. 1993]
Lavrač, N., Džeroski, S., Pirnat, V., and Križman, V. (1993). The utility of background knowledge in learning medical diagnostic rules. *Applied Artificial Intelligence*, 7:273–293.

[Li and Vitányi 1991]
Li, M. and Vitányi, P. (1991). Learning simple concepts under simple distributions. *SIAM Journal of Computing*, 20(5):911–935.

[Lloyd 1987]
Lloyd, J. (1987). *Foundations of Logic Programming.* Springer, Berlin, 2nd edition.

[Lloyd 1990]
Lloyd, J., editor (1990). *Computational Logic.* Springer, Berlin.

[Michalski 1969]
Michalski, R. S. (1969). On the quasi-minimal solution of the general covering problem. In *Proc. Fifth Int. Symposium on Information Processing, FCIP 69*, volume A3. Bled, Yugoslavia.

[Michalski 1974]
Michalski, R. S. (1974). VL1: Variable-valued logic system. In *Proc. Int. Symposium on Multiple-Valued Logic.* West Virginia University, Morgantown, WY.

[Michalski 1980]
Michalski, R. (1980). Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(4):349–361.

[Michalski 1983]
Michalski, R. (1983). A theory and methodology of inductive learning. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning: An Artificial Intelligence Approach*, volume I, pages 83–134. Tioga, Palo Alto, CA.

[Michalski and Tecuci 1991]
Michalski, R. and Tecuci, G. (1991). *Proc. First International Workshop on Multistrategy Learning.* George Mason University, Fairfax, VA.

[Michalski et al. 1986]
Michalski, R., Mozetič, I., Hong, J., and Lavrač, N. (1986). The multi-purpose incremental learning system AQ15 and its testing application on three medical domains. In *Proc. Fifth National Conference on Artificial Intelligence*, pages 1041–1045. Morgan Kaufmann, San Mateo, CA.

[Michie 1988]
Michie, D. (1988). Machine learning in the next five years. In *Proc. Third European Working Session on Learning*, pages 107–122. Pitman, London.

[Mingers 1989a]
Mingers, J. (1989a). An empirical comparison of pruning methods for decision tree induction. *Machine Learning*, 4(2):227–243.

[Mingers 1989b]
Mingers, J. (1989b). An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3(4):319–342.

[Mitchell 1982]
Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, 18(2):203–226.

[Mladenić and Karalič 1992]
Mladenić, D. and Karalič, A. (1992). Drug design by machine learning: modelling drug activity. In *Proc. Sixth ISSEK Workshop.* Jožef Stefan Institute, Ljubljana, Slovenia.

[Mooney and Richards 1992]
Mooney, R. and Richards, B. (1992). Automated debugging of logic programs via theory revision. In *Proc. Second International Workshop on Inductive Logic Programming.* Tokyo, Japan. ICOT TM-1182.

[Morales 1992]
Morales, E. (1992). *First-order induction of patterns in chess.* PhD thesis, Department of Computer Science, University of Strathclyde, Glasgow, Scotland.

[Morik 1989]
Morik, K. (1989). Sloppy modelling. In Morik, K., editor, *Knowledge Representation and Organization in Machine Learning.* Springer, Berlin.

[Morik 1991]
Morik, K. (1991). Balanced cooperative modelling. In *Proc. First International Workshop on Multistrategy Learning*, pages 65–80. George Mason University, Fairfax, VA.

[Morik et al. 1991]
Morik, K., Causse, K., and Boswell, R. (1991). A common knowledge

representation integrating learning tools. In *Proc. First International Workshop on Multistrategy Learning*, pages 81–96. George Mason University, Fairfax, VA.

[Mozetič 1984]
Mozetič, I. (1984). Qualitative model of the heart. Master's thesis, Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, Slovenia. In Slovenian.

[Mozetič 1985a]
Mozetič, I. (1985a). Compression of the ECG knowledge-base using the AQ inductive learning algorithms. Reports of Intelligent Systems Group UIUCDCS-F-85-943, Department of Computer Science, University of Illinois, Urbana Champaign, IL.

[Mozetič 1985b]
Mozetič, I. (1985b). NEWGEM: Program for learning from examples, technical documentation and user's guide. Reports of Intelligent Systems Group UIUCDCS-F-85-949, Department of Computer Science, University of Illinois, Urbana Champaign, IL.

[Mozetič 1987]
Mozetič, I. (1987). Learning of qualitative models. In Bratko, I. and Lavrač, N., editors, *Progress in Machine Learning*, pages 201–217. Sigma Press, Wilmslow, UK.

[Mozetič 1988]
Mozetič, I. (1988). The role of abstractions in learning qualitative models. In *Proc. Fourth International Workshop on Machine Learning*. Morgan Kaufmann, San Mateo, CA.

[Mozetič and Lavrač 1988]
Mozetič, I. and Lavrač, N. (1988). Incremental learning from examples in a logic-based formalism. In Brazdil, P., editor, *Proc. Workshop on Machine Learning, Meta-Reasoning and Logics*, pages 109–127. Sesimbra, Portugal.

[Mozetič et al. 1984]
Mozetič, I., Bratko, I., and Lavrač, N. (1984). The derivation of

medical knowledge from a qualitative model of the heart. In *Proc. ISSEK Workshop*. Bled, Slovenia.

[Muggleton 1987]
Muggleton, S. (1987). DUCE: An oracle-based approach to constructive induction. In *Proc. Tenth International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann, San Mateo, CA.

[Muggleton 1991]
Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, 8(4):295–318.

[Muggleton 1992a]
Muggleton, S., editor (1992a). *Inductive Logic Programming.* Academic Press, London.

[Muggleton 1992b]
Muggleton, S. (1992b). A theoretical framework for predicate invention. In *Proc. Second International Workshop on Inductive Logic Programming.* Tokyo, Japan. ICOT TM-1182.

[Muggleton and Buntine 1988]
Muggleton, S. and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proc. Fifth International Conference on Machine Learning*, pages 339–352. Morgan Kaufmann, San Mateo, CA.

[Muggleton and Feng 1990]
Muggleton, S. and Feng, C. (1990). Efficient induction of logic programs. In *Proc. First Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsha, Tokyo.

[Muggleton et al. 1989]
Muggleton, S., Bain, M., Hayes-Michie, J., and Michie, D. (1989). An experimental comparison of human and machine learning formalisms. In *Proc. Sixth International Workshop on Machine Learning*, pages 113–118. Morgan Kaufmann, San Mateo, CA.

[Muggleton et al. 1992a]
Muggleton, S., King, R., and Sternberg, M. (1992a). Protein secondary structure prediction using logic. In *Proc. Second International Workshop on Inductive Logic Programming*. Tokyo, Japan. ICOT TM-1182.

[Muggleton et al. 1992b]
Muggleton, S., Srinivasan, A., and Bain, M. (1992b). Compression, significance and accuracy. In *Proc. Ninth International Conference on Machine Learning*, pages 338–347. Morgan Kaufmann, San Mateo, CA.

[Niblett 1988]
Niblett, T. (1988). A study of generalisation in logic programs. In *Proc. Third European Working Session on Learning*, pages 131–138. Pitman, London.

[Niblett and Bratko 1986]
Niblett, T. and Bratko, I. (1986). Learning decision rules in noisy domains. In Bramer, M., editor, *Research and Development in Expert Systems III*, pages 24–25. Cambridge University Press, Cambridge.

[Pazzani and Kibler 1992]
Pazzani, M. and Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94.

[Pazzani et al. 1991]
Pazzani, M., Brunk, C., and Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. In *Proc. Eighth International Workshop on Machine Learning*, pages 432–436. Morgan Kaufmann, San Mateo, CA.

[Pearce 1988a]
Pearce, D. (1988a). The induction of fault diagnosis systems from qualitative models. Technical Report TIRM-88-029, The Turing Institute, Glasgow, Scotland.

[Pearce 1988b]
Pearce, D. (1988b). The induction of fault diagnosis systems from

qualitative models. In *Proc. Seventh National Conference on Artificial Intelligence*, pages 353–357. Morgan Kaufmann, San Mateo, CA.

[Piatetsky-Shapiro and Frawley 1991]
Piatetsky-Shapiro, G. and Frawley, W., editors (1991). *Knowledge discovery in databases.* AAAI Press/MIT Press, Menlo Park, CA/Cambridge, MA.

[Pirnat et al. 1989]
Pirnat, V., Kononenko, I., Janc, T., and Bratko, I. (1989). Medical analysis of automatically induced diagnostic rules. In *Proc. Second European Conference on Artificial Intelligence in Medicine*, pages 24–36. Springer, Berlin.

[Plotkin 1969]
Plotkin, G. (1969). A note on inductive generalization. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh.

[Quinlan 1979]
Quinlan, J. (1979). Discovering rules by induction from large collections of examples. In Michie, D., editor, *Expert Systems in the Microelectronic Age*, pages 168–201. Edinburgh University Press, Edinburgh.

[Quinlan 1986]
Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

[Quinlan 1987a]
Quinlan, J. (1987a). Generating production rules from decision trees. In *Proc. Tenth International Joint Conference on Artificial Intelligence*, pages 304–307. Morgan Kaufman, San Mateo, CA.

[Quinlan 1987b]
Quinlan, J. (1987b). Simplifying decision trees. *International Journal of Man-Machine Studies*, 27(3):221–234.

[Quinlan 1989]
Quinlan, J. (1989). Learning relations: Comparison of a symbolic and a connectionist approach. Technical report, Basser Department of Computer Science, University of Sydney, Sydney, Australia.

[Quinlan 1990]
Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, 5(3):239–266.

[Quinlan 1991]
Quinlan, J. (1991). Knowledge acquisition from structured data – using determinate literals to assist search. *IEEE Expert*, 6(6):32–37.

[Quinlan 1993]
Quinlan, J. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA.

[Reinke and Michalski 1988]
Reinke, R. E. and Michalski, R. S. (1988). Incremental learning of concept descriptions: A method and experimental results. In Hayes, J., Michie, D., and Richards, J., editors, *Machine Intelligence 11*, pages 435–454. Oxford University Press, Oxford.

[Richards and Mooney 1991]
Richards, B. and Mooney, R. (1991). First-order theory revision. In *Proc. Eighth International Workshop on Machine Learning*, pages 447–451. Morgan Kaufmann, San Mateo, CA.

[Richards et al. 1992]
Richards, B., Kraan, I., and Kuipers, B. (1992). Automatic abduction of qualitative models. In *Proc. Tenth National Conference on Artificial Intelligence.* Morgan Kaufmann, San Mateo, CA.

[Robinson 1965]
Robinson, J. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41.

[Roth et al. 1981]
Roth, B., Aig, E., Rauckman, B., Strelitz, J., Phillips, A.,

Pherone, R., Bushby, S., and Sigel, C. (1981). 2,4-diamino-5-benzylpyrimidines and analogues as antibacterial agents. 5. 3',5'-dimethoxy-4'-substituted-benzyl analogues of trimethoprim. *J. Med. Chem.*, 24:933–941.

[Roth et al. 1987]
Roth, B., Rauckman, B., Pherone, R., Baccanari, D., Champness, J., and Hyde, R. (1987). 2,4-diamino-5-benzylpyrimidines as antibacterial agents. 7. analysis of the effect of 3,5-dialkyl substituent size and shape on binding to four different dihydrofolate reductase enzymes. *J. Med. Chem.*, 30:348–356.

[Rouveirol 1990]
Rouveirol, C. (1990). Saturation: Postponing choices when inverting resolution. In *Proc. Ninth European Conference on Artificial Intelligence*, pages 557–562. Pitman, London.

[Rouveirol 1991]
Rouveirol, C. (1991). Completeness for inductive procedures. In *Proc. Eighth International Workshop on Machine Learning*, pages 452–456. Morgan Kaufmann, San Mateo, CA.

[Rouveirol 1992]
Rouveirol, C. (1992). Extensions of inversion of resolution applied to theory completion. In Muggleton, S., editor, *Inductive Logic Programming*, pages 63–92. Academic Press, London.

[Russell 1989]
Russell, S. (1989). *The Use of Knowledge in Analogy and Induction.* Pitman, London.

[Sammut and Banerji 1986]
Sammut, C. and Banerji, R. (1986). Learning concepts by asking questions. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning: An Artificial Intelligence Approach*, volume II, pages 167–191. Morgan Kaufmann, San Mateo, CA.

[Shanon and Weaver 1964]
Shanon, E. C. and Weaver, W. (1964). *The Mathematical Theory of Communications.* University of Illinois Press, Urbana, IL.

[Shapiro 1983]
Shapiro, E. (1983). *Algorithmic Program Debugging.* MIT Press, Cambridge, MA.

[Smyth et al. 1990]
Smyth, P., Goodman, R., and Higgins, C. (1990). A hybrid rule-based/Bayesian classifier. In *Proc. Ninth European Conference on Artificial Intelligence*, pages 610–615. Pitman, London.

[Solomonoff 1964]
Solomonoff, R. (1964). A formal theory of inductive inference. *Information and Control*, 7:1–22, 224–254.

[Srinivasan et al. 1992]
Srinivasan, A., Muggleton, S., and M.Bain (1992). Distinguishing exceptions from noise in non-monotonic learning. In *Proc. Second International Workshop on Inductive Logic Programming.* Tokyo, Japan. ICOT TM-1182.

[Sternberg et al. 1992]
Sternberg, M., Lewis, R., King, R., and Muggleton, S. (1992). Modelling the structure and function of enzymes by machine learning. *Faraday Discuss.*, 93.

[Tangkitvanich and Shimura 1992]
Tangkitvanich, S. and Shimura, M. (1992). Refining a relational theory with multiple faults in the concept and subconcepts. In *Proc. Ninth International Conference on Machine Learning*, pages 436–444. Morgan Kaufmann, San Mateo, CA.

[Tecuci and Kodratoff 1990]
Tecuci, G. and Kodratoff, Y. (1990). Apprenticeship learning in non-homogenuous domain theories. In Kodratoff, Y. and Michalski, R., editors, *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 514–551. Morgan Kaufmann, San Mateo, CA.

[Ullman 1988]
Ullman, J. (1988). *Principles of Database and Knowledge Base Systems*, volume I. Computer Science Press, Rockville, MA.

[Utgoff and Mitchell 1982]
Utgoff, P. and Mitchell, T. M. (1982). Acquisition of appropriate bias for inductive concept learning. In *Proc. National Conference on Artificial Intelligence*, pages 414–417. Morgan Kaufmann, Los Altos, CA.

[Valiant 1984]
Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

[Williams 1990]
Williams, B. (1990). Interaction-based invention: designing devices from first principles. In *Proc. Fourth International Workshop on Qualitative Physics.* Lugano, Switzerland.

[Winston 1975]
Winston, P. H. (1975). Learning structural descriptions from examples. In Winston, P. H., editor, *The Psychology of Computer Vision.* McGraw-Hill, New York.

[Wirth 1988]
Wirth, R. (1988). Learning by failure to prove. In *Proc. Third European Working Session on Learning*, pages 237–251. Pitman, London.

[Wirth 1989]
Wirth, R. (1989). Completing logic programs by inverse resolution. In *Proc. Fourth European Working Session on Learning*, pages 239–250. Pitman, London.

[Wnek et al. 1990]
Wnek, J., Sarma, J., Wahab, A. A., and Michalski, R. S. (1990). Comparing learning paradigms via diagrammatic visualization: A case study in single concept learning using symbolic, neural net and genetic algorithm methods. In *Proc. Fifth International Symposium on Methodologies for Intelligent Systems.* Knoxville, TN.

[Wrobel 1988]
Wrobel, S. (1988). Automatic representation adjustment in an observational discovery system. In *Proc. Third European Working Session on Learning*, pages 253–262. Pitman, London.

[Žitnik 1991]
Žitnik, K. (1991). Machine learning of qualitative models. Technical Report IJS-DP-6239, Jožef Stefan Institute, Ljubljana, Slovenia. In Slovenian.

Index