Ai miei genitori

Acknowledgements

First, I would like to thank professors Paola Mello and Evelina Lamma for their gentle guidance, keen advice and patience. Their support and incitement have been fundamental for overcoming the most difficult times.

I am grateful to professor Maurelio Boari for his uninterrupted encouragement, for creating a fertile environment for doing research and for having offered me the opportunity to work in it.

I would like to thank professor Antonis Kakas for the stimulating discussions we had on the topics of the integration of abduction and induction and for the enthusiasm he communicated me on the subject. Part of the work was done while I was visiting the University of Cyprus: my stay there was fruitful and enjoyable and I am grateful to Antonis Kakas for his hospitality.

I would like to thank professor Luis Moniz Pereira for the interest he showed in my work and for many interesting discussions on the topics of knowledge representation in learning that were very inspiring for me.

I would also like to thank all my friends at LIA, for having made more enjoyable my time at the department: Rosy Barruffi, Anna Ciampolini, Enrico Denti, Michela Milano, Andrea Omicini, Cesare Stefanelli and Franco Zambonelli.

Lastly, I would like to thank my parents Odette and Lamberto, my sister Daniela, my grandparents Odilla and Giuseppe for their love, support and faith in me.

Contents

Chapter 1

Introduction

Machine Learning is a research area whose aim is to build machines that are able to construct or modify representations of what is being experienced. One of the most important tasks in Machine Learning consists in inducing knowledge from examples and data. The current interest in Machine Learning is justified from two points of view. On one hand, the increasing diffusion of knowledge based systems calls for automated methods for the acquisition of knowledge, since this process has been recognized as one of the main bottlenecks in the development of knowledge based system. On the other hand, the task of Data Mining, or the extraction of useful information from large amounts of data, has recently received a lot of attention as the amount of data that are stored by organizations in databases and data warehouses is rapidly increasing. Some of the techniques that are studied in Machine Learning are particularly suitable for Data Mining.

Machine Learning has been applied with success to a wide variety of fields, including medical or technical diagnosis, engineering design, industrial process control or banking. As the number of fields where Machine Learning is applied increases, more and more complex domains are considered and more and more expressive representation languages are used to represent such domains. They have evolved from numerical descriptions to attribute-value languages and, finally, to first-order logic ones. Increasing the expressiveness of the representation language is a general trend in Machine Learning that has allowed to solve problems that arise in new application domains.

Recently, the language of Logic Programming has been extensively used in learning. The adoption of this language is studied in the field of Inductive Logic Programming (ILP henceforth). The language of logic programs has allowed objects in the domain to be described in a structured way, i.e., in terms of their components and relations among the components. The given relations constitute the background knowledge that is given as input to the learner together with the examples of the target relation. Such an expressive representation language has allowed ILP to tackle complex problems in various domains, including dynamic systems, molecular biology, mechanical engineering, natural language processing and software engineering.

The contribution of this thesis is twofold. First, we identify two problems of existing ILP techniques: their limited ability to learn from an incomplete background knowledge and the use of a two-valued logic that does not allow to consider some pieces of information as unknown. Second, we overcome these limits by prosecuting the general trend of increasing the expressiveness of the representation language. Two learning systems have been developed that represent knowledge using two extensions of Logic Programming, namely abductive logic programs and extended logic programs.

1.1 Limits of ILP

In the following, we briefly describe the problems that have been considered in this thesis.

The first problem concerns the fact that the acquisition of data from real world is often imperfect. The data acquired from the real world is often noisy and/or incomplete. We will consider the problem of learning from incompleteness in the background knowledge. Information about individual instances representing examples is usually expressed by means of ground facts in the background: if the acquisition of information for some instances was incomplete, then some background facts will be missing. In this case, some positive examples may not be covered due to the absence of some facts related to them in the background. This may require the learning of multiple overspecific rules for covering a set of examples that could otherwise be covered by a single general one.

Various systems have been developed to learn from imperfect data (for example, FOIL [Qui90a], mFOIL [Dže91], FOIL-I [IKI$^+$96] and LINUS [LDG91b]). However, no system has been specially designed for learning from an incomplete background knowledge.This problem can be solved by integrating abductive reasoning into induction: abduction is used in order to complete the background knowledge by making assumptions about the incomplete background predicates.

Another problem concerns the fact that most work on ILP and inductive concept learning in general has considered a two-valued logical setting. However, in some learning problems is useful to consider a three-valued logical setting. For example, this is the case of an autonomous agent that gathers information from its surrounding world by performing experiments and memorizing the results. Such an agent needs to store both positive information, about successful experiments, and negative information, about unsuccessful experiment, and learn from both positive and negative information. For example, consider the case of an agent that has to learn general rules about the effect of actions in a certain domain, with respect to its goal. The agent will try actions and see whether the action has had a positive or negative result. It will then use the results, either positive or negative, for learning a general description of actions that it will use for planning its behaviour. For such an agent, it is important to learn a description of actions that distinguishes among actions with a positive outcome, actions with a negative outcome and actions with an unknown outcome. In this way, it will be able to make decisions on what actions to perform knowing exactly the possible consequences. It may thus decide to perform an action with a negative outcome if it thinks it is necessary or try an action with an unknown outcome in order to further explore its domain.

This type of learning requires the adoption of a three-valued logical setting, where sentences can have the truth value true, false or unknown. However, most work on inductive concept learning considers a two-valued setting, where what is not entailed by the learned theory is considered as false, on the basis of the Closed World Assumption [Rei78]. In a three-valued setting one is able to learn and represent a definition for both the target concept and its opposite and to resolve the contradiction between the definitions by assigning the truth value unknown to conflicting atoms.

1.2 Proposed Solutions

Various extensions of the language of logic programs have been proposed in order to improve its expressive power. This thesis proposes the adoption of two extensions of Logic Programming as the representation language for learning in order to solve the above mentioned problems.

Abductive logic programs provide an effective mechanism for representing and reasoning with incomplete information. They allow hypothetic reasoning to be performed: assumptions can be made about a number of predicates, called abducibles, for which a definition is absent or is incomplete. Integrity constraints can be used in order to reduce the number of assumptions that are allowed. Thus, abductive logic programs consist of a logic program, a set of abducible predicates and a set of integrity constraints. By representing the background knowledge as an abductive logic program, we are able to exploit the reasoning mechanism of abduction for completing the knowledge during learning. We have designed and implemented the system ACL (Abductive Concept Learning) that learns from a background knowledge in the form of an abductive logic program: examples can be covered by making assumptions about some missing facts in the background. The system is able to learn new rules and new constraints: the theory that is learned can thus be used to classify new unseen examples that are incompletely specified. The system has been tested on a number of datasets where the knowledge is incomplete and the results obtained have been compared with those of state of the art systems like ICL-Sat [DRD96c], mFOIL [Dže91] and FOIL [Qui90a]. The performances of ACL were found to be superior or comparable with those of these systems on the considered datasets.

By means of *Extended logic programs* we are able to represent and reason with information in a three-valued logical setting. Extended logic programs contain two kinds of negation: default negation plus a second form of negation called *explicit*, that is used in order to explicitly represent negative information. By adopting extended logic programs as a representation language for learning we are able to learn a definition for both the target concept and its opposite. Special techniques have to be adopted to ensure the consistency among the definitions for the concept and its opposite: in case both the definitions cover an unseen example, the example is classified as unknown. Explicit negation is used in order to represent the opposite concept, while default negation is used in order to represent exceptions to definitions by means of default rules. We have developed the system LIVE (Learning In a three-Valued Environment) that learns definitions for both the concept and its opposite that may allow exceptions. The system is able to learn a definition for the exceptions that, on its turn, may also allow exceptions. In this way hierarchies of exceptions can be learned. The system is parametric in the technique adopted for learning the definitions for the concept and its opposite: by means of bottom-up techniques we find least general definitions, while by means of top-down techniques we find most general definitions. The possibility of choosing independently the generality of the two definitions is useful in domains where we need to take into account the risk of making a mistake in classifying erroneously an unseen instance. Various experiments have been performed to show the ability of the system to combine solutions of different generality, to show its ability to deal with contradiction and to show how hierarchies of exceptions can be learned.

1.3 Structure of the Thesis

The thesis is organized as follows. In chapter 2, we will provide an overview of Machine Learning, presenting the various learning strategies and paradigms that have been investigated in the field. Then, we concentrate on the learning strategy of inductive learning from examples by means of the paradigm of symbolic concept acquisition. The evolution of representation languages used for this task is described, going from analytical expressions, to attribute-value descriptions to Logic Programming.

Chapter 3 is devoted to presenting the problems and techniques that are studied in the field of ILP. First, some preliminaries about logic programming are given: the syntax and semantics of the language is defined. Two main learning settings exist in ILP: learning from entailment and learning from interpretations. For each of them, the learning problem is defined and examples of problems are given. Learning from entailment has been most extensively studied and is described in more details: we will discuss the properties of soundness and completeness for a learning algorithm, the criteria for classifying ILP systems, the various types of imperfections that can appear in the data, the structure of the hypothesis space and, finally, the techniques that can be adopted for learning. Some of the most representative ILP systems are then described: GOLEM [MF90], FOIL [Qui90a], mFOIL [Dže91] and ICL [DRL95].

Chapter 4 considers the problem of learning from incomplete information in the background and describes the adoption of abductive logic programs as the representation formalism. First, abductive logic programs are defined, together with a semantics and a proof procedure for them. A learning problem in this new setting is then presented that is called Abductive Concept Learning (ACL). The ACL problem can be split into two subproblems that consist of learning the program part and learning the constraint part. A system, also called ACL, is proposed that solves the problem by solving the two subproblems in sequence. ACL can be very useful as well for solving problems of multiple predicate learning: the main issues involved in these types of learning problems are discussed and an extension of ACL, called M-ACL, is presented for performing this task. Two series of experiments are then presented, the first that show the ability of ACL to learn from datasets with an incomplete background knowledge and the second to show the ability of ACL to learn multiple predicates. Finally, other works that integrates abduction and induction are discussed.

Chapter 5 discusses the problem of learning in a three-valued setting. First, we show the usefulness of a three-valued logical setting and of two types of negation for knowledge representation. Then, we provide a definition of extended logic programs and recall the *WFSX* semantic for them together with the sound proof procedure *SLX*. The utility of the introduction of a three-valued logical setting in learning is presented next, together with the definition of the new learning problem adopting extended logic programs as the representation language. Depending on the technique used to learn a definition for the concept and its opposite, we may learn a least general definition or a most general definition: we discuss the criteria that should be adopted for choosing the generality levels according to the learning conditions. Then, the issue of contradiction is presented and techniques for resolving it are described. We consider first the case in which contradiction arise in a single source of information and then the case of multiple conflicting sources. Depending on the generality of the definitions, different types of contradiction can be distinguished and different approaches for the revision of the definitions are described. Finally, the system LIVE for learning extended logic program is presented and related works are discussed.

Chapter 6 presents the conclusions of the thesis. We first recall the aim of the thesis and then we summarize the results obtained. We end by presenting directions for future works.

Chapter 2

An Overview of Machine Learning

The research area of Machine Learning includes a wide variety of different approaches. In this chapter, a brief overview of the spectrum of learning paradigms will be presented. Then we will concentrate on the problem of inductive concept learning from examples, which is the paradigm adopted in this thesis. Solving this problem by means of symbolic techniques requires the adoption of a representation language for the examples and for the concepts to be learned. In the last section of this chapter we will describe the evolution that the representation languages have undergone from early studies in Machine Learning to the more recent research in Inductive Logic Programming.

2.1 Machine Learning

Various definitions of learning have been proposed in the literature. Two main views exist, which are complementary to each other. The first view is due to H. A. Simon that has given the following definition ([Sim83], pag. 28):

> Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.

However, some learning tasks are only concerned with acquiring new knowledge, without improving the performance of any systems. Therefore, another definition was proposed by Michalski [MCM86]:

> Learning is constructing or modifying representations of what is being experienced.

The current interest in Machine Learning can be understood in the light of these two definitions. According to the first definition, Machine Learning is interesting since it aims at the engineering task of building machines that are able to modify themselves in order to perform better at a given task, following the definition of learning given by Simon.

According to the second definition, Machine Learning can be used to acquire new knowledge that can be used by humans, by machines or by both. It has been generally accepted that the main bottleneck in building knowledge based systems consists in the acquisition of knowledge. Therefore, having methods that can simplify this task is very important.

Machine Learning is also interesting for another reason. Even if there is no consensus on the definition of intelligence, there is agreement on the fact that the ability to learn is one of the key features of intelligent behaviour. Therefore Machine Learning is also important from a cognitive science point of view because it can help to improve our understanding of the mechanisms underlying learning in humans.

Various criteria have been proposed for classifying Machine Learning research [CMM83, Mic86]. We will consider here the learning strategy criterion, as suggested by [Mic86], and the research paradigm, as suggested by [Car89].

The learning strategy refers to the type of inferences performed by the system during learning, while the research paradigm refers to the approach and techniques used in the construction of the system.

2.1.1 Learning Strategies

Learning can be seen as a process where the learner transforms information provided by a teacher (or environment) into an internal form that is stored for future use. The learning strategy employed by the learner consists in the type of this transformation. Several different strategies have been identified: *rote learning, learning by instruction, learning by deduction, learning by analogy* and *learning by induction*. The latter subdivides into *learning from examples* and *learning by observation and discovery*. The strategies are listed in order of increasing complexity of the transformation performed on the knowledge.

In *rote learning*, the information from the teacher is directly memorized by the learner without undergoing any elaboration. In this type of learning, the issue is how to index stored knowledge for future retrieval. In *learning by instruction*, the learner acquires knowledge from a teacher or another organized source, such as a textbook. The learner has to simply transform the knowledge from the input language to an internally-usable representation. In *deductive learning*, the learner performs deductive inferences on the knowledge provided by the teacher and stores useful conclusions in order to obtain a more efficient and/or comprehensible theory.

Learning by analogy consists in obtaining knowledge applicable in the current situation from knowledge about past situations that bears strong similarities with the current situation. The aim of the transformation is to modify the available information so that it becomes useful in the current situation.

Induction is generally understood as reasoning from specific to general. In *inductive learning*, the learner starts from the facts and observations provided by a teacher or the environment and generalizes them, obtaining knowledge that should be valid also for cases not yet observed. Inductive learning can be subdivided into *learning from examples* and *learning by observation and discovery*. In learning from examples, the teacher provides a set of positive examples that are instances of a concept and a set of negative examples that are non-instances of the concept. The task of the learner is to build a general description that describes all the positive examples and none of the negative. In learning from observation and discovery, the task of the learner is to find regularities and general rules that hold on the observations. The observations may contain instances of multiple concepts and the leaner has to discover relations among them, rather than definitions for the individual concepts. Examples of this form of learning are conceptual clustering (grouping objects that exhibit similar properties into classes), discovering laws explaining a set of observations and formulating theories accounting for the behaviour of a system. When the result of learning must be a theory describing one or more concept in a form that is understandable by humans, we speak about *inductive concept learning from examples* and *inductive concept learning from observations*.

2.1.2 Research Paradigm

The research paradigm refers to the type of techniques used for the construction of learning systems. The different techniques used in Machine Learning can be broadly classified into four categories: *symbolic concept acquisition, analytic (or deductive) learning, evolutionary learning (or learning with genetic algorithms)* and *connectionist learning (learning with neural nets)*[1].

In *symbolic concept acquisition*, the task of the system consists in building a symbolic representation of a given set of concepts by processing a set of examples and counter-examples of these concepts. The languages used for the representation of the concepts can be first order logic, decision trees, production rules or semantic networks. Learning is usually performed by searching the space of concept descriptions.

In *analytic (or deductive) learning*, deduction is used in order to generate conclusions from the available knowledge that allow a more efficient application of domain knowledge in new cases. The aim of analytic methods is not to extend the set of available concept descriptions but to improve the efficiency of the system.

Evolutionary learning is inspired to the theory of Darwinian evolution, where a population of individuals evolve by means of gene transformation in biological reproduction (cross-over, mutations, etc.) and by survival of the fittest. In learning, the concept descriptions are the individuals and they are combined and modified by means of biological-like operators in order to obtain new concept descriptions that are then selected according to a fitness function.

In *connectionist learning* or *learning with neural nets*, the learning system is composed by a number of interconnected elements, usually neuron-like, that perform some simple logical function, typically a threshold logic function. Each connection between two elements is assigned a weight that represents the strength of the connection. The neural net will have some input connections, that come from the external world, and some output connections, that go towards the external world. The learning is performed by incrementally modifying the connection weights in order to minimize the error on the set of input-output couples provided as training examples.

Apart from the strategies of rote learning and learning by instruction, that adopt ad-hoc learning techniques, a mapping can be established among learning strategies and the above research paradigms.

Deductive learning can be performed by means of analytic (or deductive) learning techniques. Learning by analogy can be performed by combining techniques of symbolic concept acquisition and deductive learning or by connectionist learning. Inductive learning from

[1] This classification is adapted from the one proposed in [Car89] by substituting the term inductive learning with symbolic concept acquisition

examples or observations can be performed by symbolic concept acquisition, connectionist learning or evolutionary learning. However, the task of inductive concept learning can be performed only by means of symbolic concept acquisition techniques or by evolutionary learning.

In this thesis we will consider learning systems that adopt the learning strategy of inductive concept learning from examples with the research paradigm of symbolic concept acquisition.

2.2 Inductive Concept Learning from Examples

In this section, we will give a definition of inductive concept learning from examples. We will follow the definition provided in [LD94]. Learning is performed on a domain that is described by the *universe* of all the objects in the domain, represented by the set U. A *concept* C is defined as a subset of the universe: $C \subseteq U$. To learn a concept C means to learn a concept description that allows to recognize if an object belongs to C, i.e. if $x \in C$ for any $x \in U$.

In order to test the membership of an object to a concept, we need a language for describing objects, a language for describing concepts and a procedure that interprets the languages and performs the test. When the description of an object satisfies the description of a concept, we say that the concept description *covers* the object description. Therefore, in order to define an inductive concept learning problem, we need to define an *object description language*, L_o, a *concept description language*, L_c, and a procedure for testing the coverage.

We will call *fact* the description of an object and *hypothesis* the description of a concept to be learned. Learning is performed starting from a set of *examples* that are facts for which the concept membership is known. Examples are therefore *labeled* facts, for which the label represents the concept membership. The set of all the examples is called *training set* and is denoted with E.

In *single concept learning*, the labels are $\oplus$ and $\ominus$ and they indicate whether the object belongs or does not belong to the concept we want to learn. If an object belongs to a concept, we also say that it is an *instance* of the concept. Examples from E labeled $\oplus$ are called *positive examples* and form the set E^+, while examples labeled $\ominus$, are called *negative examples* and form the set E^-. Sometimes we will consider sets E^+ and E^- that contain unlabeled facts: positive examples are distinguished implicitly from negative examples from their membership to either E^+ or E^-. In this case, we will call examples also the unlabeled facts from E^+ and E^-.

In *multiple concept learning*, labels denote positive and negative examples relative to different concepts and the training set can be divided into subsets of positive and negative examples each corresponding to a concept.

The problem of inductive learning of a single concept C from examples can be stated as follows:

Definition 1 (Inductive Concept Learning) *Given a set E of positive and negative examples of a concept C described in a given object description language L_o, find a hypothesis H, expressed in a given concept description language L_c, such that*

- *every positive example $e^+ \in E^+$ is covered by H,*

- *no negative example $e^- \in E^-$ is covered by H.*

In order to test the coverage of a hypothesis H, a function

$$covers(H, e) \tag{2.1}$$

can be defined that returns true if the example e is covered by H and returns false otherwise. The implementation of this function depends on the languages L_o and L_c.

We also define the function $covers(H, E)$ that returns the set of examples in E that are covered by H

$$covers(H, E) = \{e \in E \mid covers(H, e) = true\} \tag{2.2}$$

In order for H to be a solution of a learning problem, it must cover all positive examples and none of the negative ones. When a hypothesis H covers all the positive examples, we say that the hypothesis is *complete*, while when H does not cover any negative example we say that it is *consistent*. By means of the function $covers(H, E)$, the notions of completeness and consistency can be defined as follows. A hypothesis H is *complete* if $covers(H, E^+) = E^+$ and is *consistent* if $covers(H, E^-) = \emptyset$.

In many cases, it is useful for a learner to exploit, besides examples, available knowledge on the domain. The knowledge that is available to the learner "a priori" is called *background knowledge* and it is usually expressed in the language L_c. By using background knowledge, a learner can express more naturally and more concisely the hypothesis to be learned, thus simplifying the learning task. In practice, difficult learning problems require a substantial amount of background knowledge to be solved effectively. When a background knowledge is available, the learning problem must be restated. The coverage test has to be modified so that the background knowledge is taken into account when the membership of an example to a concept is verified. Let B denote the background knowledge, the coverage functions 2.1 and 2.2 now become

$$covers(H, B, e) \tag{2.3}$$

$$covers(H, B, E) \tag{2.4}$$

The definition of the learning problem can now be restated as follows:

Definition 2 (Inductive Concept Learning with Background Knowledge) *Given a set E of positive and negative examples of a concept C described in a given object description language L_o and a background knowledge B expressed in the concept description language L_c, find a hypothesis H, expressed in a given concept description language L_c, such that H is complete and consistent with respect to the examples E.*

The definition of consistency and completeness must also be restated to take into account the background knowledge:

Definition 3 (Completeness) *A hypothesis H is* complete *with respect to background knowledge B and examples E if all the positive examples in E are covered, i.e., if $covers(H, B, E^+) = E^+$*

Definition 4 (Consistency) *A hypothesis H is* consistent *with respect to background knowledge B and examples E if no negative example in E is covered, i.e., if $covers(H, B, E^-) = \emptyset$*

The problem of concept learning can be seen as a problem of search in the space of concept descriptions [Mit82], also called *hypothesis space.* For non trivial concept description languages, the search space is extremely large and additional mechanisms are required to make the search feasible. Any mechanism employed by a learning system to constrain the search for hypothesis is called *bias.* When the bias is a modifiable parameter of the system that the user has to explicitly specify for each learning problem it is called *declarative bias.* There are two types of bias: the *search bias*, that determines the way the hypothesis space is searched, and the *language bias*, that determines the hypothesis space itself.

2.3 Representation Languages in Inductive Reasoning

The representation languages that have been used for performing inductive reasoning have undergone a substantial evolution from Pattern Recognition to early studies in Machine Learning, to recent works on learning relational concepts. The language has evolved from concept descriptions that are essentially numeric, to attribute-value languages, to relational languages, to first-order logical languages. The main reason for this evolution was to increase the expressivity of the language in order to be able to represent and to learn more and more complex concepts.

In the research area of Pattern Recognition the language of examples is represented by the values that a number of variables assume for each object of the domain. Each variable is usually measured at least on an interval scale. The language of concepts is represented by analytical expressions involving the numerical variables describing the objects and a number of numeric parameters that, when instantiated, determine the expression. The task of inductive learning is, in this case, the fine-tuning of these numeric parameters. An example [BG95] is the problem of finding the linear discriminant of a set of points in an n-dimensional space belonging to two classes (or to one class and not to the class). The goal is to find a linear discriminant of the instances of the two classes, consisting of an hyperplane: the parameters in the hyperplane equation must be determined in the training phase.

However, in many cases it is not possible or not convenient to use a set of numerical variables as the representation language [Mic80]. Often, the relevant object characteristics are not numerical but categorical. In this case, the use of numerical techniques to treat symbolic features is inefficient and inadequate [BG95]. Moreover, representing symbolic features numerically makes the resulting description poorly comprehensible to humans. In order to solve these problems, attribute-value languages are used.

In attribute-value languages objects are described by a fixed set of variables called *attributes* that can assume values from predefined sets. For example, consider the problem of deciding whether a Saturday morning is suitable for playing tennis [Mit97] on the basis of the weather conditions. This problem can be described by means of four attributes *Outlook, Temperature, Humidity* and *Wind* that can assume a value, respectively, in the sets {*Sunny, Overcast, Rain*}, {*Cold, Medium, Hot*}, {*Low, High*} and {*Weak, Strong*}. Each Saturday morning is described by a list of attribute value pairs, for example

Outlook = Sunny, Temperature = Hot, Humidity = Normal, Wind = Strong

or as a tuple of values

⟨*Sunny, Hot, Normal, Strong*⟩

On this particular Saturday morning one would play tennis, therefore this is a positive example for the concept *PlayTennis.* In attribute-value languages, concepts are described

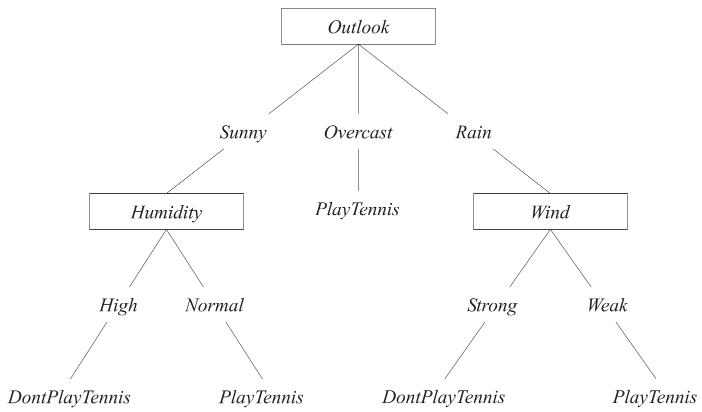

Figure 2.1: A decision tree for the concept *PlayTennis* (Taken from [Mit97])

by means of rules where the antecedent consists of conjunctions or disjunctions of attribute-value pairs (expressions of the form $Attribute = Value$) and the consequent is a concept name. No variables, quantifiers and relations among components of examples are allowed in the description of concepts. For example, a description of the concept of a Saturday can be represented as the following rule:

$$\begin{array}{ll} \textit{PlayTennis}\ \textbf{if} & \textit{Outlook=Sunny} \wedge \textit{Humidity=Normal} \\ & \vee\ \textit{Outlook=Overcast} \\ & \vee\ \textit{Outlook=Rain} \wedge \textit{Wind=Weak} \end{array}$$

Concept descriptions in attribute-value languages may also be described by means of *decision trees*. Each node of a decision tree corresponds to a test on an attribute and each branch corresponds to one of the possible values for the attribute. The leaves of the tree are labeled with one of the concepts to be learned (or with the concept and its opposite in single concept learning). An instance is classified by starting from the root node of the tree, testing the attribute on that branch and then moving down the branch corresponding to the value of the attribute in the given instance. This process is repeated until one of the leaves is reached: the classification of the instance is given by the label of the leaf. In figure 2.1 it is shown the decision tree for the concept *PlayTennis* corresponding to the rule above.

Attribute-value languages are equivalent to propositional logic. Examples of systems that adopt an attribute-value representation are AQ [Mic73], CN2 [CB89] and c4.5 [Qui93].

Attribute-value languages have two main drawbacks: they have a limited expressive power and it is difficult to use the available background information with them. The lack of possibility of expressing relations among components of the examples is particularly important when examples are complex objects that can be decomposed into various components with different relations among them. Therefore, *relational languages* were introduced that allow the representation of structured objects in terms of their components and relations among the components. Such languages are usually equivalent to a subset of first-order logic.

Some authors [Mic80, BGS88] have adopted a relational frame-like language, where an object is represented by dividing it into components and, for each component, by giving the list of values for the attributes of that component. Some attributes may be relevant only for some of the components, therefore the list of attribute-value couples for the components has a variable length. For example [BG95], an instance of a family including a grandfather can be expressed as

$$
\begin{array}{lll}
Name = david, & Son = mike, & Father = ron \\
Name = mike, & Son = junior, & Father = david \\
Name = junior, & Father = mike &
\end{array}
$$

Structured objects of this kind could be represented as well with a unique list of attribute-value pairs, by indexing the attributes with the component name. However, this list would contain as many attributes as are required by the most complex objects. If the complexity of objects is uneven, this results in a waste of space and lower comprehensibility for humans.

The concept descriptions allowed by relational languages of this kind may contain quantifiers and variables. For example, the concept of a family including a grandfather can be expressed as

$$\exists X, Y, Z \ son(X) = Y \wedge son(Y) = Z$$

Another language where relations can be expressed is *Logic Programming* [Llo87]. The research area that studies learning system adopting Logic Programming as a representation language is called Inductive Logic Programming (ILP henceforth). Logic Programming has three advantages with respect to the relational languages of the previous form [BG95]. First of all, it allows recursion in the definition of concepts, thus making possible to express a wider class of concepts. Second, the notation of logic programming is simpler, more standardized and interpreters for it are based on sound and well-understood theoretical grounds. This has brought a clarification and a more rigorous formalization in learning. Third, logic programming is also a programming language, therefore the definitions that are learned can also be interpreted as executable programs, thus providing an approach for the automated development of programs.

In Logic Programming, examples are described as ground literals, i.e., predicates applied to constant arguments, while the knowledge about the relations among the components and their attributes is expressed in the background knowledge. The previous example of a family with a grandfather, is now expressed as the fact

$grandfather(david, mike) \leftarrow$

together with a number of facts in the background knowledge

$father(ron, david) \leftarrow$

$father(david, mike) \leftarrow$

$father(mike, junior) \leftarrow$

Moreover, some background knowledge may be available that can be used by the learning system. For example, we may know that

$child(X, Y) \leftarrow father(Y, X)$

The concept will be represented by a predicate and the concept description will be a logic program. For the grandfather example, it is possible to distinguish three different definitions of grandfather [BG95]

$grandfather \leftarrow father(X, Z), father(Z, Y)$

$grandfather(X) \leftarrow father(X, Z), father(Z, Y)$

$grandfather(X,Y) \leftarrow father(X,Z), father(Z,Y)$
The first definition identifies the classes of families containing at least one grandfather, the second defines the requirements for being a grandfather and the last states the conditions that a specific person X must satisfy in order to be the grandfather of Y. With previous relational learning languages these concepts were hard to distinguish and have sometimes been confused: this example shows the clarification issue that has been addressed by Logic Programming. As regards the issue of increased expressivity thanks to recursion, Logic Programming allows definition of concepts of the form
$ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,Y)$
that were not allowed by none of the previous approaches in Machine Learning.

In the next chapter the concept and techniques from the research field of ILP will be presented in details.

Chapter 3

Inductive Logic Programming

Inductive Logic Programming (ILP) is the research field that studies the problem of inductive concept learning from examples when the representation language employed is Logic Programming. This chapter introduces the terminology of logic and Logic Programming and provides an overview of the problems and techniques that have been studied in the field of Inductive Logic Programming.

Two main formalizations of the learning problem have been given: *learning from entailment* and *learning from interpretations* [DR97]. The two settings differ in the definition of the coverage relation and in the form of examples: in learning from entailment examples are ground facts, while in learning from interpretations examples are Herbrand interpretations, i.e., sets of ground facts.

Learning from entailment is the most widely used problem setting and will be treated in more details in section 3.2. The problem of learning from interpretations will be described in section 3.3.

3.1 Logic Programming Preliminaries

In this section, we will give some basic notions on first order logic languages and Logic Programming (adapted from [Llo87] and [Fla95]) that will be used throughout the thesis.

A *first order logic language* L is defined by an alphabet that consists of seven sets of symbols: variables, constants, functions symbols, predicate symbols, logical connectives, quantifiers and punctuation symbols. The first four classes differ from language to language, while the last three are the same for all the languages. The connectives are $\sim$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\leftarrow$ (implication), $\leftrightarrow$ (equivalence); the quantifiers are the existential quantifier $\exists$ and the universal quantifier $\forall$, and the punctuation symbols are "(", ")", and ",".

Well-formed formulas (wff's) of the language are the syntactically correct clauses of the language and are inductively defined by combining elementary formulas, called *atomic formulas*, by means of logical connectives and quantifiers. On their turn, atomic formulas are obtained by applying the predicates symbols to elementary terms.

A *term* is defined recursively as follows: a variable is a term, a constant is a term, if f is a function symbol with arity n and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term. An *atomic formula* or *atom* is the application of a predicate symbol p with arity n to n terms: $p(t_1, \ldots, t_n)$.

A *well-formed formula* is defined recursively as follows:

- every atom is a wff;
- if A and B are wff's, then also $\sim A$, $A \wedge B$, $A \vee B$, $A \leftarrow B$, $A \leftrightarrow B$ are wff's (possibly enclosed in balanced brackets);
- if A is wff and X is a variable, $\forall X\ A$ and $\exists X\ A$ are wff.

The *scope* of a quantifier $\forall X$ (resp. $\exists X$) in $\forall X\ F$ (resp. $\exists X\ F$) is F. An occourence of a variable in a quantifier is *bound* if it immediately follows a quantifier, or if it occours in the scope of a quantifier with the same variable. Any other occourence of a variable in a formula is *free*. A *closed* formula is a formula without free occourences of any variables, otherwise the formula is *open*. For any formula, $\forall(F)$ denotes the *universal closure* of F, which is the closed formula obtained by adding a universal quantifier for each variable with a free occourence in F. A *variant* ϕ' of a formula ϕ is obtained by renaming all its variables.

The class of formulae called *clauses* has important properties. A *clause* is a formula of the form

$$\forall X_1 \forall X_2 \ldots \forall X_s (A_1 \vee \ldots \vee A_n \vee \sim B_1 \vee \ldots \vee \sim B_m)$$

where each A_i, B_i are atoms and $X_1, X_2, \ldots, X_s$ are all the variables occouring in $(A_1 \vee \ldots \vee A_n \vee \sim B_1 \vee \ldots \vee \sim B_m)$. The clause above can also be represented as follows:

$$A_1; \ldots; A_n \leftarrow B_1, \ldots, B_m$$

The part preceeding the symbol $\leftarrow$ is called the *head* of the clause, while the part following it is called the *body*. An atom or the negation of an atom is called a *literal*. A *positive literal* is an atom, a *negative literal* is the negation of an atom. Sometimes clauses will be represented by means of a set of literals:

$$\{A_1, \ldots, A_n, \sim B_1, \ldots, \sim B_m\}$$

A clause is a *denial* if it has no positive literal, *definite* if it has one positive literal, and *indefinite* if it has more than one positive literal. A *Horn clause* is either a definite clause or a denial. A *fact* is a definite clause without negative literals, sometimes the $\leftarrow$ symbol will be omitted for facts. A clause C is *range-restricted* if and only if the variables appearing in the head are a subset of those in the body. A *normal clause* is a clause of the form

$$A \leftarrow B_1, \ldots, B_i, not\ B_{i+1}, \ldots, B_m$$

where *not* denotes a kind of negation that is different with respect to $\sim$. A *definite logic program* is a set (conjunction) of definite clauses. A *normal logic program* is a set (conjunction) of definite and normal clauses.

The following notation for the symbols will be adopted: predicates, functions and constants start with a lowercase letter, while variable symbols start with an uppercase letter (as in the Prolog programming language, see below). A *functor* is a function symbol occouring in a clause. A *substitution* $\theta = \{X_1/t_1, \ldots, X_k/t_k\}$ is a function mapping variables to terms.

The *application* $C\theta$ of a substitution θ to a clause C means replacing all the occourences of each variable X_j in C by same term t_j.

A *ground* clause (term) is a clause (term) without variables. The *Herbrand universe* H of a language or a program is the set of all the ground terms that can be obtained combining the symbols in the language or program. The *Herbrand base* B of a language or a program is the set of ground atoms. Sometimes they will be indicated with $H(P)$ and $B(P)$ where P is the program.

The semantics of a set of formulas can be defined in terms of interpretations and models.We will here consider the special case of *Herbrand interpretations* and *Herbrand models* that are sufficient for giving a semantics to sets of clauses, both definite and indefinite. For a definition of interpretations and models in the general case see [Llo87]. A *Herbrand interpretation* I is a subset of the Herbrand base, i.e., $I \subseteq B$. Given a Herbrand interpretation, it is possible to assign a truth-value to a formula according to the following rules. A ground atom $p(t_1, t_2, \ldots, t_n)$ is true under the interpretation I if and only if $p(t_1, t_2, \ldots, t_n) \in I$. A conjunction of atomic formulas $B_1, \ldots, B_m$ is true in I if and only if $B_1, \ldots, B_m \subseteq I$. A ground clause $\{A_1, \ldots, A_n, \sim B_1, \ldots, \sim B_m\}$ is true in an interpretation I if and only if at least one of the atoms of the head is true in the case in which the body is true. A clause C is true in an interpretation I if and only if all its ground instances with terms from H are true in I. A set of clauses Σ is true in an interpretation I if and only if all the clauses $C \in \Sigma$ are true.

An interpretation I *satisfies* a set of clauses Σ, notation $I \models \Sigma$, if Σ is true in I; we also say that I is a *model* of Σ. A set of clause is *satisfiable* if it is satisfied by some interpretation, *unsatisfiable* otherwise. If all models of a set of clauses Σ are also models of a clause C, we say that Σ *logically entails* C or C is a *logical consequence* of Σ, and we write $\Sigma \models C$[1].

Herbrand interpretations and models are sufficient for giving a semantics to sets of clauses in the following sense: a set of clauses is unsatisfiable if and only if it does not have a Herbrand model. For sets of definite clauses Herbrand models are particularly important because they have the relevant property that the intersection of a set of Herbrand models for a set of definite clauses P is still an Herbrand model of P. The intersection of all the Herbrand models of P is called the *minimal Herbrand model* of P and is represented with $lhm(P)$. The least Herbrand model of P always exists and is unique. The *model-theoretic semantics* of a program P is the set of all ground atoms that are logical consequences of P. The least Herbrand model provides the model theoretic semantics for P: $P \models A$ if and only if $A \in lhm(P)$ where A is a ground atom.

A *proof procedure* consists of a set of (logical) axioms and a set of inference rules. Given a proof procedure π, we say that ϕ is *provable* from the set of formulas Σ and write $\Sigma \vdash_\pi \phi$ if there exist a finite sequence of formulas $\phi_1, \phi_2, \ldots, \phi_n$ which is obtained by successive applications of inference rules to axioms, formulas in Σ, or previous formulas in the sequence, or combinations of these, while ϕ_n is the conclusion ϕ. Such a sequence of formulas, if it exists, is called a *proof* of ϕ from Σ. A proof procedure π is *sound*, with respect to the model-theoretic semantics, if $\Sigma \models \phi$ whenever $\Sigma \vdash_\pi \phi$; it is *complete* if $\Sigma \vdash_\pi \phi$ whenever $\Sigma \models \phi$.

A proof procedure for clausal logic is *resolution* [Rob65]. In this proof procedure the

[1]We use the same symbol for the entailment relation and for the satisfaction relation between interpretations and formulas in order to follow the standard logic practice. In cases where this may cause misunderstanding, the intended meaning will be indicated in words.

set of axioms is empty since there is no interaction between logical connectives due to the normal form in which clauses are written. The set of inference rules contains only one rule, resolution, which allows one to infer, from two clauses $F_1 \vee L_1$ and $F_2 \vee \sim L_2$, the clause $(F_1 \vee F_2)\theta$, where θ is the *most general unifier* of L_1 and L_2 (the minimal substitution such that $L_1\theta = L_2\theta$). For definite clauses, this consists in matching the head of one clause with a literal in the body of another. The resolution proof procedure is not complete but is *refutation-complete*, i.e. if a set of clauses is inconsistent, resolution is able to derive the unsatisfiable empty clause $\Box$ [Fla95]. Therefore, proofs of $\Sigma \vdash \phi$ where ϕ is a conjunction of positive literals, are transformed into refutation proofs $\Sigma \cup \{\sim \phi\} \vdash \Box$, where $\sim \phi$ is a denial called a *query* and written $? - B_1, \ldots, B_m$.

Logic Programming is obtained by considering Horn clauses only and by adopting a particular version of resolution (called *SLD resolution* [Kow74]) that is efficient for Horn clauses. In SLD resolution the initial formula is the negated goal $\phi_1 = \sim \phi$ and, at each step, the new formula ϕ_{i+1} is obtained by resolving the previous formula ϕ_i with a variant of a clause from the initial set Σ. SLD resolution was proven to be *sound* and *complete* for Horn clauses (the proofs can be found in [Llo87]).

A particular Logic Programming language is defined by choosing a rule for the selection of the literals in the current formula to be reduced at each step (*computation rule*) and by choosing a search strategy, that can be either depth first or breadth first. In the *Prolog* [CKRP73] programming language the computation rule selects the *left-most* literal in the current goal and the search strategy is depth first with chronological backtracking. Moreover, Prolog adopts an extension of SLD resolution called *SLDNF resolution* that is able to deal with normal clauses by *negation as failure* [Cla78].

3.2 Learning from Entailment

In learning from entailment, the training set E is expressed as a set of ground facts, the background knowledge and the hypothesis are definite programs and the coverage relation is defined as follows [LD94]:

Definition 5 (Learning from Entailment - Coverage) *Given a background knowledge B, a hypothesis H and an example set E, the hypothesis H covers example $e \in E$ with respect to background knowledge B if $B \cup H \models e$, i.e.*

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

As a consequence, the function $covers(H, B, E)$ can be defined as

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

We say that a hypothesis H is *complete* if $covers(B, H, E^+) = E^+$ and that is *consistent* if $covers(B, H, E^-) = \emptyset$.

The framework of learning from entailment has been also called *normal setting* [MDR94] or *explanatory setting* [DRD96a] for learning because examples have to be explained by the learned theory.

The task of learning from entailment can then be defined as follows [BG95].

Definition 6 (Learning from Entailment Problem)
Given:

- *a set $\mathcal{P}$ of possible programs (language bias)*
- *a set E^+ of positive examples (ground facts)*
- *a set E^- of negative examples (ground facts)*
- *a logic program B (background knowledge).*

Find:

- *a logic program $P \in \mathcal{P}$ such that:*
- $\forall e^+ \in E^+$, $B \cup P \models e^+$ *(completeness)*
- $\forall e^- \in E^-$, $B \cup P \not\models e^-$ *(consistency)*

The program P is called *target program.* Depending on whether the training set contains facts for one or more predicate, the target program will contain a definition for one or more predicates and we speak, respectively, of *single predicate learning* or *multiple predicate learning.* Learning multiple predicates poses a number of problems that are discussed in section 4.5.1. The hypothesis space $\mathcal{P}$ is defined by the language bias and has to be restricted as much as possible in order to contain the computational complexity of the learning task. Various forms of restriction have been used in ILP, some of them are hardwired into the system while some other can be user-defined (declarative-bias). Examples of hardwired restrictions are: function-free programs (FOIL [Qui90a]) or determinacy (GOLEM [MF90]). Examples of user-defined restrictions are: types and symmetry of predicates in pairs of arguments [LDG91a], input/output modes [Sha83], program schemata or rule models [Wro88, Mor91], clause sets [BG95], parametrized languages [DR92], integrity constraints [DRBM91] and determinations [Rus89].

In the following, we will consider only a very simple bias in the form of a set of literals which are allowed in the body of the clauses for the target predicates, which corresponds to a simplified version of the clause sets adopted in [BG95].

Let us now consider a simple example.

Example 7 *Suppose we want to learn the concept grandfather from the background knowledge:*

$$
\begin{array}{l}
father(X,Y) \leftarrow parent(X,Y), male(X) \\
parent(john, mary) \\
parent(ann, mary) \\
parent(mary, steve) \\
male(john) \\
female(mary)
\end{array}
$$

and the training sets:

$$
\begin{array}{l}
E^+ = \{grandfather(john, steve)\} \\
E^- = \{grandfather(ann, steve), grandfather(john, mary)\}
\end{array}
$$

Suppose also that the hypothesis space $\mathcal{P}$ is described in this way:
$\mathcal{P}$ is the set of clauses of the type $grandfather(X,Y) \leftarrow \alpha$ where α is a conjunction of literals chosen among the following:

$father(X,Y), father(X,Z), father(Z,Y),$
$parent(X,Y), parent(X,Z), parent(Z,Y),$
$male(X), male(Y), male(Z),$
$female(X), female(Y), female(Z)$

The following program P is a solution to this ILP problem because it covers the positive examples and does not cover any of the negative ones:

$grandfather(X,Y) \leftarrow father(X,Z), parent(Z,Y)$

Operationally, the entailment relation is usually tested by means of SLD-resolution, either depth-bounded or unbounded. In depth-bounded SLD-resolution, a limit is placed on the derivation depth in order to avoid loops, as for example in MIS [Sha83] and CIGOL [MB92].

The notion of coverage defined above is called *intensional coverage* because the background knowledge B is intensional and can contain both ground facts and non-ground clauses. However, many ILP systems use a different notion of coverage, namely *extensional coverage*, where the background knowledge B is extensional, i.e., it is a set of ground facts only. Examples of systems employing extensional coverage are FOIL [Qui90a], ICN [MV95b], MULT_ICN [MV95a], FOCL [PK92], MIS [Sha83] (with the lazy strategy) and GOLEM [MF90].

In the case in which B is intensional, extensional ILP systems have first to transform it into a ground model M of B. In section 3.4.1 a technique is described for ensuring that M is finite. Given the model M of the background knowledge B, extensional ILP systems employ the coverage relation that is defined below [DRLD93].

Definition 8 (Learning from Entailment - Extensional Coverage) *A hypothesis H extensionally covers an example $e \in E$ with respect to a ground model M of the background knowledge if there exists a clause $C \in H$, $C = T \leftarrow Q$, and a substitution θ, such that $T\theta = e$ and $Q\theta = \{L_1, \ldots, L_m\}\theta \subseteq M$. In this case the following notation is used: $covers_{ext}(M,H,e) = true$.*

Operationally, in order to test recursive definitions, the model M must represent not only the background knowledge but also the predicates we want to learn, also called *target predicates*. As the definition for the target predicates is unknown at the time of learning, the model of $B \cup H$ is approximated by computing the model of $B \cup E^+$. Atoms in the training set can then be used for the resolution of recursive literals in the body of clauses.

It is important to note that extensional coverage is not equivalent to intensional coverage due to the approximations introduced: the use of a h-easy model of B and the use of positive examples for representing the definitions of target predicates. In particular, for definite logic programs, we can have the following cases [DRLD93]: (i) extensional consistency, intensional inconsistency; (ii) intensional completeness, extensional incompleteness; (iii) extensional completeness, intensional incompleteness. Let us illustrate each of these cases with an example.

We have the case of extensional consistency, intensional inconsistency when a hypothesis consistent if tested extensionally but inconsistent if tested intensionally.

Example 9 (Extensional consistency, intensional inconsistency) *Consider the problem of learning the concept father and male_ancestor from a background knowledge containing the following facts about parent, male and female:*

$B = \{parent(a,b), parent(b,d), parent(c,b), male(a), female(c)\}$

The training set is specified as follows:

$E^+ = \{male\_ancestor(a,b), male\_ancestor(a,d), father(a,b)\}$

$E^- = \{male\_ancestor(c,b), male\_ancestor(c,d), father(b,a)\}$

In this case, the following hypothesis is extensionally consistent but not intensionally consistent:

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

$male\_ancestor(X,Y) \leftarrow father(X,Y)$

$male\_ancestor(X,Y) \leftarrow male\_ancestor(X,Z), parent(Z,Y)$

because negative example $male\_ancestor(c,b)$ *(with* $female(c)$ *and* $parent(c,b)$ *in the background) will be covered.*

We have the case of intensional completeness, extensional incompleteness when a hypothesis intensionally covers all the positive examples but not extensionally because some example needed for covering other examples is missing from the training set.

Example 10 (Intensional completeness, extensional incompleteness) *Suppose the background knowledge and training set are given:*

$B = \{parent(john, steve), parent(bill, john), parent(john, mike),$
$parent(mike, sue)\}$

$E^+ = \{ancestor(john, steve), ancestor(bill, steve), ancestor(john, sue)\}$

The theory:

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

$ancestor(X,Y) \leftarrow ancestor(X,Z), parent(Z,Y)$

is intensionally complete but extensionally incomplete because it does not cover the example $ancestor(john, sue)$ *since the positive example* $ancestor(john, mike)$ *is missing.*

The case of extensional completeness, intensional incompleteness occurs when we learn a program with an infinite recursive chain.

Example 11 (Extensional completeness, intensional incompleteness) *Consider the training set:*

$E^+ = \{even(0), odd(1)\}$

and the background predicate $succ(X,Y)$ *that expresses that* Y *is the successor of* X*. The program:*

$even(X) \leftarrow succ(X,Y), odd(Y)$

$odd(X) \leftarrow succ(Y,X), even(Y)$

is extensionally complete but intensionally incomplete, because the intensional derivation of $even(0)$ *would lead to a loop.*

3.2.1 Soundness and Completeness

In this section, we define the properties of soundness and completeness for a learning algorithm with respect to the problem definition above 6. We adopt the notion of an inductive inference machine (IIM) that is a formalization of the concept of a learning system. If M is an IIM, we write $M(\mathcal{P}, E^+, E^-) = P$ to indicate that, given a hypothesis space $\mathcal{P}$, positive and negative examples E^+ and E^-, and a background knowledge B, the machine outputs a program P. We write $M(\mathcal{P}, E^+, E^-) = \perp$ when M fails in finding a solution, either because it does not terminate or because it stops without having found any program satisfying the problem conditions. A system is able to solve the ILP problem when it is produces only programs that are complete and consistent and it finds such a program when it exists. A system satisfying the first requirement is called *sound*, while a system satisfying the second requirement is called *complete*. Formally, we have the following definitions.

Definition 12 *An IIM is* sound *iff, if $M(\mathcal{P}, E^+, E^-) = P$, then $P \in \mathcal{P}$ and P is complete and consistent with respect to E^+ and E^-.*

Definition 13 *An IIM is* complete *iff, if $M(\mathcal{P}, E^+, E^-) = \perp$, then there is no $P \in \mathcal{P}$ that is complete and consistent with respect to E^+ and E^-.*

It is important to note the difference between the notions of completeness of a program with respect to the examples and the background knowledge and the completeness of an IIM. A complete program is one that entails all positive examples, while a complete IIM is an IIM that is able to find a complete and consistent program when there exist such a program in $\mathcal{P}$.

3.2.2 Classification of Systems

ILP systems adopting learning from entailment can be classified according to a number of criteria [LD94]. First, they can be divided into *batch learners* that require all the training examples to be given before the learning starts or *incremental learners* that accept examples one by one. Second, we have *interactive* and *non-interactive* learners depending on whether they rely or not on an oracle to verify the validity of generalizations and/or classify examples generated by the learner. Third, some systems learn a concept from scratch while others start from an initial definition of the concept and revise it. The latter class of systems are called *theory revisors*. Finally, some systems are able to learn the definition of just one predicate while others may learn the definition of multiple predicates.

While in principle these dimensions are orthogonal and systems can be build exhibiting any possible combination of the above features, in practice existing ILP systems are situated at two extremes of the spectrum. On one side we have batch, non-interactive systems that learn the definition of one concept from scratch, on the other side we have incremental, interactive systems that learn the definition of multiple concepts by revising an initial hypothesis. Systems of the first type are called *empirical ILP systems* while systems of the second type are called *interactive ILP systems* or *incremental ILP systems* [DR92].

Examples of empirical ILP systems are FOIL [Qui90a], Progol [Mug95b], mFOIL [Dže91, DB92], GOLEM [MF90], LINUS [LDG91b] and TRACY [BG94b]. Examples of interactive ILP systems are MIS [Sha83], MARVIN [SB86], CLINT [DRB89, DRB92b], CIGOL [MB92], and FILP [BG93].

3.2.3 Imperfect Data

Real world data is often *imperfect*, i.e., the examples and/or the background knowledge may contain various kinds of errors, either random or sistematic, or may not be complete. In such cases, the requirements imposed by the definition of the ILP problem that all the positive examples and none of the negative are covered, may be relaxed, in order to allow the system to look for true regularities in the data and to discard specific cases due to chance or error.

In [LD92] the authors distinguish various types of imperfections of the data when learning definitions of relations:

- noise, i.e., random errors in the training examples and background knowledge;
- insufficiently covered example space, i.e., too sparse training set;
- inexactness, i.e, inappropriate or insufficient hypothesis space which does not contain an exact description of the target concept;
- missing values in the training examples.

Another type of imperfection must be added to these types: missing information from the background knowledge. A ground fact from the background knowledge usually express information about a specific example, if the knowledge about that example could not be completely acquired, some of the facts relative to them may be missing. We call this type of imperfections *incompleteness of the background* and we will consider it in chapter 4.

3.2.4 Hypothesis Space Ordering

In [Mit82] it is shown that concept learning can be seen as a search problem where the states of the search space are the possible concept descriptions. In order to search the space of concept descriptions sistematically, it is necessary to structure it by introducing a partial order. Typically, this partial order is given by a *generality relation*. Intuitively, a concept description C_1 is *more general than* a concept description C_2, usually represented as $C_1 \prec C_2$, if the set of objects covered by C_2 is a strict subset of those covered by C_1.

Most ILP systems build a target program by repeatedly searching the space of possible clauses instead of the space of programs. Therefore, a generality ordering for the space of possible clauses will be defined.

The generality relation for program clauses can be defined in the following way: a clause C_1 is *more general or equally general* than a clause C_2 with respect to the background knowledge B if $B \cup \{C_1\} \models \{C_2\}$ because, in this case, all the examples covered by C_2 will be covered as well by C_1. In practice, however, a syntactic relation called *θ-subsumption* [Plo70] is used in place of entailment in the definition of generality for two reasons: first, θ-subsumption can be verified by a simple and fast algorithm, while entailment is non-decidable, and, second, it introduces a *lattice* in the space of clauses, which provides an important generalization operator, as will be shown below. In the following definition, clauses are represented as sets of literals.

Definition 14 (θ-subsumption) *Clause C_1* θ-subsumes *C_2 if there exist a substitution θ such that $C_1\theta \subseteq C_2$ [Plo70]. Two clauses C_1 and C_2 are* θ-subsumption equivalent *if C_1* θ-subsumes *C_2 and C_2* θ-subsumes *C_1. A clause is* reduced *if it is not θ-subsumption equivalent to any proper subset of itself.*

We now give some examples to illustrate the notion of θ-subsumption.

Example 15 *Consider the following clause* C_1

$$C_1 = grandfather(X, Y) \leftarrow father(X, Z)$$

Clause C_1 θ-subsumes *the clause*

$$C_2 = grandfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)$$

with the empty substitution $\theta = \emptyset$. *Clause* C_1 *also* θ-subsumes *the clause*

$$C_3 = grandfather(john, steve) \leftarrow father(john, mary)$$

with the substitution $\theta = \{X/john, Y/steve, Z/mary\}$. *Clause* C_1 θ-subsumes *the clause*

$$C_4 = grandfather(john, steve) \leftarrow father(john, mary), parent(mary, steve)$$

with the substitution $\theta = \{X/john, Y/steve, Z/mary\}$.

The following clause, instead,

$$C_5 = grandfather(X, Y) \leftarrow father(X, Z), father(W, V)$$

is θ*-subsumption equivalent to* C_1*. Therefore, clause* C_1 *is reduced, while* C_5 *is not.*

θ-subsumption has the important property that if C_1 θ-subsumes C_2, then $C_1 \models C_2$. This is the reason why it can be used to approximate entailment. On the other hand, the converse property is not always true, as it is shown by this examples proposed by Flach [Fla92].

Example 16 *Consider the following two clauses*

$$C_1 = list([V|W]) \leftarrow list(W)$$
$$C_2 = list([X, Y|Z]) \leftarrow list(Z)$$

Clearly, $C_1 \models C_2$*, as can be shown by resolving* C_1 *against itself. However, there is no substitution* θ *such that* $C_1\theta \subseteq C_2$*, since it should map* W *to both* Z *and* $[Y|Z]$*. Therefore* C_1 *does not* θ-subsumes C_2.

Thus generality can be re-defined in terms of θ-subsumption. Clause C_1 is *more general or equally general as* clause C_2 ($C_1 \preceq C_2$) if C_1 θ-subsumes C_2. Clause C_1 is *more general* than C_2 ($C_1 \prec C_2$) if $C_1 \preceq C_2$ holds and $C_2 \preceq C_1$ does not [LD94]. If $C_1 \prec C_2$ we say that C_2 is a *specialization* of C_1 or that C_1 is a *generalization* of C_2.

θ-subsumption has another important property: the generality relation $\preceq_\theta$ it induces, introduces a *lattice* in 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 the *glb* are unique up to renaming of the variables.

This property leads to the definition of the following notion.

Definition 17 (Least General Generalization) *The* least general generalization *of two reduced clauses* C_1 *and* C_2*, denoted by* $lgg(C_1, C_2)$*, is the least upper bound of* C_1 *and* C_2 *in the* θ*-subsumption lattice [Plo70].*

The algorithm for computing the lgg of two clauses was given in [Plo70]. In order to compute the lgg of two clauses, we have to compute the lgg of two terms and of two literals.

The lgg of two terms $f_1(s_1, \ldots, s_n)$ and $f_2(t_1, \ldots, t_n)$ is defined as $f_1(lgg(s_1, t_1), \ldots, lgg(s_n, t_n))$ if $f_1 = f_2$ and is a new variable V if $f_1 \neq f_2$. The variable V is used to represent the lgg of $f(s_1, \ldots, s_n), g(t_1, \ldots, t_n)$ and it must be used for all the occurrences of the lgg of the same subterm. The following are examples of lgg of terms:

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

Note that the same variable X is used in both arguments in the second example because it stands for lgg of the same two terms a and b.

The lgg of two literals $L_1 = (\sim)p(s_1, \ldots, s_n)$ and $L_2 = (\sim)q(t_1, \ldots, t_n)$ is undefined if L_1 and L_2 do not have the same predicate symbol and sign; otherwise is defined as

$$lgg(L_1, L_2) = (\sim)p(lgg(s_1, t_1), \ldots lgg(s_n, t_n))$$

The following are examples of lgg of literals:

$$lgg(parent(john, mary), parent(john, steve)) = parent(john, X)$$
$$lgg(parent(john, mary), \sim parent(john, steve)) = undefined$$
$$lgg(parent(john, mary), father(john, steve)) = undefined$$

The lgg of two clauses $C_1 = \{L_1, \ldots, L_n\}$ and $C_2 = \{K_1, \ldots, K_m\}$ is defined as:

$$lgg(C_1, C_2) = \{lgg(L_i, K_j) \mid L_i \in C_1,\ K_j \in C_2 \text{ and } lgg(L_i, K_j) \text{ is defined}\}$$

For example, consider the clauses:

$$C_1 = father(john, mary) \leftarrow parent(john, mary), male(john)$$
$$C_2 = father(david, steve) \leftarrow parent(david, steve), male(david)$$

The lgg of these clauses is:

$$lgg(C_1, C_2) = father(X, Y) \leftarrow parent(X, Y), male(X)$$

As another example, consider the lgg of the following two clauses [BG95]:

$$C_1 = win(conf1) \leftarrow occ(place1, x, conf1), occ(place2, o, conf1)$$
$$C_2 = win(conf2) \leftarrow occ(place1, x, conf2), occ(place2, x, conf2)$$

that represents two winning configuration in a two-person game with two places that can be occupied by an x or an o. The lgg of the two clauses is

$$\begin{aligned} lgg(C_1, C_2) = win(Conf) \leftarrow\ & occ(place1, x, Conf), occ(L, x, Conf), \\ & occ(M, Y, Conf), occ(place2, Y, Conf) \end{aligned}$$

This clause is not reduced, some literals are redundant and can be eliminated obtaining the following reduced clause:

$$lgg(C_1, C_2) = win(Conf) \leftarrow occ(place1, x, Conf), occ(place2, Y, Conf)$$

expressing that a configuration is winning if it contains x in the first place and anything in the second.

The length of the lgg of two clauses C_1 and C_2 can be at most $|C_1| \times |C_2|$, therefore the repeated application of this operator can produce clauses with an exponential length. However, clauses produced by lgg often contain irrelevant literals and should be reduced in order to get more compact and/or efficient theories. Plotkin proposed an algorithm for reducing clauses which unfortunately is NP-complete.

There are two broad categories of ILP methods which adopt θ-subsumption to obtain learning from entailment: *bottom-up* methods that search the space of clauses from specific to general, and *top-down* methods, that search the space of clauses from general to specific.

3.2.5 Bottom-up methods

In bottom-up methods, clauses are generated by starting with the most specific clause that covers one or more positive examples and no negative example, and by iteratively applying generalization operators to the clause until it cannot be further generalized without covering negative examples. Bottom-up methods are best suited for interactive and incremental learning from few examples.

Examples of bottom-up techniques are: Relative Least General Generalization (RLGG) [Plo70], Inverse Resolution [MB92] and Inverse Implication [LM92]. In the following, Relative Least General Generalization will be presented.

Relative Least General Generalization

The notion of least general generalization [Plo70] provides a generalization operator. However, this operator can not be used directly in practical systems since it does not take into account the background knowledge, therefore Plotkin introduced the notion of *relative least general generalization*. This generalization operator is used, for instance, in GOLEM (see section 3.4.1).

When the background knowledge consists of ground facts, the relative least general generalization ($rlgg$) of two clauses $C_1 = H_1 \leftarrow B_1$ and $C_2 = H_2 \leftarrow B_2$ can be defined as

$$rlgg(C_1, C_2) = lgg((H_1 \leftarrow B_1, K), (H_2 \leftarrow B_2, K))$$

where K represents the conjunction of all the background facts.

Thus the problem of computing the $rlgg$ of two clauses can be reduced to the problem of computing the lgg of two clauses, for which an algorithm was given by Plotkin [Plo70].

In the following, two examples of $rlgg$ of two clauses are given.

Example 18 *Consider the two positive examples* $e_1 = father(john, mary)$ *and* $e_2 = father(david, steve)$ *and consider a background knowledge* B *consisting of the conjunction of the following facts*

$$parent(john, mary), parent(ann, mary), parent(mary, steve),$$
$$parent(david, steve), male(john), female(mary),$$

The $rlgg$ *of the two example* e_1 *and* e_2 *(that can be interpreted as the clauses* $e_1 \leftarrow$ *and* $e_2 \leftarrow$*) is*

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

Example 19 *[BG95] Consider the following two clauses:*

$C_1 = uncle(X,Y) \leftarrow brother(X, father(Y))$
$C_2 = uncle(X,Y) \leftarrow brother(X, mother(Y))$

and a background knowledge containing the two facts

$parent(father(X), X)$
$parent(mother(X), X)$

The rlgg of the two clauses with respect to the available background is:

$rlgg(C_1, C_2) = uncle(X,Y) \leftarrow brother(X,Z), parent(Z,Y)$

If we had computed the lgg of the two clauses without taking into account the background knowledge, we would have obtained the clause:

$lgg(C_1, C_2) = uncle(X,Y) \leftarrow brother(X,Z)$

which is not very representative of the uncle relation.

3.2.6 Top-down Methods

Top-down methods search the space of clauses from general to specific. They employ a *refinement operator* that is based on θ-subsumption.

Definition 20 (Refinement Operator) *Given a space of possible clauses $\mathcal{C}$ (defined by the language bias), a refinement operator ρ maps a clause C into the set of clauses $\rho(C)$ that are specializations (refinements) of C:*

$$\rho(C) = \{C' \mid C' \in \mathcal{C},\ C \prec C'\}$$

Typically a refinement operator generates only the minimal (most general) refinements of a clause. A refinement operator applies two basic syntactic operations to a clause:

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

Top-down methods share a basic algorithm that is given as follows (adapted from [LD94]):

algorithm LearnTopDown(
  inputs : E : training set,
    B : background theory,
  outputs : H : learned theory)
Initialize $H := \emptyset$
Initialize $E_{cur} := E$
repeat (*Covering loop*)
  GenerateClause($E_{cur}, B; C$)
  Add C to H to obtain the new hypothesis $H' := H \cup C$

Remove positive examples covered by C from E_{cur} to get
$E'_{cur} := E_{cur} - covers(B, H, E_{cur})$
Assign $E_{cur} := E'_{cur}$, $H := H'$
until Sufficiency stopping criterion is satisfied

procedure GenerateClause(
inputs : E : training set,
B : background theory,
outputs : C : clause)
Select a predicate p that must be learned
Initialize C to be $p(\overline{X}) \leftarrow$.
repeat (*Specialization loop*)
Find the refinement $C_{best} \in \rho(C)$
according to some heuristic function
Assign $C := C_{best}$
until Necessity stopping criterion is satisfied
return C

FOIL [Qui90b], mFOIL [Dže91] and Progol [Mug95a] are examples of systems based on this algorithm.

The algorithm starts with an empty hypothesis H and a current set of example E_{cur} that is initially set to the entire training set. The algorithm is composed of two repeat loops, referred to as *covering* and *specialization* loop.

At each iteration of the covering loop a clause is generated, it is added to the theory and the positive examples covered by it are removed from the training set. The loop terminates when the sufficiency stopping criterion is met, which typically happens when no more positive examples are left in the training set.

Each clause is generated by the specialization loop. The loop starts with a clause of the form $T \leftarrow$ and successively refines it by means of the ρ refinement operator. Given a clause $C = T \leftarrow Q$, ρ builds the set of its refinements $\rho(C) = \{C' \mid C \prec C'\}$ by adding a literal to the body of C. Therefore, every refinement C' has the form $C' = T \leftarrow Q, L$. One of two search strategies are usually employed: hill-climbing (as in the algorithm above) or beam-search. In hill-climbing the algorithm stores the clause that is best according to some heuristic function and replaces it with the best refinement at each specialization step, until the necessity stopping criterion is satisfied. In beam-search, a set of clauses is kept instead of one and the best one of them is chosen for refinement at each step.

The two repeat loops are controlled by two stopping criteria:

- a necessity stopping criterion, that decides when to stop the addition of a clause to a theory in the covering loop,
- a sufficiency stopping criterion, that decides when to stop adding literals to a clause.

The stopping criteria differ in case of domains where the information is perfect and domains where the information is noisy. In domains with perfect data, the necessity stopping criterion requires consistency, i.e., no negative examples must be covered by the clause, while the sufficiency criterion requires completeness, i.e., all the positive examples must be covered. In domains with noisy data, heuristic stopping criteria are employed that relax the consistency and completeness requirements.

Heuristics

Different heuristics can be used for clause evaluation. They can be basically divided into two families, those based on the *expected accuracy* of a clause and those based on *informativity*. The expected accuracy of a clause C is defined as

$$A(C) = p(\oplus|C)$$

where $p(\oplus|C)$ is the probability that an example covered by C is positive. Informativity is defined as

$$I(C) = -\log_2 p(\oplus|C)$$

that represents the information needed to signal that an example randomly chosen among those covered by C is positive.

In some systems, clauses are evaluated on the basis of the gain produced by the addition of a literal: we have *accuracy gain* $AG(C, C') = A(C') - A(C)$ and *information gain* $IG(C, C') = I(C) - I(C')$. Since these heuristics may favour very specific clauses with high gain, weights are introduced in these equations in order to take into account the number of examples covered by each clause. If $n^{\oplus}(C)$ and $n^{\oplus}(C')$ are, respectively, the number of positive examples covered by C and C', the weight is given by $n^{\oplus}(C)/n^{\oplus}(C')$. Therefore we have *weighted accuracy gain* $WAG(C, C') = (n^{\oplus}(C)/n^{\oplus}(C')) \times (A(C') - A(C))$ and *weighted information gain* $WIG(C, C') = (n^{\oplus}(C)/n^{\oplus}(C')) \times (I(C) - I(C'))$. All the heuristic functions previously described are based on the probability $p(\oplus|C)$ that an example covered by clause C is positive. This probability can be estimated from the current training set E_{cur} by using various estimating functions the simplest of which is the *relative frequency* of covered positive examples $n^{\oplus}(C)$ with respect to all the examples $n(C)$ covered by the clause C: $p(\oplus|C) = \frac{n^{\oplus}(C)}{n(C)}$. More appropriate probability estimates are the *Laplace estimate* and the *m-estimate* that will be discussed in section 3.4.3.

Example 21 *In the following, we show the behaviour of the top-down algorithm in the case of example 7. The heuristic function adopted is expected accuracy using the relative frequency as the probability estimate and the stopping criteria are completeness and consistency.*

The algorithm starts by initializing E_{cur} to

$$\{grandfather(john, steve)^{\oplus}, grandfather(ann, steve)^{\ominus}, grandfather(john, mary)^{\ominus}\}$$

and H to $\emptyset$. Then the covering loop is entered and the procedure GenerateClause is called. The clause C is initialized to $grandfather(X, Y) \leftarrow$ and the specialization loop is started. The refinement operator $\rho(C)$ that is employed takes the clause C and adds (by set union) one of the following literals to C[2]*:*

$$father(X, Y), father(X, Z), father(Z, Y),$$
$$parent(X, Y), parent(X, Z), parent(Z, Y)$$

This operator determines the search space, that is shown in figure 3.1.

In the first iteration of the specialization loop, the refinements shown at the first level of the search tree are generated. Among those, the clause

$$C_1 = grandfather(X, Y) \leftarrow father(X, Z)$$

[2] A simpler language bias with respect to example 7 is considered for simplicity.

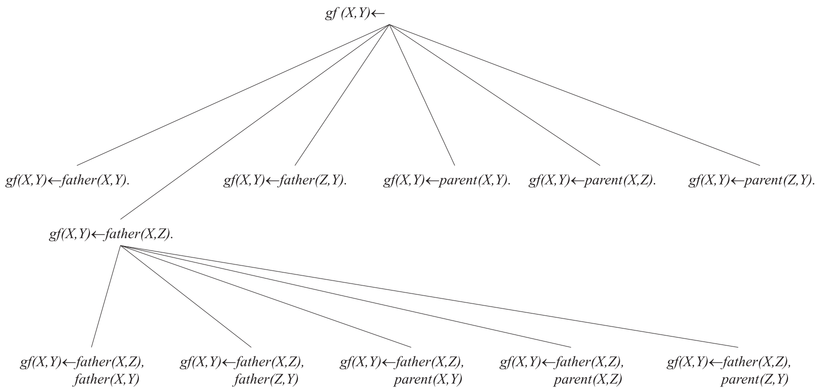

Figure 3.1: The search space for the predicate *grandfather* (abbreviated *gf*).

is chosen for further refinement because it is the one that has the highest accuracy, covering the positive examples and only one of the negative.

Among the refinements generated in the second iteration of the specialization loop, the clause

$$C_2 = grandfather(X,Y) \leftarrow father(X,Z), parent(Z,Y)$$

is chosen since it is the most accurate, covering one positive example and no negative one.

At this point, the specialization loop ends since the clause is consistent and control is given back to the covering loop. The clause is added to the current hypothesis obtaining

$$H = \{grandfather(X,Y) \leftarrow father(X,Z), parent(Z,Y)\}$$

and the positive examples covered by C_2 are removed from E_{cur}. Since no positive example is left, the covering loop terminates and the algorithm ends by returning H.

3.2.7 Generality of Learned Solutions

Both bottom-up and top-down methods find clauses that are consistent and cover a subset of positive examples. However, depending on the technique adopted, the generality of clauses differs. Suppose a bottom-up method finds a clause C_1 that covers a subset of positive examples E_1^+, then C_1 will be the least general clause in the hypothesis space that covers E_1^+ and is consistent. Suppose a top-down method finds a clause C_2 that covers a subset of positive examples E_2^+, then C_2 will be the most general clause in the hypothesis space that covers E_2^+ and is consistent.

Example 22 *Suppose we want to learn the concept growl from the background knowledge:*

$wolf(albert)$	$has\_four\_legs(albert)$	$has\_tail(albert)$
$wolf(virginia)$	$has\_four\_legs(virginia)$	$has\_tail(virginia)$
$coyote(wile)$	$has\_four\_legs(wile)$	$has\_tail(wile)$
$coyote(peter)$	$has\_four\_legs(peter)$	$has\_tail(peter)$

$cat(tom)$	$has\_four\_legs(fufy)$	$has\_tail(fufy)$
$cat(krazy)$	$has\_four\_legs(krazy)$	$has\_tail(krazy)$

and the training sets:

$$E^{+} = \{growl(albert), growl(virginia), growl(wile), growl(peter)\}$$
$$E^{-} = \{growl(tom), growl(krazy)\}$$

The hypothesis space $\mathcal{P}$ is given by the set of clauses of the type $growl(X) \leftarrow \alpha$ where α is a conjunction of literals chosen among the following:

$$wolf(X), coyote(X), cat(X), has\_four\_legs(X), has\_tail(X)$$

A bottom-up algorithm would find the clauses:

$$C_{b,1} = growl(X) \leftarrow wolf(X), has\_four\_legs(X), has\_tail(X)$$
$$C_{b,2} = growl(X) \leftarrow coyote(X), has\_four\_legs(X), has\_tail(X)$$

the first covering the example set $E_1^+ = \{growl(albert), growl(virginia)\}$ and the second covering the example set $E_2^+ = \{growl(wile), growl(peter)\}$. A top-down algorithm would, instead, find the clauses:

$$C_{t,1} = growl(X) \leftarrow wolf(X)$$
$$C_{t,2} = growl(X) \leftarrow coyote(X)$$

covering, respectively, the same sets of examples E_1^+ and E_2^+.

Given the hypothesis space $\mathcal{P}$, $C_{b,1}$ and $C_{b,2}$ are the least general clauses covering the set of examples E_1^+ and E_2^+, while $C_{t,1}$ and $C_{t,2}$ are the most general clauses covering E_1^+ and E_2^+.

Let us now consider solutions of the learning problem that consist of the set of clauses $P = \{C_1, \ldots, C_n\}$ covering respectively, the sets of examples $E_1^+, \ldots, E_n^+$. In this case, a bottom-up method will find a solution composed of least general clauses, that is the Least General Solution (LGS for short) among those of the form above. On the other hand, a top-down method will find a solution composed of most general clauses, that is the Most General Solution (MGS for short) among those of the form above. In example 22, $P_b = \{C_{b,1}, C_{b,2}\}$ is the LGS, while $P_t = \{C_{t,1}, C_{t,2}\}$ is the MGS.

In general, a bottom-up method and a top-down method may find solutions that partition differently the set of positive examples. In this case, the two solutions may not be directly comparable in terms of generality. However, with abuse of terminology, we still say that the solution that is found by a bottom-up method is a LGS, and that the solution that is found by a top-down method is a MGS. In this case, instead of "the" LGS and "the" MGS we speak of "a" LGS and "a" MGS because LGSs and MGSs are not unique, since they depend on the way the positive examples set is partitioned.

The ILP techniques to be used thus depends on the level of generality that we want to have for the definition of a predicate. In chapter 5 we will discuss various criteria that can be adopted for choosing the generality of the definition of a predicate.

3.3 Learning from Interpretations

In learning from interpretations, examples are Herbrand interpretations, i.e., sets of ground facts, and the theory that is learned is a clausal theory. Each example represents observations relative to a particular situation in the world. The coverage relation is defined as follows [DR97]:

Definition 23 (Learning from Interpretations) *Given a background knowledge B, a hypothesis H and an example set E, the hypothesis H covers example $e \in E$ with respect to background knowledge B if $M(B \cup e)$ is a model for H, i.e.*

$$covers(B, H, e) = true \text{ if } M(B \cup e) \models H$$

As a consequence, the function $covers(H, B, E)$ can be defined as

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

The test of whether a clause $C = A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$ makes an interpretations true or not can be performed by using Prolog by asserting both the background knowledge and an interpretation into the knowledge base, and running the query $? - B_1, \ldots, B_m, not\ A_1, \ldots, not\ A_n$.

The task of learning from interpretations can then be defined as follows [BG95]:

Definition 24 (Learning from Interpretations Problem)
Given

- *a set $\mathcal{H}$ of possible clausal theories (language bias)*
- *a set of positive examples E^+ (interpretations),*
- *a set of negative examples E^- (interpretations),*
- *a logic program B (background knowledge).*

Find *a clausal theory H such that*

- *for all $e^+ \in E^+$, $M(B \cup e^+)$ is a true interpretation of H, i.e., $M(B \cup e^+) \models H$ (Completeness);*
- *for all $e^- \in E^-$, $M(B \cup e^-)$ is a false interpretation of H, i.e., $M(B \cup e^-) \not\models H$ (Consistency).*

As in learning from entailment, the hypothesis space $\mathcal{H}$ is defined by the language bias and formalisms have been defined for restricting the space, as for example the DLAB [DRD96b] formalism.

When learning from interpretations, the generality relation is defined in the following way: given two hypothesis H_1 and H_2, H_1 is *more general or equally general* than H_2 if and only if $H_2 \models H_1$. In fact, according to the definition of entailment, all the interpretations that are models for H_2 are also models for H_1. Therefore, all the interpretations covered by H_2 will also be covered by H_1. Note that the direction of the entailment relation is the opposite with respect to the one for learning from entailment.

Learning from interpretations was first developed for finding interesting regularities in unclassified data. In this case, no negative example is given and there is the further requirement that the theory is *maximally general*. This means that, if $C \in H$, then any clause more general that C should be false under at least one of the positive examples. The task of learning is the one of finding a theory that holds in all the observed situations, thus expressing interesting regularities on data. This learning framework has been studied by different authors under a number of different names: *non-monotonic setting* [MDR94], *characterizing induction* [DRD96a] or *confirmatory induction* [Fla95]. This setting is particularly useful for performing *data mining* or *knowledge discovery in databases*. An example of a system that learns in such a setting is *Claudien* [DRB93].

Let us now consider an example of the above learning problem when no negative example is given, taken from [DRB93].

Example 25 *Suppose we have the following two interpretations containing observations about different gorilla colonies:*

$$e_1^+ = \{female(liz), male(richard)\}$$
$$e_2^+ = \{female(ginger), male(fred), male(rudolph)\}$$

and we have the background theory B:

$$gorilla(X) \leftarrow female(X)$$
$$gorilla(X) \leftarrow male(X)$$

Suppose also $\mathcal{H}$ be the set of range-restricted, constant-free clauses. A solution is:

$$gorilla(X) \leftarrow female(X)$$
$$gorilla(X) \leftarrow male(X)$$
$$male(X), female(X) \leftarrow gorilla(X)$$
$$\leftarrow male(X), female(X)$$

Note that these clauses express regularities on the given database. All of them are true in the minimal Herbrand models:

$$M(B \cup e_1^+) = \{female(liz), male(richard), gorilla(liz), gorilla(richard)\}$$
$$M(B \cup e_2^+) = \{female(ginger), male(fred), male(rudolph), gorilla(ginger), gorilla(fred), gorilla(rudolph)\}$$

When some negative interpretations are also given, the aim of the system is to find a theory that discriminate positive from negative interpretations, thus expressing regularities on positive interpretations that are false for negative ones. An example of such a system is ICL [DRL95].

The following is an example of learning from interpretations from positive and negative examples.

Example 26 *Suppose we have the same positive observations and background knowledge as example 25, plus the following two sets of negative observations*

$$e_1^- = \{female(liz), male(liz)\}$$
$$e_2^- = \{female(liz), male(liz), fruit(banana)\}$$

Suppose also $\mathcal{H}$ be the set of range-restricted, constant-free clauses. A solution is:

$$\leftarrow male(X), female(X)$$
$$male(X), female(X) \leftarrow gorilla(X)$$

The first clause is necessary to rule out the negative interpretation e_1^-, while the second clause is necessary to rule out the negative interpretation e_2^-.

3.4 Examples of ILP Systems

3.4.1 GOLEM

GOLEM [MF90] is a system that learns theories bottom-up by using *rlgg*. The background knowledge B must contain only ground facts. If B contains some non-ground *Horn* clauses, it must be transformed into a finite ground model. To this purpose, *h-easy* ground models of B ($M_h(B)$) are considered that contains all the ground facts that can be derived from B by a SLD-proof tree of depth less than h. Note that $M_h(B)$ can be still infinite: for example, if $B = \{member(X, [X, Y])\}$ there is only one atom derivable from B but there are infinite ground instantiations of it, such as $member([], [[]])$, $member([], [[], []])$, ... Therefore, an additional constraint is imposed on B: all the clauses in B must be *syntactically generative*, i.e., all the variables in their head must be a subset of the variables in the body. This ensures that the model $M_h(P)$ is finite [MF90].

GOLEM generates a single clause by randomly picking couples of examples, by computing their *rlgg* with respect to the background knowledge and by choosing the one with the greatest coverage of other positive examples. This clause is then generalized by randomly choosing new uncovered positive examples and by computing the *rlgg* of the clause and each of the examples. The resulting clause that covers more examples is chosen and is generalized again until the coverage of the clause stops increasing or until a further generalization would cover some negative examples. Then a post-processing phase follows where irrelevant literals are discarded: if the removal of a literal from the body of a clause does not cause the clause to cover any negative examples, then the literal is irrelevant. In this way the clause is further generalized.

In the case in which there is not a single clause that covers all the positive examples, a covering approach is adopted: the positive examples covered by the generated clause are removed from E^+ and the procedure is iterated until no uncovered positive example remains.

The *rlgg* of two clauses can be very large, in the worst case it grows exponentially with the number of examples. In order to reduce the complexity of clauses, GOLEM uses a constraint on the literals that can appear in the body of a clause. These literals must contain only variables that are *determined*, i.e., their values have to be, directly or indirectly, uniquely determined by the values of the variables in the head of the clause. In order to further reduce the complexity of clauses, GOLEM uses mode declarations (specifying the input and output arguments of a predicate) to reduce the size of the clauses (see [MF90] for details).

3.4.2 FOIL

FOIL [Qui90a] is an empirical top-down system that adopts extensional coverage. The hypothesis language L_c is restricted to functor-free normal program clauses. The language bias can not be explicitly defined by the user but is encoded in the system: literals in the body of clauses can have either a predicate from the background knowledge or a target predicate. At least one of the variables in the arguments of a body literal must appear in the head of the clause or in the literals to its left.

In FOIL, the background knowledge B is given extensionally. Both background and training example facts are represented as tuples of constants. In particular, the training set is represented as a set of tuples labeled by $\oplus$, corresponding to positive examples, and a set of tuples labeled by $\ominus$. Every argument of the target and background predicates is assigned a type, that can be either continuous or discrete. If it is discrete, the set of constants allowed in the type must be specified.

The FOIL algorithm is basically the same as the generic top-down ILP algorithm adopting hill-climbing search. The refinement operator adopted by FOIL refines a clause of the form

$$C_i = p(X_1, X_2, \dots, X_n) \leftarrow L_1, L_2, \dots, L_{i-1}$$

by adding a literal L_i to the body. The literal L_i can be of the following form: $q_k(Y_1, Y_2, \dots, Y_{n_k})$ or $not(q_k(Y_1, Y_2, \dots, Y_{n_k}))$, where q_k is a relation and the Y_j are variables appearing in the clause or new variables; $V_i = V_j$ or $V_i \neq V_j$, where V_i and V_j are variables already present in C_i and of the same type; $V_i = c$ or $V_i \neq c$, where V_i is an already existing variable and c is a constant of the appropriate type, and $V_i \leq V_j$, $V_i > V_j$, $V_i \leq t$ and $V_i > t$, where V_i and V_j are variables already present, with numerical values and of the same type, with t a threshold value chosen by FOIL.

In the specialization loop, FOIL makes use of a *local training set* which is initially set to the current training set $E_1 = E_{cur}$. While E_{cur} consists of n-tuples, the local training set consists of m-tuples, where m is the number of variables in the current clause. Let E_i denote the local training set of tuples that satisfy the current clause $C_i = p(X_1, X_2, \dots, X_n) \leftarrow L_1, L_2, \dots, L_{i-1}$. The local training set E_i can be divided into the set of positive tuples E_i^+ and the set of negative tuples E_i^-.

At each refinement step, the clause C_{i+1} is obtained by adding a literal L_i to the body of the clause C_i. Some of the variables $Y_1, Y_2, \dots, Y_{n_k}$ in L_i belong to the 'old' variables already occurring in C_i, $\{OV_1, \dots, OV_{Old}\}$, while some are 'new', $\{NV_1, \dots, NV_{New}\}$, i.e., they are introduced by the literal L_i. The set of tuples E_{i+1} covered by clause C_{i+i} is the set of ground $(Old + New)$-tuples (instantiations of $\langle OV_1, OV_2, \dots, OV_{Old}, NV_1, \dots, NV_{New}\rangle$) for which the body $L_1, L_2, \dots, L_{i-1}, L_i$ is true. In relational algebra terminology, the new training set E_{i+1} is the natural join of E_i with the relation corresponding to the literal L_i.

The heuristic function used by FOIL is a form of *weighted information gain* where the probability $p(\oplus|C)$ is estimated by using the relative frequency of the positive tuples in the current training set. Let n_i be the number of tuples in E_i, of which $n_i^{\oplus}$ are positive, and let n_{i+1} be the number of tuples in E_{i+1}, of which $n_{i+1}^{\oplus}$ are positive. The information gain obtained by adding the literal L_i to the clause C_i is therefore given by

$$IG_{FOIL}(C_i, C_{i+1}) = \log_2\left(\frac{n_{i+1}^{\oplus}}{n_{i+1}}\right) - \log_2\left(\frac{n_i^{\oplus}}{n_i}\right).$$

Note that each tuple of E_i^+ may correspond to zero, one or more tuples of E_{i+1}. The gain function is weighted by the number $n_i^{\oplus\oplus}$ of positive tuples in E_i that correspond to one or more tuples in E_{i+1}. Thus, the heuristic function is given by

$$WIG_{FOIL}(C_i, C_{i+1}) = n_i^{\oplus\oplus} \times (IG_{FOIL}(C_i, C_{i+1}))$$

In order to deal with noisy datasets, the stopping criteria used by FOIL are heuristic and are based on the *encoding length restriction*, that 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. The number of bits needed to explicitly indicate the $n^{\oplus}(C)$ positive examples covered by a clause C out of the n_{cur} examples in the current training set is

$$ExplicitBits(C, E_{cur}) = \log_2(n_{cur}) + \log_2 \binom{n_{cur}}{n^{\oplus}(C)}$$

The number of bits needed to encode a clause with m literals in the body is computed as

$$ClauseBits(C) = \Sigma_{i=1}^{m}(1 + \log_2(l) + \log_2(V_{q_i})) - \log_2(m!)$$

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.

The construction of a clause is stopped (the necessity stopping criterion is satisfied) when no negative example is covered by the clause or when adding any literal with positive gain would cause $ClauseBits(C)$ to exceed $ExplicitBits(C, E_{cur})$. If there are no more bits available for adding a literal but the clause is still 85% accurate (a threshold chosen ad hoc), then the clause is retained in the hypothesis, otherwise it is discarded.

The construction of a hypothesis stops (the sufficiency stopping criterion is satisfied) when all the positive examples are covered or when no new clause can be generated under the encoding length restriction.

3.4.3 mFOIL

mFOIL [Dže91] extends FOIL approach by adopting specially designed search heuristic and stopping criteria that improve noise-handling. Moreover, it adopts beam-search instead of hill-climbing, and it uses intensional coverage instead of extensional one. Therefore, the background knowledge may contain intensional definitions of predicates.

In place of the weighted information gain used by FOIL, mFOIL adopts an accuracy estimate as the search heuristic, i.e., a clause is evaluated in terms of its accuracy on the training set. The accuracy estimate that is used can be either the *Laplace estimate* or the *m-estimate*.

The Laplace estimate is used in order to improve the reliability of the relative frequency estimate for small training sets: in the extreme case of only one positive example in E_{cur}, the estimate of $p(\oplus|C)$ is 1. This estimate is clearly too optimistic even in the absence of noise. To avoid this problem, the Laplace law of succession was used [NB86]: 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. The Laplace estimate is therefore given by

$$p(\oplus|C) = \frac{n^{\oplus}(C) + 1}{n(C) + 2}$$

In the case in which both $n^{\oplus}(C)$ and $n(C)$ are 0, the probability is $\frac{1}{2}$, which reflects the fact that an empty training set can not alter our a priori assumptions that positive and negative examples have the same probability.

However, this assumption is rarely true in practice. Therefore the m-estimate [Ces90] was introduced that takes into account as well the prior probabilities of the classes:

$$p(\oplus|C) = \frac{n^{\oplus}(C) + m \times p_a(\oplus)}{n(C) + m}$$

where 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 value of m expresses our confidence in the representativeness of the training set. The actual value of m should be set subjectively according to the amount of noise in the examples (larger m for more noise). As m grows towards infinity, the m-estimate approaches the prior probability of the positive class. For $m = \frac{1}{2}$, the m-estimate becomes the Laplace estimate.

In the specialization loop, mFOIL keeps a set of the most promising clauses found so far (the beam) as well as the most *significant* clause found so far. At each step of the loop, the refinements of all the clauses in the beam are generated and evaluated using the search heuristic. The new beam will contain the best refinements that satisfy two conditions: they improve the heuristic function with respect to the clause from which they have been generated and they are *possibly significant*. When no such clause exists, the beam becomes empty and the search terminates. In this case, the best significant clause found so far is retained in the hypothesis if its expected accuracy is better than the default accuracy, given by the probability of the more frequent of the classes $\oplus$ and $\ominus$. This probability is estimated from the entire training set by the relative frequency estimate.

The significance test is based on the *likelihood ratio statistic* [Kal79]. Given a clause C, the likelihood ratio of C is given by

$$\mathit{LikelihoodRatio}(C) = 2n(C) \times \left(p(\oplus|C) \log \left(\frac{p(\oplus|C)}{p_a(\oplus)} \right) + p(\oplus|C) \log \left(\frac{p(\ominus|C)}{p_a(\ominus)} \right) \right)$$

where $n(C)$ are the examples covered by C, $n^{\oplus}(C)$ of which are positive, $p_a(\oplus)$ and $p_a(\ominus)$ are the prior probabilities of classes $\oplus$ and $\ominus$, estimated by the relative frequency of positive and negative examples in the entire training set: $p_a(\oplus) = \frac{n^{\oplus}}{n}$ and $p_a(\ominus) = \frac{n^{\ominus}}{n}$. Moreover, $p(\oplus|C) = \frac{n^{\oplus}(C)}{n(C)}$ is the probability that an example covered by a clause C is positive, and $p(\ominus|C) = 1 - p(\oplus|C)$.

The likelihood ratio statistics is distributed approximately as χ^2 with one degree of freedom. A clause is deemed *significant* if its likelihood ratio is higher than a certain significance threshold. The default value for the threshold is 6.64, that corresponds to a significance level of 99%.

In the specialization loop, clauses are pruned when they are not *possibly significant*, i.e. when none of its refinement can be significant. Consider a clause C that 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 examples and no negative example. In this case, the likelihood ratio statistics would be $-2n^{\oplus}(C) \times \log(p_a(\oplus))$. If this value is less then the significant threshold, no refinement of this clause can be significant and the clause can be pruned.

mFOIL stops adding a clause to the theory when too few positive examples remain for a clause to be *significant* or when no significant clause can be found with expected accuracy greater than the default.

3.4.4 ICL

Inductive Constraints Logic (ICL) [DRL95] is a system that learns from positive and negative interpretations. ICL adopts an algorithm similar to the one of mFOIL, where the coverage by entailment is replaced by coverage by interpretations and the covering loop is performed on the set of negative examples instead of the set of positive ones. It starts with an empty hypothesis H and repeatedly tries to find a clause C to add to the hypothesis H. Each clause found will be true in all positive interpretations and false in some negative ones. The negative interpretations that falsify C are removed from the E^-. This process is repeated until no negative interpretations remains.

Each clause is generated by beam search, starting from the clause $true \leftarrow false$ that is the most specific according to the generality relation for interpretations. Besides a $Beam$ of candidate clauses, ICL keeps as well the best clause found so far ($BestClause$) that is also statistically significant. At each step of the beam search, all the possible refinements Ref of the clauses in the beam are generated by means of a θ-subsumption operator and are evaluated. Depending on the value of the heuristic function, Ref is added to the $Beam$ and/or it becomes the new current $BestClause$.

The heuristic function used for evaluating clauses is given by the probability that an example interpretation is negative, given that clause C is false in the interpretation, i.e. $p(\ominus|\overline{C})$. Notice the difference with the classical accuracy heuristic function of mFOIL where the probability $p(\oplus|C)$ is used. The Laplace estimate is used to measure this probability

$$HV(C) = p(\ominus|\overline{C}) = \frac{n^{\ominus}(\bar{(}C) + 1)}{n(\bar{(}C) + 2)}$$

ICL adopts the same statistical significancy test used by mFOIL to ensure that the clause represents a genuine regularity in the examples and not a regularity due to chance. A clause is significant if its likelihood ratio is higher than a user defined threshold.

Two types of pruning are performed by ICL on the basis of these heuristics. First, a clause C can be pruned if no refinement of C can become better than the best clause at the moment: the best value we can achieve with further refinements of a clause is a clause that is false for the same negative interpretations and true for all the positive. Second, as in mFOIL, ICL stops refining a clause when it is not possibly significant.

The refinement operator adopted in ICL takes into account a declarative bias in order to restrict the search space. The declarative bias is expressed using clause models that define the syntax of the clauses that can appear in hypothesis. The refinement operator adopts these models to generate only the clauses that are allowed by the syntax. The formalism is described in details in [ADRB95, VLDDR94].

Chapter 4

Abductive Reasoning in Learning

4.1 Introduction

As discussed in section 1.1, the problem of learning from an incomplete background knowledge is still an open issue in ILP research. In real world problems, the knowledge acquisition process is often imperfect and some relevant pieces of information may be difficult or impossible to be acquired. Information about specific examples is usually expressed by means of ground facts in the background, therefore the imperfections of the knowledge acquisition process often results in the absence of some background facts. This type of data imperfection is called *incompleteness of the background*. In this case, some positive examples may not be covered due to the absence of some facts related to them in the background. This may require the learning of multiple overspecific rules for covering a set of examples that could otherwise be covered by a single more general one.

This problem can be solved by integrating abductive reasoning into induction by means of a new learning framework called Abductive Concept Learning (ACL). The framework was initially defined in [DK96] and was successively developed in [ELM$^+$96, LMMR97, LMMR98, KR97, KR98]. The present chapter closely follows the treatment given in [KR98]. Abductive Concept Learning is an extension of ILP that allows us to represent both the background and target theories as *abductive logic programs*. Indeed abduction is well-suited for representing problems with incomplete information (see e.g. [PGA87, KM90b, DDS92, Ino94, IS94, KKT97]) able to formulate a variety of such problems in Artificial Intelligence and other areas of Computer Science.

Abductive logic programs are composed of a logic program, a set of abducible predicates and a set of integrity constraints that provide additional information on the abducible predicates by limiting the number of assumptions that are allowed. The incomplete background is represented as an abductive logic program and abduction is used in order to complete the background knowledge by making assumptions about the abducibles. In ACL, also the target program is an abductive logic program that can contain both new rules for the concept(s) to be learned as well as new integrity constraints. Abductive reasoning of abductive logic programs is then used as the basic coverage relation for learning: assumptions about background facts can be made in order to cover examples, thus resulting in more compact theories that can alleviate the problem of overfitting due to the incompleteness in the data.

In this chapter, we will present the basic ACL framework and an algorithm for solving it. ACL provides a principled way to handle incomplete information in learning based on an underlying theory of abduction for knowledge representation.

The central problem of learning abductive theories in ACL contains several useful and interesting subproblems that are of practical relevance. These problems include: (i) concept learning from incomplete background data where some of the background predicates are incompletely specified and (ii) concept learning from incomplete background data together with given integrity constraints that provide some information on the incompleteness of the data.

A specific subcase of these two problems and an important third subproblem is that of (iii) multiple predicate learning, where each predicate is required to be learned from the incomplete data for the other predicates. Here the abductive reasoning can be used to suitably connect and integrate the learning of the different predicates. This can help to overcome some of the non-locality difficulties of multiple predicate learning, such as order-dependence and global consistency of the learned theory.

These subproblems of the full ACL task can be captured in a simpler subproblem of ACL, which we will call ACL1. Within ACL1 we learn only the rule part of an abductive theory but this is sufficiently general in many cases to allow us to address interesting problems as those described above. Apart from its practical relevance, the identification of the ACL1 subproblem is also useful in breaking the full ACL learning task into two separate but strongly inter-related phases of ACL1 and ACL2. ACL1 together with its rules also provides additional input, through abducible assumptions (which are related to the learned rules), to the second phase of ACL2 for learning integrity constraints that can confirm (partly) the correctness of these abducible assumptions. In this way, ACL synthesizes together the two main learning settings of ILP, namely those of learning from entailment [Mug95a, MDR94] and learning from interpretations [DRD94, Fla95].

An algorithm for ACL based on this separation into ACL1 and ACL2 is given. Within ACL1, this algorithm adapts the basic top-down method of ILP to deal with the incompleteness of information and to take into account the use of integrity constraints. It incorporates an abductive proof procedure and other abductive reasoning mechanisms from ALP that are suitably adapted for the context of learning. In the second phase of ACL2, the algorithm takes as input the output of ACL1 and calls on the ICL [DRL95] learner to generate appropriate integrity constraints.

This algorithm has been implemented in a new ILP system also called ACL and ACL1 for its subsystem. Based on these, a separated system for multiple predicate learning, called M-ACL, has also been developed. Suitably adapted heuristics have been used that take into account the incompleteness of information. Several experiments are presented that demonstrate the ability of ACL to learn with incomplete information and its usefulness in multiple predicate learning. ACL has also been applied to problems of analyzing data from market research questionnaires where the available data could be incomplete in several ways.

The problem of learning under incomplete or missing information in an ILP framework has received relatively little attention. Some recent exceptions to this include ICL-Sat [DRD96c] which learns from incomplete interpretations and FOIL-I [IKI$^+$96] which can learn from partial training sets. A recent work with an approach similar to ours for learning the rules of an abductive theory under incomplete information is that of [KK98]. There are also several works [DB92, LDB96] that deal with the related problem of noise in the learning data but this is a different problem where the methods used can not always be applied as effectively to missing information. Most of the machine learning systems that deal with incomplete information are attribute-value learners. An ILP system for learning with incomplete information is LINUS [LDG91a] but, again, it essentially relies on an attribute value representation. In general, these systems adopt different methods to first complete the missing information and then learn from the completed data. In contrast, in ACL the incomplete information is handled dynamically *within the learning process* in a principled way based on an underlying theory of abduction. In this way, ACL combines in a non-trivial way the methods of abduction for dealing with incomplete information with methods of ILP learning.

The chapter is organized as follows. Section 4.2 presents a short review of ALP needed for the formulation and description of the main properties of ACL which are presented in section 4.3. Section 4.4 presents the basic algorithm for ACL and its properties for the single predicate case, while section 4.5 describes the application of ACL to multiple predicate learning. Section 4.6 presents our initial experiments with ACL, section 4.7 discusses related work and section 4.8 concludes the chapter.

4.2 Abductive Logic Programming

In this section we briefly review some of the elements of Abductive Logic Programming (ALP) needed for the formulation of the learning framework of Abductive Concept Learning (ACL). For a more detailed presentation of ALP the reader is referred to the survey [KKT93] (and its recent update [KKT97]) and references therein.

Abductive Logic Programming is an extension of Logic Programming to support abductive reasoning with theories (logic programs) that incompletely describe their problem domain. In ALP this incomplete knowledge is captured (represented) by an abductive theory T. We will consider abductive theories of the following form.

Definition 27 (Abductive theory) *An* **abductive theory T** *in ALP is a triple $\langle P, A, I\rangle$, where P is a definite logic program, A is a set of predicates called* **abducible predicates** *(or simply* **abducibles***), and I is a set of range-restricted clauses called* **integrity constraints**.

For simplicity of presentation we have assumed that the logic program P of an abductive theory is a definite Horn program with no negation (negation as failure) appearing in the body of the rules of P. However, this condition is not restrictive since negation as failure in a non definite logic program can be treated through abduction in an associated abductive theory whose program is definite [EK89].

As a knowledge representation framework, when we represent a problem in ALP via an abductive theory T, we generally assume that the abducible predicates in A carry all the incompleteness of the program P in modelling the external problem domain in the sense that if we (could) complete the information on the abducible predicates in P then P would completely describe the problem domain.

An abductive theory can support abductive (or hypothetical) reasoning for several purposes such as diagnosis, planning or default reasoning. The central notion used for this is that of an *abductive explanation* for an observation or a given goal. Informally, an abductive explanation consists of a set of ground facts (called *abductive assumptions*) on some of the predicates in A which, when added to the program P, make the observation or goal true. The integrity constraints in I must be satisfied by the extension of the program P with such abductive assumptions for these to form an allowed abductive explanation.

To formalize this we need first the notion of *generalized model* of an abductive theory. Generalized models are inspired to *generalized stable models* [KM90b]: since no negation is allowed in the program, the stability condition is not required.

Definition 28 (Generalized model) *Let $T = \langle P, A, I \rangle$ be an abductive theory and Δ a set of ground abducible facts from A. $M(\Delta)$*[1] *is a* **generalized model** *of T iff*

- *$M(\Delta)$ is the minimal Herbrand model of $P \cup \Delta$, and*
- *$M(\Delta)$ is a model of I, i.e., $M(\Delta) \models I$*

We say that Δ is an **abductive extension** *of T.*

Here the semantics of integrity constraints is defined by the second condition in the definition. Their satisfaction requires that they are true statements in the computed model of the extension of the program with Δ for this extension to be allowed. In this case, we say that Δ is *consistent* with the constraints. We will assume that, for any abductive theory, the empty set of abducible assumptions is consistent.

An abductive theory is thus viewed as representing a collection of different allowed states given by the set of its generalized models.

Definition 29 (Abductive explanation) *Let $T = \langle P, A, I \rangle$ be an abductive theory and ϕ any formula*[2] *called an observation (or a query). An* **abductive explanation** *for ϕ in T is any set Δ of abducible facts from A such that*

- *$M(\Delta)$ is a generalized model of T, and*
- *$M(\Delta) \models \phi$.*

Based on this we define a *credulous* form of abductive entailment.

Definition 30 (Abductive entailment) *Let $T = \langle P, A, I \rangle$ be an abductive theory and ϕ any formula. Then, ϕ is* **abductively entailed** *by T, denoted by $T \models_A \phi$, iff there exists an abductive explanation of ϕ in T. If the explanation is Δ, we also write $T \models_A \phi$ with Δ.*

Note that, although the integrity constraints reduce the number of possible explanations for an observation, it is still possible for several explanations that satisfy (do not violate) the integrity constraints to exist. This problem is known as the multiple explanations problem. In order to solve this problem, various criteria can be adopted. We will require the one of minimality (with respect to set inclusion) of the explanations.

The following example illustrates the above ideas.

Example 31 *Consider the following abductive theory $\langle P, A, I \rangle$ with P the logic program on family relations:*

[1] Sometimes we will represent $M(\Delta)$ as $M_P(\Delta)$

[2] In general, ϕ can be any formula but in practice in many cases it suffices for ϕ to be a conjunction of ground facts.

$father(X,Y) \leftarrow parent(X,Y), male(X)$
$mother(X,Y) \leftarrow parent(X,Y), female(X)$
$son(X,Y) \leftarrow parent(Y,X), male(X)$
$daughter(X,Y) \leftarrow parent(Y,X), female(X)$
$child(X,Y) \leftarrow son(X,Y)$
$child(X,Y) \leftarrow daughter(X,Y)$
$loves(X,Y) \leftarrow parent(X,Y)$

the integrity constraint $I = \{\leftarrow male(X), female(X)\}$*, and abducible predicates* $A = \{parent, male, female\}$.
Consider now the observation $O_1 = father(bob, jane)$ *An abductive explanation for* O_1 *is the set* $\Delta_1 = \{parent(bob, jane), male(bob)\}$*. This is the unique minimal explanation. Let now* $O_2 = child(john, mary)$ *be another observation. This has two possible explanations* $\Delta_2 = \{parent(john, mary), male(john)\}$ *and* $\Delta_2^{'} = \{parent(john, mary), female(john)\}$*. If we also knew that* $male(john)$ *holds then* $\Delta_2^{'}$ *would be rejected due to the violation of the integrity constraint. In fact, these two explanations are incompatible with each other.*

We will now introduce the concept of *strong abductive explanation* and we will consider both positive and negative observations. A strong abductive explanations is such that it contains some extra assumptions, with respect to a minimal explanation, that ensure that the addition of further assumptions to it would not result in the violation of the integrity constraints. Strong abductive explanations have the important property that the union of the explanations for two goals is an explanation for the conjunction of them. This property is useful in learning in order to ensure that explanations for different examples will explain their conjunction (see section 4.3). Moreover, we will also define strong abductive explanations for negative observations that will be required in learning for the explanation of negative examples (see section 4.3).

In order to obtain the properties of strong abductive explanations, we need to be able to make explicitly negative assumptions. This is obtained by considering, for each abducible predicate $abd(X)$, a new abducible predicate $not\_abd(X)$ and that is related to $abd(X)$ by means of the constraint $\leftarrow abd(X), not\_abd(X)$. The addition of abducible predicates expressing falsity allows to define a new semantics for abductive theory, called *three-valued generalized model*, where abducible atoms can be true, false or undefined, differently from generalized models where all the abducible facts not in the model are considered as false. This extension is needed since, when dealing from incomplete information, we want to represent the fact that some abducible atoms are certainly true, some are certainly false and we can not say anything about the rest, due to lack of information. The addition of negative abducibles and of the relative constraints is obtained by means of the following theory transformation.

Definition 32 (Three-valued Version of a Theory) *Given an abductive theory* $T = \langle P, A, I\rangle$*, the* **three-valued version** *of* T *is the theory* $T^* = \langle P, A \cup A^*, I \cup I^*\rangle$ *where, for each predicate* $a \in A$*,* A^* *contains the new predicate symbol* not_a *and* I^* *contains the denial* $\leftarrow a(\vec{X}), not\_a(\vec{X})$.

We define the *complement* $\overline{l}$ of an abducible literal l as

$$\overline{l} = \begin{cases} not\_a(\vec{X}) & \text{if } l = a(\vec{X}) \\ a(\vec{X}) & \text{if } l = not\_a(\vec{X}) \end{cases}$$

Given the three-valued version $T^* = \langle P, A \cup A^*, I \cup I^* \rangle$ of an abductive theory, a set of assumptions Δ from predicates of $A \cup A^*$ is called *self-consistent* if and only if it does not contain both a literal and its complement, i.e., iff $\Delta^* \models I^*$;

The previous definition of generalized model is extended to the following.

Definition 33 (Three-valued generalized model) *Let $T = \langle P, A, I \rangle$, be an abductive theory with $T^* = \langle P, A \cup A^*, I \cup I^* \rangle$ its three-valued version and let Δ^* be a set of ground abducible facts from $A \cup A^*$. $M(\Delta^*)$ is a* **three-valued generalized model** *of T iff*

- *Δ^* is self-consistent;*
- *$M(\Delta^*)$ is the minimal Herbrand model of $P \cup \Delta^*$;*
- *$M(\Delta^*) \models I$.*

A set Δ^* is an **abductive explanation** *for a formula ϕ if and only if $M(\Delta^*)$ is a three-valued generalized model and $M(\Delta^*) \models \phi$.*

In a three-valued generalized model $M(\Delta^*)$ of T, an abducible fact $a(\overline{c})$ is assumed true if $a(\overline{c}) \in \Delta^*$, is assumed false if $not\_a(\overline{c}) \in \Delta^*$ and is undefined otherwise.

From this point onwards, unless otherwise specified, we will consider abductive theories in their three-valued version. Therefore, when we write $T_1 = \langle P_1, A_1, I_1 \rangle$, we mean the three-valued version of a theory $T = \langle P, A, I \rangle$, with $P_1 = P$, $A_1 = A \cup A^*$ and $I_1 = I \cup I^*$. Also when we refer to a generalized model we will mean a three-valued generalized model.

We can now define the notion of strong abductive explanation.

Definition 34 (Strong abductive explanation) *Let $T = \langle P, A, I \rangle$ be an abductive theory, $T^* = \langle P, A \cup A^*, I \cup I^* \rangle$ its three-valued version and O a ground atomic fact called an observation (or a goal). A* **strong abductive explanation** *for O in T is any set Δ^* of abducible facts from $A \cup A^*$ such that*

- *$M(\Delta^*)$ is a generalized model of T^* such that $M(\Delta^*) \models \phi$, and*
- *for any $\Delta^{'} \subseteq A \cup A^*$, if $M(\Delta^{'}) \models I$ and $\Delta^{'} \cup \Delta^*$ is self-consistent, then $M(\Delta^{'} \cup \Delta^*) \models I$.*

The latter condition can be intuitively expressed in this way: Δ^* must be such that any other abductive extension Δ' that is self-consistent with Δ^* can be added to Δ^* without violating the integrity constraints. We say that $M(\Delta^*)$ is a **strong generalized model** *for T and that Δ^* is a* **strong abductive extension** *of T*[3].

In the case of example 31, a strong abductive explanation for $O_1 = father(bob, jane)$ would be $\Delta_1^* = \{parent(bob, jane), male(bob), not\_female(bob)\}$. Now, the assumption $female(bob)$, that would violate the integrity constraints, can not be self-consistently added to Δ_1^*.

We will now give the definition of a strong abductive explanation for a negative observation not_O. In this case we want an explanation that can not be extended in order to explain O.

[3]Note that under our previous assumption the empty set is always consistent for any abductive theory; this means that it has always a strong abductive extension.

Definition 35 (Strong abductive explanation of negative observations) *Consider an abductive theory $T = \langle P, A, I \rangle$ and let $T^* = \langle P, A \cup A^*, I' \cup I^* \rangle$ be its three-valued version and let a negative observation (or a goal), denoted by not_O, be given. A* **strong abductive explanation** *for not_O is any set Δ^* of abducible facts from $A \cup A^*$ such that*

- *$M(\Delta^*)$ is a strong generalized model of T and $M(\Delta^*) \not\models O$*
- *for any $\Delta' \subset A \cup A^*$, if Δ' is an abductive explanation of O then $\Delta' \cup \Delta^*$ is not self-consistent.*

In this case we say that not_O is abductively entailed by T and denote this by $T \models_A$ not_O with Δ^*.

Hence $O \notin M(\Delta^*)$ and more importantly Δ^* cannot be consistently extended to derive O. The strong abductive explanation is thus a set of sufficient assumptions which, when adopted, ensures that O can not be abductively entailed in a way that would be self-consistent with these assumptions. The second condition in the definition of strong abductive explanation corresponds to the *admissibility condition* introduced by Dung [Dun91] in its definition of the preferential semantics for normal logic programs.

In order to illustrate this, consider again example 31 and consider the negative observation that *not_father*(*jane*, *john*). A strong abductive explanation for this observation is $\Delta_1^* = \{not\_parent(jane, john)\}$ or $\Delta_2^* = \{not\_male(jane)\}$ since *father*(*jane*, *john*) can not be derived by any self-consistent extension of either of these sets. In contrast, the empty explanation is an abductive explanation for *not_father*(*jane*, *john*) since $father(jane, john) \notin M(\emptyset)$ but this explanation is not strong since it can be consistently extended with $\Delta' = \{parent(jane, john), male(jane)\}$ to derive $father(jane, john)$.

The strongness of an explanation Δ^* for a negative literal *not_e* means that it invalidates every possible explanation for e. This is expressed by the following property, that is a direct consequence of the definition of strong abductive explanation.

Property 36 *Given an abductive theory T and an atom e, it holds*

$$T \models_A not\_e \text{ with } \Delta^* \Rightarrow \forall \Delta^+ : T \models_A e \text{ with } \Delta^+, \quad \exists \bar{l} \in \Delta^* : l \in \Delta^+$$

In other words, a strong abductive explanation Δ^* for not_e contains the complement of (at least) one assumption from every possible explanation Δ^+ of e.

The following example illustrate the above property.

Example 37 *Consider the following abductive theory $T = \langle P, A, I \rangle$*

$$\begin{aligned} P = \{&sibling(X, Y) \leftarrow brother(X, Y),\\ &sibling(X, Y) \leftarrow sister(X, Y)\}\\ I = \{&\}\\ A = \{&brother, sister\} \end{aligned}$$

and the observation $O = sibling(bob, jane)$. The strong abductive explanations for not_O is $\Delta^* = \{not\_sister(bob, jane), not\_brother(bob, jane)\}$, while the explanations for O are $\Delta_1^+ = \{sister(bob, jane)\}$ and $\Delta_2^+ = \{brother(bob, jane)\}$: Δ^* contains the complement of a literal from both Δ_1^+ and Δ_2^+.

The definition of strong abductive explanation can be generalized for a conjunction of positive and negative observations $C = O_1 \wedge \ldots \wedge O_m \wedge not\_O_1 \wedge \ldots \wedge not\_O_n$. A **strong abductive explanation** for the conjunction is any set Δ^* of abducible facts from $A \cup A^*$ such that Δ^* is a strong abductive explanation for every conjunct taken singularly.

The strong abductive explanation for the conjunction of two observations $O_1 \wedge O_2$ can be obtained by taking the union of the strong abductive explanations for O_1 and O_2.

Proposition 38 *Let $T = \langle P, A, I \rangle$ be an abductive theory in its three-valued version and let Δ_1 and Δ_2 be two strong abductive explanations of, respectively, G_1 and G_2, where G_1 and G_2 can be either positive or negative goals. If $\Delta_1 \cup \Delta_2$ is self-consistent, then $\Delta_1 \cup \Delta_2$ is a strong abductive explanation for $G_1 \wedge G_2$.*

Proof: See appendix A.2. □

This property thus allows us to combine together explanations of different observations effectively reducing the consistency requirement with respect to the integrity constraints of the theory to the simpler requirement of self-consistency of this union of explanations. This is important for computational reasons when we have a collection of different (positive and negative) observations to consider together.

As we will see in the next sections, in the Abductive Concept Learning framework and system, deductive entailment is replaced by the abductive entailment as the coverage relation. Thus the deductive SLD (and SLDNF) proof procedures of Logic Programming are replaced by abductive proof procedures [EK89, KM90a, KM90c, DDS92, SI92] of ALP. Any abductive procedure satisfying the following notion of abductive derivability is suitable.

Definition 39 (Abductive derivability) *Given an abductive theory $T = \langle P, A, I \rangle$, a goal G and an initial strong abductive explanation Δ_i, we say that a procedure* abductively derives *G from T if it returns a set of assumptions Δ_G such that Δ_G is a strong abductive explanation of G and $\Delta_G \cup \Delta_i$ is consistent, i.e., $M(\Delta_G \cup \Delta_i) \models I$. In this case, we write $T \vdash_{\Delta_i}^{\Delta_G} G$.*

For our study of Abductive Concept Learning we will employ an abductive proof procedure based on the one in [KM90c] (reported in Appendix A.3). This procedure has been modified according to the notion of derivability defined above to return the full set of assumptions Δ_G required to explain G irrespective of the fact that some of these may already be present in Δ_i. The proof procedure interleaves phases of *abductive* and *consistency derivations*. Intuitively, an abductive derivation is the standard Logic Programming derivation suitably extended in order to consider abducibles. When an abducible atom δ is encountered, it is added to the current set of assumptions (if it is not already there). The addition of δ must not result in a violation of the integrity constraints. To this purpose, a consistency derivation for δ is initiated to check this. Each integrity constraint is resolved against δ and it is verified that every resulting goal fails. In the consistency derivation, when a new abducible is encountered in one of these reduced goals, an abductive derivation for its complement is started in order to ensure the failure of this abducible. This subsidiary abductive derivation will often result in additional assumptions in the explanation set.

The modified version of the procedure which we will use is sound with respect to the notion of (strong) abductive derivability above for the case in which the integrity constraints are restricted to be denials with at least one abducible appearing explicitly in the body of the denial. This result follows directly from the soundness of the original procedure in

[KM90c] which computes strong explanations. The more general case of integrity constraints in the form of range restricted clauses, $A_1; \ldots ; A_k \leftarrow B_1, \ldots, B_m$ can be dealt with in the following way. The constraints are first transformed into their equivalent denial form $\leftarrow B_1, \ldots, B_m, \sim A_1, \ldots, \sim A_k$ and then classical negation is approximated by negation by default obtaining $\leftarrow B_1, \ldots, B_m, not\_A_1, \ldots, not\_A_k$ that can be processed by the abductive proof procedure. This transformation is similar to one into the three-valued version of the theory: the literals $not\_A_1, \ldots, not\_A_k$ are new positive abducible literals and constraints $\leftarrow A_i, not\_A_i$ are added to the set of constraints.

4.3 Learning with Abduction

We will now restate the problem of learning from entailment for the case in which both the background knowledge and the learned theory are abductive theories in their three-valued version. The following restrictions on the language of the hypothesis and of the background are considered[4].

- The background knowledge $T = \langle P, A, I \rangle$ does not contain any target predicate(s) neither in the program P nor in the integrity constraints I. The empty set of abducible assumptions is a consistent abductive extension of T.
- The integrity constraints are range-restricted clauses $A_1; \ldots ; A_k \leftarrow B_1, \ldots, B_m$, with at least one of $B_1, \ldots, B_m$ abducible. Also, for each A_j in the head of the clause its definition, in the program P of the background theory, does not depend on abducibles, namely A_j is not abducible and recursively none of the conditions in the rules of P for A_j is abducible.

The language of the examples is simply that of atomic ground facts on the target predicate(s).

Definition 40 (Abductive Concept Learning)
Given

- *a hypothesis space $\mathcal{T} = \langle \mathcal{P}, \mathcal{I} \rangle$ consisting of a space of possible programs $\mathcal{P}$ and a space of possible constraints $\mathcal{I}$ satisfying the language restrictions given above except that now a possible program can contain the target predicate(s).*
- *a set of positive examples E^+,*
- *a set of negative examples E^-,*
- *an abductive theory $T = \langle P, A, I \rangle$ as background theory,*

Find

A set of rules $P' \in \mathcal{P}$ and a set of constraints $I' \in \mathcal{I}$ such that the new abductive theory $T' = \langle P \cup P', A, I \cup I' \rangle$ satisfies the following conditions

- $T' \models_A E^+$,
- $\forall e^- \in E^-, T' \not\models_A e^-$.

[4]These language restrictions are not necessary for the definition of the ACL problem but rather are needed for the development of the algorithms to solve this problem.

where E^+ stands for the conjunction of all positive examples.

We say that an individual example e is covered *by a theory T' if and only if $T' \models_A e$.*

In effect, we have replaced the deductive entailment in the ILP problem with abductive entailment to define the ACL learning problem.

The fact that the conjunction of positive examples must be entailed means that, for every positive example, there must exist an abductive explanation and the explanations for all the positive examples must be consistent with each other. For negative examples, it is required that no abductive explanation exists for any of them. Abductive Concept Learning can be illustrated as follows.

Example 41 *Suppose we want to learn the concept father. Let the background theory be $T = \langle P, A, \emptyset \rangle$ where:*

$P = \{parent(john, mary), male(john),$
$parent(david, steve),$
$parent(kathy, ellen), female(kathy)\}$
$A = \{male, female\}$.

Let the training examples be:

$E^+ = \{father(john, mary), father(david, steve)\}$
$E^- = \{father(kathy, ellen), father(john, steve)\}$

In this case, a possible hypotheses $T' = \langle P \cup P', A, I' \rangle$ learned by ACL would consist of

$P' = \{father(X, Y) \leftarrow parent(X, Y), male(X)\}$
$I' = \{ \leftarrow male(X), female(X)\}$

This hypothesis satisfies the definition of ACL because:

1. $T' \models_A father(john, mary), father(david, steve)$ *with* $\Delta = \{male(david)\}$,

2. $T' \not\models_A father(kathy, ellen)$, *as the only possible explanation for this goal, namely* $\{male(kathy)\}$ *is made inconsistent by the learned integrity constraint in I'.*

3. $T' \not\models_A father(john, steve)$, *as this has no possible abductive explanations.*

Hence, despite the fact that the background theory is incomplete (in its abducible predicates), ACL can find an appropriate solution to the learning problem by suitably extending the background theory with abducible assumptions. Note that the learned theory without the integrity constraint would not satisfy the definition of ACL, because there would exist an abductive explanation for the negative example $father(kathy, ellen)$, namely $\Delta^- = \{male(kathy)\}$. This explanation is prohibited in the complete theory by the learned constraint together with the fact $female(kathy)$.

It is important to note that the treatment of positive and negative examples in ACL is asymmetric with respect to the existence of abductive explanations. For positive examples, it is sufficient that there exists one explanation for the conjunction of all positive examples that is consistent with the constraints, whereas, for each negative example, all possible explanations must be made inconsistent by the constraints.

In order to achieve this, we require the existence of a **strong** abductive explanation for the (complement of) negative examples. Adding these strong abductive explanations to the background theory then ensures that no negative example can be abductively explained. In the example above, the negative example $father(kathy, ellen)$ can be covered by adding the strong abductive explanation $\Delta^* = \{not\_male(kathy)\}$ for $not\_father(kathy, ellen)$ to the theory. This is sufficient to ensure that this negative example can no longer be abductively entailed even in the absence of any integrity constraints in I'. Moreover, these strong abductive explanations can suggest what new integrity constraints can be learned in I' so that the negative examples will not be covered.

This observation suggests a natural way in which the full ACL problem can be split into two subproblems: (1) learning the rules together with appropriate explanations and strong explanations and (2) learning integrity constraints. We will see that the solutions of the two subproblems can be combined to obtain a solution for the original problem.

The first subproblem, called ACL1, has the following definition.

Definition 42 (ACL1)
Given

- *a set of positive examples* E^+,
- *a set of negative examples* E^-,
- *an abductive theory* $T = \langle P, A, I\rangle$ *as background theory,*
- *a hypothesis space of possible programs* $\mathcal{P}$.

Find

A set of rules $P' \in \mathcal{P}$ *such that the new abductive theory* $T_{ACL1} = \langle P \cup P', A, I\rangle$ *satisfies the following conditions*

- $T_{ACL1} \models_A E^+$ *with* Δ^+,
- $T_{ACL1} \models_A not\_E^-$ *with* Δ^-,
- $\Delta^+ \cup \Delta^-$ *is self-consistent.*

where not_E^- *stands for the conjunction of the complement of every negative example.*

We say that a theory T **ACL1-covers** *an individual positive example* e^+ *iff* $T \models_A e^+$ *and that* T **does not ACL1-cover** e^+ *if and only if* $T \not\models_A e^+$.

If $T \models_A e^+$ *with* $\Delta = \emptyset$, *then we say that* e^+ *is* **ACL1-covered without abduction**, *otherwise we say that it is* **ACL1-covered with abduction**.

For negative examples, we say that a theory T **ACL1-uncovers** *an individual negative example* e^- *iff* $T \models_A not\_e^-$ *and that* T **does not ACL1-uncover** e^- *iff* $T \not\models_A not\_e^-$.

If $T \models_A not\_e^-$ *with* $\Delta = \emptyset$, *then we say that* e^- *is* **ACL1-uncovered without abduction**, *otherwise we say that it is* **ACL1-uncovered with abduction**.

ACL1 and ACL differ only in their requirements on negative examples. ACL1 requires that in the learned theory there must be a strong abductive explanation for the complement of every negative example. Indeed, this is weaker than the condition required by the full ACL problem which is that every negative example is false in all the abductive extensions of the theory.

Nevertheless, the information generated by ACL1 through the strong abductive explanations for negative examples can be used to provide a solution of the full ACL problem through a second learning phase. From the output of ACL1, i.e., its set of rules and the sets of assumptions Δ^+ and Δ^- for covering positive examples and uncovering negative ones, a solution to ACL can be found by learning constraints that are consistent with Δ^+ and inconsistent with the complement of every abducible in Δ^-. In fact, the strong abductive explanation Δ^- will contain, for every negative example e^-, a strong abductive explanation Δ_{e^-} for not_e^-. This explanation, according to property 36, contains assumptions that would invalidate directly any possible abductive explanation of e^-. Hence by making all the complements of assumptions in Δ^- inconsistent, we make all possible explanations of every e^- inconsistent.

Thus the definition of the second subproblem, called ACL2, can be given as follows.

Definition 43 (ACL2)
Given

- *a solution of ACL1*
  - $T_{ACL1} = \langle P \cup P', A, I \rangle$,
  - Δ^+,
  - Δ^-,
- *a hypothesis space of possible constraints $\mathcal{I}$ satisfying the same requirements as in ACL.*

Find

A set of constraints $I' \in \mathcal{I}$ such that the new abductive theory $T' = \langle P \cup P', A, I \cup I' \rangle$ satisfies the following condition

- $M_{P \cup P'}(\Delta^+) \models I'$,
- $\forall l \in \Delta^-$, $M_{P \cup P'}(\{\bar{l}\}) \not\models I'$.

Note that the third condition of ACL1 requiring $\Delta^+ \cup \Delta^-$ to be self-consistent helps to avoid the case of posing an empty ACL2 problem. If this cannot be satisfied, i.e., $\Delta^+ \cup \Delta^-$ is not self-consistent, then the corresponding ACL2 problem cannot have any solutions.

The theory $T' = \langle P \cup P', A, I \cup I' \rangle$, obtained by combining the solutions of the two subproblems, gives a solution to the full ACL problem.

Theorem 44 *Let $T_{ACL1} = \langle P \cup P', A, I \rangle$, Δ^+ and Δ^- be the solution of ACL1 given training sets E^+ and E^-, background theory $T = \langle P, A, I \rangle$ and space of possible programs $\mathcal{P}$. Moreover, let $T' = \langle P \cup P', A, I \cup I' \rangle$ be the solution to ACL2 given the previous solution of ACL1 and hypothesis space $\mathcal{I}$. Then T' is a solution to the ACL problem that has E^+ and E^- as training sets, T as background theory and $\mathcal{P}$ and $\mathcal{I}$ as spaces of possible programs and constraints.*

The proof of this theorem is reported in Appendix A.1. Once decomposed into its two subproblems, it becomes clear that ACL combines the two ILP settings of learning from entailment and learning from interpretations. In fact, ACL1 can be seen as a problem of learning from entailment, while ACL2 as a problem of learning from interpretations.

The algorithm we present in the next section solves the ACL problem by first solving ACL1 and then ACL2. In example 41, the solution of ACL1 consists of the rule $father(X,Y) \leftarrow parent(X,Y), male(X)$, together with the explanations $\Delta^+ = \{male(david), not\_female(david)\}$ and $\Delta^- = \{not\_male(kathy)\}$. Given this intermediate solution, we can now apply a second phase where integrity constraints are learned from the background knowledge and the explanations obtained in the first phase. We want to make $male(kathy)$ inconsistent while keeping Δ^+ consistent: $\leftarrow male(X), female(X)$ is a constraint that satisfies these conditions.

We note that in many cases, ACL1 can be useful on its own merit e.g., when we have sufficient information in the integrity constraints of the background theory or for problems where indeed this weaker requirement on negative examples is sufficient. We will see examples of such cases in the following sections 5 and 6.

4.3.1 Monotonicity and Generality

Abductive Logic Programs are inherently non-monotonic. Given two abductive theories $T_1 = \langle P_1, A, I_1 \rangle$ and $T_2 = \langle P_2, A, I_2 \rangle$ both entailing a goal, their union $T = \langle P_1 \cup P_2, A, I_1 \cup I_2 \rangle$ does not necessarily entail this goal. Non-monotonicity poses problems in learning as algorithms based on the covering approach can not be used. In general, we can not learn a theory by iteratively adding a clause to a partial hypothesis because the addition of a clause can reduce the number of positive examples covered by the hypothesis by making some of the abductive assumptions inconsistent.

By splitting the ACL problem into the two phases of ACL1 and ACL2, we can recover a form of restricted monotonicity. In the first phase of ACL1, where the integrity constraints remain fixed, we have two cases to consider: (i) monotonicity under the addition of a new clause in the program P of the current hypothesis and (ii) monotonicity under the addition of new abductive assumption as we move from one training example to another. The second case can be dealt with by employing a suitable abductive proof procedure for ALP based on strong abductive explanations, as discussed in the previous section, carrying the explanation of the previous examples when testing the abductive coverage (or uncoverage if the example is negative) of the next example. The first case is in general more difficult but in the particular case of interest, since the new (learned) clauses can only affect the extension of the target(s) predicates, we can satisfy this monotonicity requirement by restricting (as we have) the language of the integrity constraints and the program of the background theory to be independent of the target predicate(s).

In the second phase of ACL2, where the program of the abductive hypothesis is fixed and we vary the integrity constraints, monotonicity in ensured by the specific definition of the ACL2 problem that we have adopted where, by construction, the new learned integrity constraints must be consistent with the abductive assumptions Δ^+ required for the coverage of the positive examples. Hence these examples will continue to be abductively entailed by the theory after the addition of the new integrity constraints generated by ACL2.

The non-monotonic nature of the hypothesis space of abductive theories introduces another difficulty in the task of solving the ACL problem. It makes it difficult to have a generality structure on this space that can be useful in the search for solutions to our learning problem. In general, there is no natural generality structure on the full space of abductive theories but again the separation of the problem into its two phases of ACL1 and

ACL2 allows us to adopt the separate generality relations for the rule part P and integrity constraints I of the abductive theories.

Let us recall here the definitions of the generality relations. For the rule part (see section 3.2.4), we have that P_1 is more general or equally general as P_2 if and only if $P_1 \models P_2$, while, for the constraints part (see section 3.3), I_1 is more general or equally general as I_2 if and only if $I_2 \models I_1$.

The use of these usual generality relations on the separate parts of an abductive theory means, as we shall see in the next section, that we can adapt standard ILP techniques, e.g., generalization and specialization operators based on θ-subsumption [Plo70, Plo71], in developing algorithms for the separate phases of ACL1 and ACL2.

4.4 An Algorithm for ACL

The ACL problem can be solved by the following algorithm, also called ACL. The algorithm is composed of two steps, one for each of the subproblems of the full ACL problem.

Algorithm ACL:

1. **Learn rules** (ACL1): find a set of rules P' and two sets of assumptions Δ^+ and Δ^- such that

   - $\langle P \cup P', A, I \rangle \vdash_{\emptyset}^{\Delta^+} E^+$,
   - $\langle P \cup P', A, I \rangle \vdash_{\Delta^+}^{\Delta^-} not\_E^-$

   where $\vdash_{\Delta}^{\Delta'}$ denotes an abductive derivability satisfying definition 39. The requirement that Δ^+ is given as input for the abductive derivation of negative examples ensure the third condition of the ACL1 problem definition that requires the consistency among Δ^+ and Δ^-.

2. **Learn constraints** (ACL2): find a set of integrity constraints I' such that

   - $M(\Delta^+) \models I'$,
   - $\forall l \in \Delta^-$, $M(\{\bar{l}\}) \not\models I'$

   where $M(\Delta^+)$ and $M(\{\bar{l}\})$ denote the minimal Herbrand model of $P \cup P' \cup \Delta^+$ and $P \cup P' \cup \{\bar{l}\}$ respectively.

ACL1 is solved by an algorithm also called ACL1 that will be presented in section 4.4.1. Note that this algorithm uses strong abductive explanations for the positive examples E^+ (as well as the negative examples) thus exploiting the property of proposition 38 for combining separate explanations and in particular for ensuring that the union $\Delta^+ \cup \Delta^-$ of the computed assumptions is consistent with the learned theory. ACL2 can be solved by employing a system that learns from positive and negative interpretations, such as ICL [DRL95]. We will explain in more detail how ICL can be applied in section 4.4.2.

algorithm ACL1(
  inputs : E^+, E^- : training sets,
    $T = \langle P, A, I \rangle$: background abductive theory,
  outputs : H : learned theory, Δ^+, Δ^- : abduced literals)

$H := \emptyset$
$\Delta^+ := \emptyset$
$\Delta^- := \emptyset$
repeat
  Specialize($T, H, E^+, E^-, \Delta^+, \Delta^-; Rule, E^+_{Rule}, \Delta^+_{Rule}, \Delta^-_{Rule}$)
  $E^+ := E^+ \setminus E^+_{Rule}$
  $H := H \cup \{Rule\}$
  $\Delta^+ := \Delta^+ \cup \Delta^+_{Rule}$
  $\Delta^- := \Delta^- \cup \Delta^-_{Rule}$
until $E^+ = \emptyset$ (*sufficiency stopping criterion*)
output H, Δ

Figure 4.1: ACL1, the covering loop

4.4.1 An Algorithm for ACL1

The algorithm for ACL1 is based on the generic top-down ILP algorithm (see section 3.2.6) and extends the algorithm in [ELM+96]. In this paragraph, we consider only the single predicate learning task. We will discuss in section 4.5 the problem of learning multiple predicates.

The top level covering and specialization loops of the algorithm are shown in figure 4.1 and figure 4.2 respectively. The generic top-down algorithm has been extended in several ways to take into account the abductive coverage relation of ACL1.

New clauses are generated by beam search, initialized to a clause with an empty body for the target predicate, using a specially defined heuristic evaluation function. This is adapted from the usual accuracy function to allow for the possibility of missing information on some of the background predicates.

The evaluation of a clause is done by starting an abductive derivation for each positive example and for the complement of each negative example (see figure 4.3). The derivation is performed using a procedure based on the abductive procedure outlined in Appendix A.3. For each example e we have a call AbductiveDerivation($e, \langle P \cup H \cup \{Rule\}, A, I\rangle, \Delta_{in}; \Delta_e$). This returns a strong abductive explanation Δ_e for the goal e (which is either of the form e^+ or not_e^-) starting from an initial set of assumptions Δ_{in}, i.e., $\langle P \cup H \cup \{Rule\}, A, I\rangle \vdash^{\Delta_e}_{\Delta_{in}} e$. Δ_{in} consists of the set of assumptions abduced for earlier examples thus ensuring that the assumptions made during the derivation of the current example are consistent with the ones made before. Note that Δ_e contains all the assumptions needed to explain e, even those that are already contained in Δ_{in}. This is needed for the evaluation of the heuristic value of the clause as well as for the second phase (ACL2) of the ACL algorithm, where we learn the constraints, as the learned constraints must make inconsistent all the assumptions in the explanations $\Delta_{not\_e^-}$ of any negative example e^-.

procedure Specialize(
  inputs : T : background theory,
    H : current hypothesis, E^+, E^- : training sets,
    Δ^+, Δ^- : current set of abduced literals
  outputs : $Best$: rule, E^+_{Best} : examples covered by $Best$,
  $\Delta^+_{Best}, \Delta^-_{Best}$: literals abduced when testing $Best$)

$Beam := \{ \langle p(X) \leftarrow true., Value \rangle$, where p is a target predicate,
  $Value$ is the value of the heuristic function for the rule$\}$
Select and remove the best rule $Best$ from $Beam$
repeat
  $BestRefinements :=$ set of refinements of $Best$ allowed
    by the language bias
  for all $Rule \in BestRefinements$ **do**
    $Value :=$ Evaluate$(Rule, T, H, E^+, E^-, \Delta^+, \Delta^-)$
    if $Rule$ covers at least one pos. ex. **then**
      add $\langle Rule, Value \rangle$ to $Beam$
  endfor
  Remove rules in $Beam$ exceeding the beam size
  Select and remove the best rule $Best$ from $Beam$
until $Best$ uncovers every $e^- \in E^-$ (*necessity stopping criterion*)
Test the coverage of $Best$ obtaining:
  E^+_{Best} the set of positive examples covered by $Best$
  Δ^+_{Best} and Δ^-_{Best} the sets of literals abduced during
  the derivation of e^+ and not_e^- $(e^+ \in E^+_{Best}, e^- \in E^-)$
output $Best, E^+_{Best}, \Delta^+_{Best}, \Delta^-_{Best}$

Figure 4.2: ACL1, the specialization loop

function Evaluate(
    inputs : Rule: rule, $T = \langle P, A, I\rangle$: background theory,
        H : current hypothesis, E^+, E^- : training sets,
        Δ^+, Δ^- : current sets of abduced literals)
    returns the value of the heuristic function for *Rule*

$n^{\oplus} := 0$, number of pos. ex. ACL1-covered by *Rule* without abduction
$n^{\oplus}_A := 0$, number of pos. ex. ACL1-covered by *Rule* with abduction
$n^{\ominus} := 0$, number of neg. ex. not ACL1-uncovered by *Rule*
$n^{\ominus}_A := 0$, number of neg. ex. ACL1-uncovered by *Rule* with abduction
$\Delta_{in} := \Delta^+ \cup \Delta^-$
for each $e^+ \in E^+$ **do**
    if AbductiveDerivation($e^+, \langle P \cup H \cup \{Rule\}, A, I\rangle, \Delta_{in}; \Delta_{e^+}$)
        succeeds **then**
        if $\Delta_{e^+} = \emptyset$ **then**
            increment $n^{\oplus}$
        else
            increment $n^{\oplus}_A$
        endif
        $\Delta_{in} := \Delta_{in} \cup \Delta_{e^+}$
    endif
endfor
for each $e^- \in E^-$ **do**
    if AbductiveDerivation($not\_e^-, \langle P \cup H \cup \{Rule\}, A, I\rangle, \Delta_{in}; \Delta_{e^-}$)
        succeeds **then**
        if $\Delta_{e^-} \neq \emptyset$ **then**
            increment $n^{\ominus}_A$
        endif
        $\Delta_{in} := \Delta_{in} \cup \Delta_{e^-}$
    else
        increment $n^{\ominus}$
    endif
endfor
return Heuristic($n^{\oplus}, n^{\oplus}_A, n^{\ominus}, n^{\ominus}_A$)

Figure 4.3: ACL1, evaluation of a clause

Covered positive and negative examples are counted, distinguishing between examples covered (uncovered) with or without abduction, and these numbers are used to calculate the heuristic function. This heuristic function of a clause (or rule) C takes the form of an *expected classification accuracy* (see section 3.2.6): $A(C) = p(\oplus|C)$, where $p(\oplus|C)$ is defined as the probability that an example covered by C is positive. The probability is estimated by means of a form of relative frequency that gives different strength to positive examples covered (negative examples uncovered) with assumptions (i.e. $T \vdash^{\Delta}_{\Delta_{in}} e$ with $\Delta \neq \emptyset$, where e is either e^+ or not_e^-) or without assumptions (i.e. $\Delta = \emptyset$).

The heuristic function used is

$$A(C) = \frac{n^{\oplus} + k^{\oplus} \times n^{\oplus}_{A}}{n^{\oplus} + n^{\ominus} + k^{\oplus} \times n^{\oplus}_{A} + k^{\ominus} \times n^{\ominus}_{A}}$$

where, for any given clause C, $n^{\oplus}, n^{\oplus}_{A}, n^{\ominus}, n^{\ominus}_{A}$ are defined as in figure 4.3 according to the abductive coverage of positive and negative examples by clause C.

The coefficients $k^{\oplus}$ and $k^{\ominus}$ are introduced in order to take into account the degree of confidence in the assumptions made, respectively, for positive and negative examples. They are an estimate of the fraction of assumptions made that are correct. For example, consider a clause C of the form:

$p(X) \leftarrow Body(X)$

where $Body(X)$ is a conjunction of literals not containing an abducible. Suppose we want to evaluate the refinement C' obtained by adding to C the abducible literal $abd(X)$. Clause C covers $n^{\oplus}(C)$ positive examples without abduction: out of these, C' will cover $n^{\oplus}(C')$ positive examples without abduction (for which a fact of the form $abd(\vec{t})$ is in the background program), $n^{\oplus}_{A}(C')$ with abduction (a fact of the form $abd(\vec{t})$ is abduced) and it will not cover $n^{\oplus}(C) - n^{\oplus}(C') - n^{\oplus}_{A}(C')$ examples ($abd(\vec{t})$ could not be abduced because of constraints). $k^{\oplus}(C')$ expresses an estimate of the fraction of the $abd(\vec{t})$ assumptions that are correct in the sense that, if the knowledge were complete, $abd(\vec{t})$ would be known to be true. This percentage is estimated by assuming that the ratio of true facts over the total number of facts for the unknown atoms is the same for the known atoms. Therefore $k^{\oplus}(C')$ is given by the following formula

$$k^{\oplus}(C') = \frac{\#\text{ of true atoms}}{\#\text{ of known atoms}} = \frac{n^{\oplus}(C')}{n^{\oplus}(C) - n^{\oplus}_{A}(C')}$$

The true atoms are the facts (in the background program) of the form $abd(\vec{t})$ that corresponds to examples covered by C', therefore their number is $n^{\oplus}(C')$. The false atoms are the ones for which the constraints inhibited the assumption of a fact of the form $abd(\vec{t})$. The unknown atoms are those for which it was possible to make an assumption of the form $abd(\vec{t})$, therefore their number is $n^{\oplus}_{A}(C')$. The number of known atoms is given by the total number of atoms in the sample universe (i.e. of the examples covered by C) minus the number of unknown atoms.

In the case in which no constraint is available, $n^{\oplus}(C') + n^{\oplus}_{A}(C') = n^{\oplus}(C)$ and $k^{\oplus}(C')$ is always 1. In this case, we use following more conservative estimate

$$k^{\oplus}(C') = \frac{n^{\oplus}(C')}{n^{\oplus}(C)}$$

with a lower bound, set by default to 0.1, so that $k^{\oplus}(C')$ can not drop below this threshold. This estimate turned out to be often more realistic also when constraints are available, due to the fact that much more positive information (represented by facts of the programs) is usually available rather than negative information (represented by constraints). Therefore, this more conservative estimate was used in most of the experiments.

Finally, we must consider the case in which some abducibles were already present in $Body(X)$. We will assume that all the examples covered by C' with abduction are covered with abduction as well by C. $k^{\oplus}(C')$ must then express the probability that both the current assumptions and those made before are true at the same time. Therefore:

$$k^{\oplus}(C') = k^{\oplus}(C) \times \frac{n^{\oplus}(C')}{n^{\oplus}(C)}$$

The formula for $k^{\ominus}(C')$ can be derived with a similar reasoning:

$$k^{\ominus}(C') = k^{\ominus}(C) \times \frac{n^{\ominus}(C')}{n^{\ominus}(C)}$$

Implementation

Prolog was chosen for the implementation of ACL1 because it is particulary suitable for the elaboration of data in the form of Prolog programs, since there is no syntactic difference between code and data: terms and atomic formulas have the same structure. Prolog offers also a number of built-in meta-level predicates for accessing the program clauses that allowed the implementation of the abductive proof procedure as a meta-interpreter. Moreover, the availability of lists as primitive data structures is particularly useful for implementing algorithms that search a state space: a list was used to represent the beam of possible clauses in the procedure Specialize.

The ACL1 code is composed of the following main procedures. `i(File)` is the command to be given at the Prolog prompt for starting the induction. It reads the file that contains the input data, starts the covering loop and writes the output to a file.

`covering_loop(Eplus,Eminus,RulesIn,RulesOut,DeltaIn,DeltaOut)` implements the covering loop: it first initializes the beam with a clause with an empty body for every target predicate and starts the specialization loop. Then it adds the clause generated in the specialization loop to the current set of rules and updates the training set and the assumption set.

`specialize(BeamIn,BeamOut,Eplus,Eminus,DeltaIn,N)` implements the specialization loop. The recursion is stopped when the first clause in `BeamIn` is consistent or when the maximum number of specialization steps `N` is reached.

The predicate `evaluate(Value,Clause,Eplus,Epluscovered,Eminus,Eminuscovered, Eplus,Epluscovered,DeltaIn,DeltaOut,...)` is used in order to evaluate clauses. It takes as input the clause to be evaluated `Clause`, the current training set `Eplus, Eminus`, the current set of assumptions `DeltaIn`, and returns the values of the heuristic function `Value` together with the sets of covered examples `Epluscovered, Eminuscovered` and the new set of assumptions `DeltaOut`.

The system has been implemented using SICStus Prolog [Swe97] and is available at the following address `http://www-lia.deis.unibo.it/Software/ACL/`.

4.4.2 Learning Integrity Constraints

The second subproblem ACL2 of learning integrity constraints can be seen as a problem of learning from interpretations (see section 3.3) where we have to discriminate between allowed interpretations (explanations for positive examples) and forbidden interpretations (explanations for negative examples). The ICL system ([DRL95], see also section 3.4.4) solves exactly this problem, and we can therefore use it to solve the ACL2 problem. We recall here the definition of the problem solved by ICL.

Definition 45 (ICL Problem)
Given

- *a definite clause background theory B,*
- *a set of positive interpretations P,*
- *a set of negative interpretations N.*

Find *a clausal theory H such that*

- *for all $p \in P$, $M(B \cup p)$ is a true interpretation of H, i.e. $M(B \cup p) \models H$ (Completeness);*
- *for all $n \in N$, $M(B \cup n)$ is a false interpretation of H, i.e. $M(B \cup n) \not\models H$ (Consistency);*

In our case, we have to learn integrity constraints on abducibles by using the information contained in the sets Δ^+ and Δ^- generated from ACL1. ICL can be used to solve the ACL2 problem with the following inputs:

- the program $P \cup P'$ as the background knowledge B,
- one positive interpretation $p = \Delta^+$;
- one negative interpretation $n_i = \{\overline{l_i}\}$ for each $l_i \in \Delta^-$.

Learned constraints will be true in the model $M(\Delta^+)$ and will be false in each model $M(\{\overline{l_i}\})$. Therefore, when the integrity constraints are added to the final abductive theory, they will not allow any of the abductive assumptions $\overline{l}_i$ with $l_i \in \Delta^-$. This in turn means (see theorem 44) that negative examples cannot be abductively entailed as required for the full ACL problem.

We mention here that another possibility of integrating the two subproblems of ACL1 and ACL2 is to record in ACL1 all possible explanations Δ_{e^-} for each negative example e^- in its three-valued version and to give to ICL each one of these explanations Δ_{e^-} as a negative interpretation. In this way, we do not decide a priori in ACL1 how (i.e., on which assumption) each of the explanations for negative examples must be made inconsistent later by the constraints produced by ACL2. This decision is taken a-posteriori by ACL2 itself when it produces the constraints. Hence ICL has the freedom to make Δ_{e^-} inconsistent on any of the abducibles in it. Learning constraints is now easier because ICL can choose which abducible to make inconsistent in each explanation Δ_{e^-}. However, this alternative way of splitting the ACL problem is only appropriate when assumptions for positive examples cannot contradict those for negative examples. Otherwise, such an inconsistency will not be detected until the end of the second phase requiring the (costly) return to the first phase.

4.4.3 Properties of the Algorithm

In this section, we show the soundness of the ACL algorithm given in the previous section and discuss its (lack of) completeness.

Let us first adapt the properties of soundness and completeness of an inductive algorithm defined in section 3.2.1 for the problem definition of ACL. Given an algorithm, $\mathcal{A}$, for ACL we shall write $\mathcal{A}(\langle \mathcal{P}, \mathcal{I} \rangle, E^+, E^-, T) = T'$ to indicate that, given the hypothesis space $\langle \mathcal{P}, \mathcal{I} \rangle$, the positive and negative examples E^+ and E^-, and the background knowledge T, the algorithm outputs a program T'. We write $\mathcal{A}(\langle \mathcal{P}, \mathcal{I} \rangle, E^+, E^-, T) = \bot$ when $\mathcal{A}$ does not produce any output.

With respect to the ACL problem definition of section 4.3, the definitions of soundness and completeness are given as follows.

Definition 46 (Soundness) *An algorithm $\mathcal{A}$ is sound if whenever $\mathcal{A}(\langle \mathcal{P}, \mathcal{I} \rangle, E^+, E^-, T) = T'$, then the theory $T' = \langle P \cup P', A, I \cup I' \rangle$ that is computed satisfies the conditions of definition 4.3, i.e. $P' \in \mathcal{P}$, $I' \in \mathcal{I}$ and*

- $T' \models_A E^+$,
- $\forall e^- \in E^-$, $T' \not\models_A e^-$.

Definition 47 (Completeness) *An algorithm $\mathcal{A}$ is complete if whenever $\mathcal{A}(\langle \mathcal{P}, \mathcal{I} \rangle, E^+, E^-, T) = \bot$ then there is no computed theory T' that satisfies the conditions of definition 4.3.*

The ACL algorithm is sound but not complete.

Theorem 48 (Soundness) *The algorithm ACL is sound.*

The proof of this theorem is given in appendix A.2. □

The ACL algorithm is incomplete because the search space of ACL1 is not completely explored. In particular, there are two choice points which are not considered in order to reduce the computational complexity of the algorithm. The first choice point is related to the greedy search in the space of possible programs as in most ILP systems. When no new clause can be added by the specialization loop, no backtracking is performed on previous clauses added. This can prevent the system from finding a solution when it is learning a recursive predicate because of the interaction among clauses: an overgeneral clause may make inconsistent a correct clause still to be learned that calls it. This problem is alleviated in the system M-ACL, (see section 4.5 below) where backtracking on clause addition is performed.

The second choice point concerns the different abductive explanations that may be available for each example: the choice of an explanation for an example can affect the coverage of future examples. The algorithm does not perform backtracking on example explanations, it just selects one and commits to it.

Finally, we comment that, with respect to the generality of the two separate parts of the hypothesis space, the solution found by the ACL algorithm combines a most general program with a most specific set of integrity constraints. Finding most specific integrity constraints means that these constraints will restrict as much as possible the number of abductive extensions that are allowed by the learned theory. This is desirable since initially, with no constraint, any set of assumptions is allowed: with the learned constraints we want to maximize the information gained from them by maximizing the collection of assumption sets that they exclude.

4.5 ACL for Multiple Predicate Learning

ACL finds a natural application in the problem of multiple predicate learning multiple in ILP. In multiple predicate learning we have a learning situation which is similar to the problem of learning with incompleteness in the background data, since each predicate to be learned forms part of the background theory for the other predicates and the available definitions for the target predicates are incomplete during learning.

Multiple predicate learning is a task that poses a number of problems to most ILP systems. These problems and difficulties have been exposed in [DRLD93]. In this section we will discuss these problems and show how they can be addressed within the ACL framework by a suitable extension of the ACL1 algorithm and system.

4.5.1 Multiple Predicate Learning: Problems and Difficulties

In order to learn multiple predicates, it may seem at first natural to repeat several times a single predicate learning task. However, this simple approach suffers from several problems [DRLD93]. It is sensitive to the order in which predicates are learned, it can produce overgeneral theories and is unable to learn mutually recursive predicates for which it is necessary to alternatively learn clauses for different predicates. In addition, a top-down covering algorithm that can interleave the learning of clauses for different predicates faces the problem that a clause that is consistent with the negative examples for one predicate can make the theory inconsistent with the negative examples for another predicate.

In order to illustrate this central problem of generating inconsistent hypotheses, we distinguish between two types of consistency of a clause: *local* and *global consistency* of a new clause with respect to the theory learned so far (*current hypothesis*). The definitions we give extend those given in [DRLD93] by relating the consistency of a clause to the current partial hypothesis. Intuitively, a clause is locally consistent if it does not cover any negative example for its head predicate when it is added to a consistent partial hypothesis. Instead, a clause is globally consistent if it covers no negative example for any target predicate. Before giving the definitions, let us first introduce some terminology.

Let the training set be $E = E^+ \cup E^-$. We assume that E contains examples for m target predicates $p_1, \ldots, p_m$ and we partition E^+ and E^- in $E^+_{p_i}$ and $E^-_{p_i}$ for $i = 1, \ldots, m$, according to these predicates. A *hypothesis* H is a set of clauses for some of the target predicates. The function $covers(B, H, E)$ gives the set of examples in E that are covered by the hypothesis H with background knowledge B, i.e., $covers(B, H, E) = \{e \in E \mid B \cup H \models e\}$.

Definition 49 (Local consistency) *Let H be a consistent hypothesis and C a clause for the predicate p_i. Then C is* **locally consistent with respect to** *H if and only if $covers(B, H \cup \{C\}, E^-_{p_i}) = \emptyset$.*

Definition 50 (Global consistency) *Let H be a consistent hypothesis and C a clause for any target predicate. Then C is* **globally consistent with respect to** *H if and only if*

$covers(B, H \cup \{C\}, E^-) = \emptyset$.

By repeating several times a single predicate learning task, we repeatedly add locally consistent clauses to the current partial hypothesis. However, when learning multiple predicates, adding a locally consistent clause to a consistent hypothesis can produce a globally inconsistent hypothesis as it is shown in the next example adapted from [DRLD93].

Example 51 *Suppose we want to learn the definitions of ancestor and father from the knowledge base:*

$B = \{parent(a, b), parent(d, b), parent(b, c), male(a), female(b)\}$

and the training sets:

$E^+ = \{ancestor(a, b), ancestor(d, c), father(a, b)\}$

$E^- = \{ancestor(b, a), ancestor(a, d), father(b, c), father(a, c)\}$

Suppose that the system has first generated the rules:

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

$father(X, Y) \leftarrow ancestor(X, Y), male(X)$

The second rule is incorrect but the system has no means of discovering it at this stage, since it is locally and globally consistent with respect to the partial definition for ancestor.

Then the system learns the recursive rule for ancestor:

$ancestor(X, Y) \leftarrow parent(X, Z), ancestor(Z, Y)$

This clause is locally consistent with respect to the current hypothesis because none of the negative examples for ancestor will be covered, but it is globally inconsistent because the negative example $father(a, c)$ will be covered.

In order to address this problem, most top-down ILP learning systems use extensional coverage (see section 6). In this way, clauses are learned independently from each other and hence these systems simply avoid considering the problems of global inconsistency during their learning process. However, extensional coverage leads to other problems since the learned theory can be both inconsistent and incomplete, as it is shown in [DRLD93].

Instead of adopting extensional coverage, the system MPL [DRLD93] uses intensional coverage and solves the problem of maintaining the global consistency of the current hypothesis by re-testing the negative examples for all predicates and by performing backtracking on clause addition to the theory.

Another problem that can arise in multiple predicate learning concerns the case when scarce training examples, particularly negative examples, are available for a subsidiary predicate. In this case, a system could learn an overgeneral definition for the subsidiary predicate and this may prevent the system from finding a consistent definition for other predicates. The next example illustrates this.

Example 52 *Suppose we want to learn the predicates grandfather and father. Let the background theory be:*

$P = \{parent(john, mary), male(john),$
$parent(david, steve), male(david), male(steve),$
$parent(steve, jim), male(jim),$
$parent(mary, ellen), female(mary), female(ellen)$
$parent(ellen, sue), female(sue)\}$

and let the training data for both concepts be:

$E^+ = \{grandfather(john, ellen), grandfather(david, jim), father(john, mary)\}$

$E^- = \{grandfather(mary, sue), grandfather(mary, john),$
$father(john, ellen), father(david, jim), father(jim, david)\}$

A system that learns first the rule for father, may learn the overgeneral rule

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

since it is consistent with the negative examples for father. Then, it would not be able to accept the correct rule for grandfather, since

$grandfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)$

would cover as well the negative example $grandfather(mary, sue)$.

On the other hand, if the system first learns the above correct rule for grandfather, it again needs to recognize that this implies additional negative examples for father in order to avoid the same overgeneral rule for father.

4.5.2 M-ACL: a Multiple Predicate Learning framework

The following quote from [DRLD93] succinctly summarizes the major challenges of multiple predicate learning. Learning multiple predicates requires an approach that "...discovers a good order of learning the predicates; interleaves the learning of different predicates; recovers from overgeneralization; and takes into account global effects". We will show here how the ACL framework, in particular ACL1, can be suitably employed to provide these characteristics.

The basic idea of multiple predicate learning through ACL is to set the target predicates to be learned as abducible predicates and use the abductive information that ACL1 generates on these to link the learning of the different predicates. This information can be used in two inter-related ways. Firstly, it acts as extra training examples for the target predicates. After the generation of each clause by ACL1, the associated assumptions Δ^+ and Δ^- about other target predicates are added to the training set according to their sign. In effect, training information for one predicate is transformed into training information for other predicates. At the same time, this abductive information generated by ACL1 is used to give us an extra mechanism for ensuring global consistency in the hypothesis in a way similar to abductive truth maintenance systems [KM90c, GM90]. The multiple predicate learning algorithm and system is obtained from ACL1 by encompassing this in a process that uses the abductive information, produced by ACL1, to detect and restore consistency.

The M-ACL algorithm is therefore based on a dynamic set of training examples E for the target predicates that contains the given training examples together with those generated through abduction. It rests on the important observation that, for definite logic programs, we can detect the local or global consistency of a clause by testing the training examples for its head predicate as follows:

- a clause is **locally consistent** if it does not cover any negative example from the **original training set**, while
- a clause is **globally consistent** if it does not cover any negative example from the abductively **extended training set**.

To illustrate this, consider two predicates p and q, where q depends on p. Suppose that, when testing a rule for q, a negative example $p(\vec{t_p})$ for p is generated for uncovering a negative example $q(\vec{t_q})$ for q. Afterwards, if we learn a clause for p that does not cover $p(\vec{t_p})$, then also $q(\vec{t_q})$ will not be covered and the clause for p will be globally consistent.

procedure M-ACL(
  inputs : E^+, E^- : training sets,
    P : background theory,
  outputs : H : learned theory, E_A : abduced examples)

$H := \emptyset$
$\Delta := \emptyset$
$E_c := E^+ \cup not\_E^-$
repeat
  $\text{Specialize}_M(P, H, E_c, \Delta; r, E_r^+, E_r^-, \Delta_r)$
  $E_c := E_c \setminus E_r^+$
  $H := H \cup \{r\}$
  $\text{Test}(H,\ \Delta_r^-; \Delta_f^-)$
  while Δ_f^- is non-empty:
    $\text{Choose}(\Delta_f^-;\ \text{Ab})$
    $\text{Refine}(H,\text{Ab},\Delta, E_c;\ H, E_c, \Delta)$
    $\Delta_f^- := \Delta_f^- \setminus \{\text{Ab}\})$
  until $E_c^+ = \emptyset$ (*covering loop*)
  If $E_r^- \neq \emptyset$ then
    $\text{RetractClauses}(H, \Delta, E_r^-, E_c; H, \Delta, E_c)$
  $\text{Update}(E_c, \Delta_r; E_c)$
  $\Delta := \Delta \cup \Delta_r$
endwhile
$E_A := E_c \setminus (E^+ \cup E^-)$
output H, E_A

Figure 4.4: The M-ACL algorithm

The M-ACL algorithm extends ACL1 and is shown in figure 4.4. H, Δ and E_c represent, respectively, the current hypothesis, the current set of assumptions and the extended set of training examples. At first, an extension of the ACL1 Specialize procedure (see figure 4.2) is called, denoted by Specialize_M. This uses *extensional coverage* and tries to generate a new clause r that is correct with respect to the current **extended** set of training examples E_c. If this is possible, then the generated clause, r, will cover a set of positive examples E_r^+ and no negative example ($E_r^- = \emptyset$) with the assumptions Δ_r. If no rule consistent with the current set of negative examples can be found, then Specialize_M looks for a clause that is consistent only with the original set of examples but covers the subset E_r^- of negative examples generated by abduction. If no such clause can be found, then Specialize_M fails and M-ACL also fails.

We then check if the generated clause, that was found extensionally consistent, is also intensionally consistent. To this purpose, the set of negative assumptions $\Delta_r^- \subseteq \Delta_r$ generated by Specialize_M using extensional coverage is tested against the current intensional hypothesis: the assumptions are considered as negative examples that must not be covered. If some of these assumptions are violated (Δ_f^- denotes this set of violated assumptions), we try to remove these violations by iteratively choosing some assumption(s) Ab from Δ_f^- and refining the current hypothesis. The refinement consists in specializing (or retracting and re-learning) the existing rules that currently define the target predicate of the assumption(s) Ab and are causing the violation with Ab.

If E_r^- is not empty, then the clause is locally but not globally consistent and we backtrack on the clauses that generated the covered examples. These rules are deleted from the current hypothesis, positive examples covered by them are re-added to the training set and assumptions and examples generated by them are removed from Δ and from the training set. In order to support the backtracking required at this step, the abductive procedure employed by ACL1 is extended to record, for every assumption, the clause responsible for generating it.

These two tests on Δ_r^- and E_r^-, when they are successful (i.e. when both Δ_f^- and E_r^- are empty), ensure that the next candidate hypothesis, i.e. $H \cup \{r\}$, is globally intensionally consistent. If one of the tests is not successful, this means that there is a possible conflict between the clause just learned and the hypothesis H. There are two types of conflicts. The first type ($\Delta_f^- \neq \emptyset$) appears because the assumptions for the rule r are inconsistent with the current hypothesis, while the second type ($E_r^- \neq \emptyset$) appears because the rule r covers some of the negative examples generated by other rules. In the first case we refine the clauses that cover the negative assumption Ab, so in effect we keep the assumption Ab generated by r and we reject the clauses that derive its opposite. In the second case, instead, we reject the (abductively assumed) negative examples covered by r by retracting the clause that has generated them earlier. In both cases, the clause r is kept and the theory is modified in order to be in accordance with it.

The specialization or re-learning of the Refine procedure is carried out using, again, the M-ACL procedure where the agenda of Specialize$_M$ initially contains the rule to be refined instead of rules with an empty body. Therefore, new assumptions may be generated and new retraction/refinement of previous rules may be necessary.

When the resolution of the violations is completed (sometimes this may not be possible), assumptions about target predicates are moved from Δ_r to the current training set before returning back to the first step to learn a new rule for covering the remaining positive examples of the original training set.

We point out that the generation in M-ACL of the candidate clauses by Specialize$_M$ (using extensional coverage) allows the interleaving of learning clauses for different target predicates. M-ACL does not require a given order in which to learn the target predicates: the specialization loop in Specialize$_M$ is initialized with an empty body clause for each target predicate and the same heuristic function is used in order to select the next clause to refine and therefore the next predicate to learn.

Let us now examine how this algorithm and the M-ACL system that is based on it, behaves in the cases of examples 51 and 52 (see also appendix A.4 for the output behaviour of M-ACL on these examples). In example 51, suppose the system has generated in the current hypothesis the clauses

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

$father(X,Y) \leftarrow ancestor(X,Y), male(X)$

When testing the above clause for *father*, the test of the negative example $father(a,c)$ will produce the assumption $\{not\_ancestor(a,c)\}$. These assumptions then become additional negative examples for *ancestor*. Their test does not produce a violation and so at this point the system tries to find a clause covering the remaining positive examples for *ancestor*. The correct solution

$ancestor(X,Y) \leftarrow parent(X,Z), ancestor(Z,Y)$

is not globally consistent, since it covers the new negative example $ancestor(a,c)$ generated from the rule for *father*. However, this clause is locally consistent, since it does not cover any of the negative examples in the original training set of *ancestor*. It is therefore added to the current hypothesis and the system backtracks to the clause that has generated this violating assumption, namely to the clause for *father*. This clause, together with the assumptions that it has generated, are retracted and the examples for *father* are re-added to the training set for this concept to be learned again. At this point, the system is able to learn the correct rule for *father*

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

This example shows one way in which the M-ACL system uses the dynamically generated abductive information on the target predicates to have a focussed mechanism of detection and repair of global inconsistencies. The system can directly detect previously generated wrong clauses and re-learn other rules for these predicates. Thus, it can recover from an incorrect rule that was generated from an inappropriate order in the learning of the different predicates. In this sense the system is less sensitive to the order of learning.

In example 52, M-ACL learns first the rule for *grandfather* because more information is available about it and the heuristic function prefers it to any of the rules for *father*. When M-ACL generates the rule

$grandfather(X,Y) \leftarrow parent(Z,Y), father(X,Z)$

it uses the examples for *father* as background knowledge making also assumptions about it when this is needed.

Given the training examples for *grandfather*

$E^+_{gf} = \{grandfather(john, ellen), grandfather(david, jim)\}$

$E^-_{gf} = \{grandfather(mary, sue), grandfather(mary, john)\}$

The above rule will be learned by M-ACL by making the assumptions $\{not\_father(mary, ellen), father(david, steve)\}$ that become additional training examples for *father*. From this new training set, the system is then able to generate the correct rule for *father*. Note that, without the new negative example $father(mary, ellen)$, it would have been impossible to generate the correct rule for *father* and the overgeneral rule $father(X,Y) \leftarrow parent(X,Y)$ would have been learned. Thus M-ACL avoids (in this case) the problem of overgeneralization.

M-ACL does not overgeneralize even if the system first generates the overgeneral rule for *father*. In this case, extensional coverage still allows Specialize$_M$ to generate the correct rule for *grandfather* and to generate the same negative assumption on *father* as above. At this stage the M-ACL system will recognize that we have a violation on the assumption Ab=$\{not\_father(mary, ellen)\}$ and the Refine procedure will lead the system to specialize (or re-learn) the rule for *father* thus producing at the end the same correct and complete hypothesis as above. Hence, independently of the order of learning, the same extra assumptions are generated and effectively used to produce the same final result.

This is a general pattern of the overall mechanism for global inconsistency detection and restoration within M-ACL. If a violating assumption is generated first then this is taken into account as an extra training example for Specialize$_M$ when learning rules for this predicate and hence the potential violation can be avoided at the stage of generation of candidate clauses. On the other hand, if the violating assumption is generated after an overgeneral rule has already been generated, this can be directly detected and the repair of this rule is effected. In this way we alleviate the problem of overgeneralization and the system does not depend crucially on the order in which the predicates are learned.

Summarizing, we point out that in effect the M-ACL system uses a hybrid of extensional and intensional coverage: extensional coverage in the generation of candidate clauses using examples of other target predicates as background facts together with an intensional test of the theory on the generated negative assumptions. This combination, together with its overall mechanism for detecting and repairing inconsistency, allows the system to interleave the learning of the different target predicates with little dependence on the order of learning and to overcome the problem of overgeneralization. An important characteristic of M-ACL is the fact that its test for global consistency is performed only on a "narrow" subset of the negative examples, resulting in a focussed handling of global inconsistency with a direct detection and repair of the inconsistency. The M-ACL system tests only the negative abduced examples for the head predicate of the clause under test. In contrast, the system MPL [DRLD93] checks the negative examples for all target predicate after the addition of a clause. In this way M-ACL performs a smaller number of tests with respect to MPL. However, the speedup obtained can be partially counterbalanced by the time and memory needed to keep track of the abductive assumptions.

4.6 Experiments

Two series of experiments have been performed in order to show the ability of ACL to (1) learn from incomplete background knowledge and (2) to perform multiple predicate learning.

4.6.1 Learning from Incomplete Background Knowledge

The main purpose of these experiments was to test how well ACL could learn under incomplete information and to investigate its behaviour under different forms and degrees of incompleteness. The following three problems have been considered: (1) the multiplexer example from [dV89], (2) learning family relations from a database with varying degree of incompleteness and (3) new problems of learning from (real-life) data of market research questionnaires which is incomplete due to unanswered questions or "don't care" answers.

The same incomplete data was also given to other ILP systems such as FOIL. As expected these produced theories with a larger number of overspecific rules. Hence for these experiments, the results of ACL have also been compared with those of the system mFOIL [DB92] that has special techniques for handling more generally imperfect data, both noisy and incomplete (see section 3.4.3). Let us briefly recall here mFOIL's approach for dealing with incomplete data that consists in relaxing the completeness requirement for the sufficiency stopping criterion: mFOIL stops adding a clause to the theory when too few positive examples remain for a clause to be *significant* or when no significant clause can be found with expected accuracy greater than the default. The significance test is based on the *likelihood ratio statistic* [Kal79]: a clause is deemed significant if its likelihood ratio is higher than a certain significance threshold. The default value for the significance threshold is 6.64 corresponding to a significance level of 99%. Unless otherwise specified, mFOIL was run in all the experiments with the parameters set in the following way: the heuristic function is the m-estimate with m=2, the beam size is 5, no negation as failure literals are allowed in the language bias, the minimum number of examples that each rule must cover is 1 and the significance threshold is 6.64.

The major part of these experiments has concentrated on investigating the behaviour of the ACL1 subsystem. In all experiments though full abductive theories have been learned that include integrity constraints which support the abductive rules generated by ACL1.

Multiplexer

The multiplexer example is a well-known benchmark for inductive systems [dV89]. It has recently been used in [DRD96c] for showing the performance of the system ICL-Sat on incomplete data. We performed experiments on the same data of [DRD96c] and compared the results of ACL1 with those of ICL-Sat and mFOIL [DB92].

The problem consists in learning the definition of a 6 bits multiplexer, starting from a training set where each example consists of 6 bits, where the first two bits are interpreted as the address of one of the other four bits. If the bit at the specified address is 1 (regardless of the values of the other three bits), the example is considered positive, otherwise is considered negative. For example, in the tuple 10 0110, the first two bits specify that the third bit should be at 1, so this example is positive.

For the 6-bit multiplexer problem we have $2^6 = 64$ examples, 32 positive and 32 negative. We perform three experiments as in [DRD96c]: the first on the complete dataset, the second on an incomplete dataset and the third on the incomplete dataset plus some integrity constraints. The incomplete dataset was obtained by considering 12 examples out of 64 and by specifying for them only three bits where both the examples and bits were selected at random. E.g. the above example 10 0110 could have been replaced by 1? 0?1?. The dataset of the third experiment is obtained by including additional integrity constraints to this incomplete dataset.

In order for ACL to learn from the incomplete datasets, we used a representation formalism where the incompleteness is contained in the background knowledge. Examples are represented as atoms of the form $mul(c)$ where c is a constant that represent a specific tuple and background predicates are used to express the bit values for that specific tuple. For example, the tuple 10 0110 is represented by the atom $mul(e1)$ in the training set and by the following facts in the background knowledge

$bit1at1(e1)$ $bit2at0(e1)$ $bit3at0(e1)$
$bit4at1(e1)$ $bit5at1(e1)$ $bit6at0(e1)$

All the predicates of the form $bitNatB$ are declared as abducibles and integrity constraints of the form below are added to the background theory

$$\leftarrow bitNat0(X), bitNat1(X)$$

The incomplete tuple 1? 0?1? can now be represented as

$bit1at1(e1)$ $bit3at0(e1)$ $bit5at1(e1)$

and assumptions can be made about bits 2, 4 and 6.

In the third experiment, the constraints are such that: 1) the value of unknown attributes is still unknown (could still be 1 or 0); 2) some combination of values incompatible with

Experiments	ACL1	ICL-Sat	mFOIL
1) Complete background	100 %	100 %	100 %
2) Incomplete background	98.4 %	82.8 %	96.875 %
3) Incomplete background plus constraints	96.875 %	92.2 %	96.875

Table 4.1: Performance on the multiplexer data

the known class is now impossible. E.g., for the example ?0 ?1?1 (negative), the following constraints were added:

$$\leftarrow bit1at1(e1), bit5at1(e1)$$
$$bit1at1(e1) \leftarrow bit3at1(e1)$$

The first constraint states that bits 1 and 5 can not be both 1, otherwise the example would be positive, while the second constraint states that if the third bit is 1, then also the first bit must be at 1.

ACL1 and mFOIL were run on all the three dataset. The measure of performance that was adopted is classification accuracy, defined as the number of positive and negative examples correctly classified over the total number of examples in the testing set, i.e. the number of positive examples covered plus the number of negative examples not covered over 64. In order to test the learned theory, the complete background knowledge was used in all three experiments.

The results are reported in table 4.1. In experiment 2), ACL1's accuracy was significantly better than ICL-Sat's and slightly better than mFOIL's, while in experiment 3) ACL1's accuracy was only slightly superior to ICL-Sat's and the same as mFOIL's. The accuracy for mFOIL was the same in experiment 2) and 3) since mFOIL is not able to exploit the integrity constraints. The accuracy of ACL1 in the third experiment is lower than in the second: this unexpected result is due to the fact that negative examples are tested as $T' \models_A not\_e^-$ during learning and as $T' \not\models_A e^-$ when evaluating the performance. The high accuracies obtained show that the ACL1 system has in this case solved successfully the full ACL problem.

We also tested the theory with incomplete testing data thus showing the ability of the generated theories to classify incomplete examples. We tested the theory on the same incomplete data set used for learning in experiments 2 and 3. The results of this different testing are reported in table 4.2, where $n^{\oplus}$ is the number of covered positive examples, while $n^{\ominus}_{ACL}$ is defined as the number of negative examples that are (incorrectly) covered according to the full ACL problem (i.e. for which $T' \models_A e^-$). In this case, the accuracy has increased from experiment 2) to 3) as expected. Similar results are obtained with other randomly generated incomplete subsets of the complete training examples.

Learning Family Relations

In this experiment we considered the problem of learning family predicates, e.g. that of father, from a database of family relations [BDR96] containing facts about several predicates parent, son, daughter, grandfather, male and female etc. We performed several experiments

Experiments	$n^{\oplus}$	$n^{\ominus}_{ACL}$	Accuracy
Experiment 2	32	5	92.2 %
Experiment 3	32	2	96.9 %

Table 4.2: Testing with incomplete data

with different degree of incompleteness of the background knowledge and we compared the results of ACL1 with those of mFOIL.

The complete background knowledge contains, amongst its 740 facts, 72 facts about parent, 31 facts about male and 24 facts about female. The training set contains 36 positive examples of father taken from the family database and 200 negative examples of father that were generated randomly. Experiments were performed from datasets containing 100%, 90%, 80%, 70%, 60%, 50% and 40% of the facts. The incomplete datasets were generated by randomly taking out facts from the background knowledge, while the training set was the same for all experiments.

The experiments with ACL1 were performed first by considering a background knowledge with no constraint and then by adding the following integrity constraints:

$\leftarrow male(X), female(X)$
$\leftarrow son(X,Y), female(X)$
$\leftarrow daughter(X,Y), male(X)$
$\leftarrow son(X,Y), not\ parent(Y,X)$
$\leftarrow daughter(X,Y), not\ parent(Y,X)$

The results are shown in table 4.3. As regards mFOIL, the experiments were done with a significance threshold of 6.64 and of 10. In both cases, the results of completeness and consistency are the same for all the incompleteness levels and the number of rules learned is similar (in some cases slightly lower for significance 10). The table reports the number of clauses for significance 6.64.

ACL1 without constraints was able to learn theories that are simpler (i.e. they contain less rules with shorter bodies) than those learned by mFOIL and that are consistent in all but two case, while the theories learned by mFOIL are always inconsistent for incomplete data. However, mFOIL always learns complete theories, while ACL1 without constraints learns incomplete theories at 40% and 80%. If integrity constraints are used with ACL1, the system is able to learn the simplest complete and consistent theory (i.e. $father(X,Y) \leftarrow parent(X,Y), male(X)$) at all levels of incompleteness.

ACL1 was able to learn more compact theories because it can exploit abduction for covering positive examples for which no information is available, while mFOIL is obliged to learn new rules for covering the examples that are not covered by the correct theory due to the lack of information. When using abduction to complete the missing data it is sometimes possible to make wrong assumptions especially when the incompleteness is spread over many background predicates. This could then result in learning a wrong rule, as it happened with incompleteness 40% and 80% where, respectively, the following two rules were learned:

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

When integrity constraints are used in the background theory, the possibility of making wrong assumptions is reduced and for all incompleteness levels a complete and consistent

Data	N. of clauses			Complete			Consistent		
	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
100%	1	1	1	y	y	y	y	y	y
90%	1	1	4	y	y	y	y	y	n
80%	1	1	6	n	y	y	n	y	n
70%	1	1	7	y	y	y	y	y	n
60%	1	1	8	y	y	y	y	y	n
50%	1	1	8	y	y	y	y	y	n
40%	1	1	10	n	y	y	n	y	n

Table 4.3: Performance on the family data: (1)=ACL1, (2)=ACL1+IC, (3)=mFOIL

theory is learned.

Other experiments on the same database were also performed, where the incompleteness is isolated only in some of the predicates. For example, an experiment was performed where only male and female are incomplete, in order to show that ACL1 still learns the simplest theory by making abductions on male, instead of generating overcomplex theories where the information about the sex is taken from other predicates, for example by using the literal $grandfather(X, Z)$ to ensure that X is a male. On the same data, mFOIL learns instead overcomplex theories.

Moreover, for these experiments, we have applied the ICL system to solve also the ACL2 problem of learning integrity constraints from the associated assumptions generated in the first phase by ACL1 and thus solving the full ACL problem. In some cases, this generated the constraints that we usually expect from this domain such as

$\leftarrow male(X), female(X)$, or

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

In other cases, instead, the generated constraints are more specific, such as

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

$parent(X, Y); mother(X, Y) \leftarrow son(Y, X)$

This is due to the fact that the purpose of the generated constraints in ACL2 is just to support the assumptions of the ACL1 part, without considering other data from the background theory, and therefore ICL selects any set of constraints that is sufficient to achieve this specific task.

Marketing Research Data

ACL has been used on several sets of real world data from market research questionnaires aiming to understand the possible success or failure of selling a new product. In this subsection we report on one such experiment.

This case concerns a market research on a new soft drink brand. The research was conducted by asking 100 people to taste the drink and to fill a questionnaire regarding the characteristics of the drink and their personal tastes. The concept we want to learn is $buy(X)$ that expresses whether the person would buy the drink or not. Out of the 100 people interviewed, 52 answered that they would buy the product, 32 would not and 16 don't know. Therefore we have 52 positive examples and 32 negative ones.

There are 24 background predicates representing the answers to the questionnaire. Some questions require an answer chosen among a number of values: for example, the question about the aroma of the drink can be answered with low, right or high. These values have been represented using the predicates $lowaroma(X)$, $higharoma(X)$ and $rightaroma(X)$. Instead, questions requiring a yes-no answer have been represented using a single predicate: for example, whether the person likes natural things is encoded with the predicate $likenatural(X)$.

Some questions are unanswered or have don't care answers and these have been treated as incomplete information in the background. Out of 24 background predicates, 9 are incomplete with degree of incompleteness from 37% (i.e. 37 people out of 100 have not answered or have answered don't care) up to 89%. The incomplete background predicates have been considered as abducibles and integrity constraints have been introduced in order to avoid the abduction of two different answers for the same question. For example, for the question of "overall flavour" we have the following constraint on the abducible predicates that record answers to this question:

$\leftarrow goodflavouroverall(X), poorflavouroverall(X)$

ACL1, mFOIL and FOIL were run on this data. All three systems found theories with a dominant clause covering the majority of the examples plus other very specific rules. Both ACL1 and mFOIL found the following dominant clause:

$buy(X) \leftarrow goodflavouroverall(X), rightsweetness(X)$

According to ACL1, this clause covers 47 positive examples, 10 of which by abduction, and it does not cover any negative example, while, according to mFOIL, it covers 37 positive examples and no negative one. This clause was judged to be very meaningful by experts in the field with the right sweetness as one of the most important factors in the success of a soft drink. FOIL, instead, has found the following general clause:

$buy(X) \leftarrow goodflavouroverall(X), rightmouthfeel(X)$

that covers 37 positive examples and no negative one.

The second phase of ACL was also run on this data and the following constraint has been found:

$\leftarrow goodflavouroverall(X), higharoma(X)$

which, again, was judged to be significant by experts. This constraint can be used in order to complement the available knowledge on $goodflavouroverall$.

4.6.2 Multiple Predicate Learning

In this section we present some experiments that have been performed with the M-ACL system: learning a definite clause grammar for simple sentences, learning the definitions of the mutually recursive predicates *even* and *odd* and learning multiple family relations. Moreover, as reported earlier in section 5, the M-ACL system was tested on multiple family relations (see examples 5.3, 5.4 and appendix A.4 for details of the M-ACL system's behaviour on these examples) in order to verify its ability to backtrack from a wrong clause and to use additional examples generated from abduction to avoid overgeneralization.

Grammar

The data for this experiment is taken from [DRD96c]. The aim is to learn the following definite clause grammar for parsing very simple English sentences:

(1) $sent(A,B) \leftarrow np(A,C), vp(C,B)$
(2) $np(A,B) \leftarrow det(A,C), noun(C,B)$
(3) $vp(A,B) \leftarrow verb(A,B)$
(4) $vp(A,B) \leftarrow verb(A,C), np(C,B)$

In [DRD96c] Claudien-Sat is used to solve this task starting from different input interpretations.

The first interpretation corresponds to a complete syntactic analysis of the sentence "the dog eats the cat". Therefore the data set contains all the positive and negative facts mentioning the following lists: [the,dog,eats,the,cat], [dog,eats,the,cat], [eats,the,cat], [the,cat], [cat] and []. Another interpretation contains some ungrammatical sentences and corresponds to several attempts to analyze "the cat the cat". It includes all positive and negative facts mentioning the following lists: [the,cat,the,cat], [cat,the,cat], [cat,cat], [the,cat], [cat], [cat,the] and []. Similarly, another interpretation contains all positive and negative facts mentioning the lists [the,cat,eats], [cat,eats], [cat,sings], [the,cat,sings], [dog,cat], [sings], [eats],[the] and [].

M-ACL has learned the above rules in the following order: (2), (3), (1), (4). Note that the definition for *sent* was learned at a point where the definition for *vp* was not complete. This was possible because the system used the examples for *vp* to complete its definition, by exploiting the hybrid form of coverage. In this case the training set was such some negative assumptions about *np* were necessary in order to avoid the coverage of negative examples.

Mutually Recursive Predicates

The task consists in learning the following mutually recursive definition for the predicates $even(X)$ and $odd(X)$

(1) $even(X) \leftarrow zero(X)$
(2) $odd(X) \leftarrow succ(X,Y), even(Y)$
(3) $even(X) \leftarrow succ(X,Y), odd(Y)$

The background knowledge contains the fact $zero(0)$ and the definition of the predicate $succ(X,Y)$ whose meaning is "X is the successor of Y". The training set is obtained from a complete training set containing facts for all the numbers from 0 to 9 by removing some of these. For example, we may remove the positive examples $odd(1), odd(5), odd(7), even(2)$, $even(6)$ and the negative examples $even(3), even(7), even(9), odd(6), odd(8)$. The training set is therefore given by:
$E^+ = \{odd(3), odd(9), even(0), even(4), even(8)\}$
$E^- = \{even(1), even(5), odd(0), odd(2), odd(4)\}$

M-ACL generated the following output:

```
/* Execution time 0.440000 seconds. Generated rules */

rule(even(A),[zero(A)],c2)
GC: yes, LC: yes
Covered positive examples: [even(0)]
Covered positive abduced examples: []
Covered negative abduced examples: []
```

```
Abduced literals: []


rule(even(A),[succ(A,B),odd(B)],c13)
GC: yes, LC: yes
Covered positive examples: [even(8),even(4)]
Covered positive abduced examples: []
Covered negative abduced examples: []
Abduced literals: [[odd(7),c13]]


rule(odd(A),[succ(A,B),even(B)],c21)
GC: yes, LC: yes
Covered positive examples: [odd(9),odd(3)]
Covered positive abduced examples: [odd(7)]
Covered negative abduced examples: []
Abduced literals: [[not(even(3)),c21]]
```

M-ACL has learned clause (3) by exploiting the examples for *odd* as background knowledge and by abducing the missing example *odd*(7). This example is then added to the training set and is covered by clause (2). The negative abduced assumptions on *even* also help in preventing the system to subsequently learn an incorrect clause for this predicate. This experiment shows the ability to learn mutually recursive predicates, exploiting both extensional coverage and abduction.

Multiple Family Relations

Several experiments to learn multiple family relations were carried out. In one such experiment the problem is to learn the predicates brother and sibling from a background knowledge containing facts about parent, male and female. The bias allowed the body of the rules for brother to be any subset of

$$\{parent(X,Y), parent(Y,X), sibling(X,Y), sibling(Y,X),\\ male(X), male(Y), female(X), female(Y)\}$$

while the body of the rules for sibling can be any subset of

$$\{parent(X,Y), parent(Y,X), parent(X,Z),\\ parent(Z,X), parent(Z,Y), parent(Y,Z),\\ male(X), male(Y), male(Z), female(X), female(Y), female(Z)\}$$

Therefore, the rules we are looking for are

$$brother(X,Y) \leftarrow sibling(X,Y), male(X)\\ sibling(X,Y) \leftarrow parent(Z,X), parent(Z,Y)$$

The family database considered for these experiments, taken from [DRLD93], contains 16 facts about brother, 38 about sibling, 22 about parent, 9 about male and 10 about female. The background knowledge was obtained from this database by considering all the facts about male and female and only 50 % of the facts about parent (selected randomly). The

training set contains all the facts about brother and 50 % of the facts about sibling (also selected randomly). Negative examples were generated by making the Closed World Assumption and taking a random sample of the false atoms: 36 negative examples for sibling and 37 for brother. For this problem the abducible predicates are the target predicates *brother* and *sibling* plus the background predicate *parent*.
From this data, M-ACL has constructed first the rule

$brother(X,Y) \leftarrow sibling(Y,X), male(X)$

It has exploited both the positive examples of sibling to cover positive examples of brother and negative examples of sibling to avoid covering negative examples for brother. This rule was constructed first because the heuristics preferred it to the rules for sibling, as more information was available for the predicate sibling rather than for parent. When learning this rule, ACL1 has made a number of assumptions on sibling: it has abduced 3 positive facts (that become positive examples for sibling) and 33 negative facts (that become negative examples for sibling). Then, M-ACL constructs the rule for sibling

$sibling(X,Y) \leftarrow parent(Z,X), parent(Z,Y)$

using this new training set and making assumptions on parent.

This experiment shows again how M-ACL is able to learn multiple predicates exploiting the information available and generating new data for one predicate while learning another.

4.7 Related Work

This chapter basically presents the work discussed in [KR98]. The work builds on earlier proposals in [DK96] and [ELM+96, LMMR97, LMMR98] for learning simpler forms of abductive theories. The use of abduction in learning, either in an implicit or explicit form, has recently been examined by several works [Abe98, AD94, AD95, Coh92, DRB92a, Moo98, IS95, KK98, Sak98]. The abductive assumptions generated during learning are then used in different ways depending on the kind of learning task the system is performing.

In this thesis, abduction is used explicitly as the basic covering relation for defining the concept learning problem. In many other cases, abduction is used as a useful mechanism that can support some of the activities of the learning system. For example, in theory revision, abduction is used as one of the basic revision operators for the overall learning process [AD94, DRB92a, Moo98, Sak98]. For each individual positive example that is not entailed by the theory, abduction is applied to determine the set of assumptions that would allow it to be proved. These assumptions are then used to suggest where the current theory should be revised. In [DRB92a, AD94] the assumptions are either added as facts to the theory or new clauses are learned for covering them. In addition, some of these systems use abductive assumptions for revising overspecific rules by removing from them the literal(s) that generated the assumption [Moo98]. This type of integration of abduction and induction has been studied in a principled way in [AD95] where an integrated framework that combines Abductive and Inductive Logic Programming is proposed.

Abduction is also used as a suitable mechanism for extending Explanation Based Learning [Coh92, O'R94] in cases where the given domain theory is incomplete in the description of some of its predicates. These predicates are then treated as abducible and proofs can be completed by abduction before they are generalized.

The work of [TM94] proposes an approach where abduction is used as the basic covering relation for learning in a different way with respect to the ACL approach. Abduction is carried out on the concept to be learned rather than on the (incomplete) background predicates. A system, called LAB, is presented that uses a simple, propositional form of abduction in the context of a particular application of learning theories for a diagnostic reasoning model. In this reasoning model, theories are composed of rules of the form $symptom \Leftarrow disorder$ and the task of abduction is to find a (minimum) set of disorders that explains all the symptoms. LAB is given as input a set of training cases each consisting of a set of symptoms together with their correct diagnosis (set of disorders) and it produces a theory such that the correct diagnosis for each training example is a (minimum) abductive explanation. In LAB, therefore, the explanations themselves are considered as the output of the target theory (the target predicates are the abducible *disorder* predicates) requiring that the learned theory respects the input-output couples given in the training cases.

Recently, the deeper relationship between abduction and induction has been the topic of study of two workshops [FK96, FK97] where various (preliminary) works on the integration of abduction in learning have been proposed [Abe98, Sak98, KK98]. Of these, [KK98] is the closest to this work: the authors present a top-down learning algorithm that employs an abductive proof procedure for testing the coverage of examples. They consider a cost for each explanation by assigning a cost to every abducible literal. The minimum cost for explaining examples is then taken into account in a FOIL-like clause evaluation function. As ACL, the system can be applied to learn from incomplete background data.

Several other proposals for learning with incomplete information exist. In attributed-based or propositional learning one common way to handle incomplete information (i.e. missing attribute values) is to replace each example with a missing value with several examples, one for each of the possible values of the attribute, and to associate with each example a fractional weight, representing the conditional (with respect to the class of the example) probability of that particular value. The conditional probability of the different values is estimated with the relative frequency from the set of instances. This is the approach followed by ASSISTANT [CKB87], CN2 [CB89] and C4.5 [Qui93]. Various approaches to the handling of incomplete information are empirically compared in [Qui91].

An early ILP system that is able to deal with missing information is that of LINUS [LDG91a]. This learns first order theories by first translating an ILP problem into an attribute-value representation and by then employing an attribute-value algorithm that handles incomplete information. In this way, it is able to deal both with missing arguments and missing facts in the background knowledge. The drawbacks of this approach are the large number of attributes that may be necessary and the restriction of the language of target programs to determinate Datalog clauses.

The FOIL-I system [IKI$^+$96] is an ILP system that learns from incomplete information in the training set but not in the background knowledge, with a particular emphasis on learning recursive predicates. In [WD95] the authors propose several frameworks for learning from partial interpretations. A particular framework that can learn form incomplete information is that of learning from satisfiability [DRD96c]. This framework is more general than ACL as both the examples and the hypotheses can be clausal theories. On the other hand, theories learned by this framework correspond only to the integrity constraints part of an abductive theory with no (or a trivial default) rule part.

A problem that is related to learning from incomplete data is that of learning from noisy or in general imperfect data [Qui90a, DB92, LDB96]. This problem is handled by relaxing the requirements of consistency and completeness in the necessity and sufficiency stopping criteria and by adopting special heuristic functions for guiding the search (see section 3.2.6. Relaxing the sufficiency stopping criterion is particularly useful when learning from incomplete data. It effectively allows us to avoid the coverage of some of the positive examples for which insufficient background data is given. For example, mFOIL ([DB92], see also section 3.4.3) stops adding a clause to the theory when too few positive examples are left for a generated clause to be significant or when no significant clause can be found with expected accuracy greater than the default. Instead, FOIL ([Qui90a], see also section 3.4.2) FOIL stops generating clauses when all the literals that can be added to the current clause require more than the available number of bits.

In general, these systems see incompleteness as special case of noise and hence it may be that methods for handling noise are too coarse for incompleteness. Indeed, when we know in which predicates the incompleteness lies, then we would expect that we can use more specialised techniques, like the ACL framework, to get better results than the more general methods for noise. This is confirmed by some of the experiments presented in section 4.6.

As we have seen, ACL can use integrity constraints as part of its background knowledge. Learning from integrity constraints was first examined in [DRB92a] and [DR92]. Recently, the system Progol [Mug95a] is able to learn from integrity constraints. However, in these cases, integrity constraints are used to impose conditions on the target predicates that need to be respected by the learned clauses. In ACL, instead, constraints impose conditions on background rather than target predicates and are used to restrict the assumptions of background facts rather than for specializing the clauses.

On the other hand, ACL also learns new integrity constraints as part of its final learned theory. Hence ACL involves a combination of learning from entailment and learning from interpretations. Although several ILP systems (e.g. [Mug95a, DRL95, DRB93]) can produce theories that combine rules and integrity constraints, all of these use a single form of induction to generate both parts of the theory. Finally, we point out that, as abductive theories are non-monotonic in nature, ACL can provide us with a form of non-monotonic learning. It can thus be used to address similar learning problems as those tackled in the work of [Hel89, BM91, DK95].

4.8 Conclusions

The chapter presents the new learning framework of Abductive Concept Learning (ACL), setting up its theoretical foundations and developing a first system for it. This framework integrates abduction and induction extending the Inductive Logic Programming paradigm in order to learn abductive theories: both the background and target theories are abductive theories and deductive entailment as the coverage relation in ILP is replaced by an abductive entailment in the learning problem of ACL. The main application of ACL is learning from incomplete information.

The ACL problem can be decomposed into two subproblems, ACL1 and ACL2, the first consisting of learning the rule part of the abductive theory and the second consisting of learning the constraint part. ACL1 is a learning from entailment problem, while ACL2 is a learning from interpretations problem. Based on this decomposition, a system for learning in this new framework has been developed that solves the ACL problem by first solving ACL1 and then ACL2. These separate problems are solved using and adapting algorithms and techniques from the existing ILP frameworks of learning from entailment and learning from interpretations. In this way, ACL represents a non-trivial and useful integration of these two main ILP settings.

The ACL framework allows us also to tackle effectively the problem of multiple predicate learning, where each predicate is required to be learned from the incomplete data for the other predicates. By employing abduction we are able to link the learning of the different predicates and ensure the coherence among the definitions learned for them. A separated system for multiple predicate learning, called M-ACL, has been developed by suitably modifying the ACL system.

Several experiments were performed, some of which were drawn from real-life problems of analyzing market research questionnaires, to test ACL on problems of learning from incomplete information. The performances of ACL were comparable or superior to those of FOIL, mFOIL and the ICL-Sat system adapted from ICL for learning with partial interpretations. Other experiments were also done that confirmed the ability of M-ACL to learn multiple predicates.

The development of the ACL algorithm and system in this chapter was heavily based on the separation of the full ACL problem into the ACL1 and ACL2 subproblems, adapting traditional ILP techniques to solve these. Further work is needed to examine other ways of synthesizing these subproblems and more importantly to develop algorithms that would search directly the full space of abductive theories. This involves the definition of generality orderings for this space and the development of suitable refinement operators that would allow the simultaneous learning of both parts (rules and constraints) of an abductive theory.

Chapter 5

Learning in a Three-valued Setting

5.1 Introduction

Most work on inductive concept learning considers a two-valued setting. In such a setting, what is not entailed by the learned theory is considered false, on the basis of the Closed World Assumption (CWA) [Rei78]. However, in practice, it is more often the case that we are confident about the truth or falsity of only a limited number of facts, and are not able to draw any conclusion about the remaining ones, because the available information is too scarce. Like it has been pointed out in [DRB90, DR92], this is typically the case of an autonomous agent that, in an incremental way, gathers information from its surrounding world. Such an agent needs to distinguish between what is true, what is false and what is unknown, and therefore needs to learn within a richer three-valued setting.

The class of *extended logic programs* is particularly suited for representing information in a three-valued setting. Extended logic programs contain two kinds of negation: default negation plus a second form of negation, called *explicit*, whose combination has been recognized as very useful for knowledge representation. The adoption of extended logic programs allows one to represent exceptions through default negation, as well as with verily *negative* information through explicit negation [PA92, AP96, APP98]. For instance, in [AP96, BG94a, DPP97, DP98, LP98] it is shown how extended logic programs are applicable to such diverse domains of knowledge representation as concept hierarchies, reasoning about actions, belief revision, counterfactuals, diagnosis, updates and debugging.

This chapter is based on the work presented in [LRP88b, LRP88a, LRP88c]. We discuss various approaches and strategies that can be adopted in ILP for learning with extended logic programs. The learning process starts from a set of positive and negative examples plus some background knowledge in the form of an extended logic programs. Positive and negative information in the training set are treated equally, by learning a definition for both a positive concept p and its (explicitly) negated concept $\neg p$. Coverage of examples is tested by adopting the SLX interpreter for extended logic programs under the Well-Founded Semantics with explicit negation (*WFSX*) defined in [AP96, DPP97], and valid for its paraconsistent version [DP98].

Default negation is used in the learning process to handle *exceptions* to general rules. Exceptions to a positive concept are identified from negative examples, whereas exceptions to a negative concept are identified from positive examples. A definition for the class of exceptions is then learned which may include new exceptions. The process is then iterated thus possibly producing a hierarchy of exceptions.

We adopt standard ILP techniques to learn one concept and its opposite. Depending on the technique adopted, one can learn the most general or the least general definition for each concept. Accordingly, four epistemological varieties occur, resulting from the mutual combination of most and least general solutions for the positive and negative concept. These possibilities are expressed via extended logic programs, and we discuss some of the factors that should be taken into account when choosing the level of generality of each, and their combination, to define a specific learning strategy, and how to cope with contradictions.

Indeed, separately learned positive and negative concepts may conflict and, in order to handle possible *contradiction*, contradictory learned rules are defused by making the learned definition for a positive concept p depend on the default negation of the negative concept $\neg p$, and vice-versa, i.e., each definition is introduced as an exception to the other. This way of coping with contradiction can be generalized for multiple source learning, and modified in order to take into account preferences among multiple learning agents or information sources. Moreover, we discuss how detecting different kinds of uncovered atoms points to different opportunities for theory extension.

The chapter is organized as follows. We first provide some basic notions on extended logic programs in section 5.2 and introduce the new ILP framework in section 5.3. We then examine, in section 5.4, factors to be taken into account when choosing the level of generality of learned theories. Section 5.5 proposes how to avoid inconsistencies on unseen atoms and their opposites, through the use of mutually defusing ("non-deterministic") rules, for the case of single and multiple learning agents, and how to incorporate exceptions through negation by default. Section 5.6 discusses how to identify diverse inconsistent or undefined cases in order to refine or extend learnt definitions. A description of the algorithm for learning extended logic programs hierarchies with exceptions follows next, together with an example of its behaviour, in section 5.7, and the overall system implementation in section 5.8. Finally, we examine related works in section 5.9, and conclude.

5.2 Preliminaries

In this section, we first discuss the usefulness of three-valuedness and of two types of negation for knowledge representation and then we provide some basic notions on extended logic programs and on WFSX.

5.2.1 Three-valuedness, default and explicit negation

In order to represent negative information, *default negation* [EK89, Dun91] was introduced by AI researchers via Logic Programming. The default negation of an atom P, "*not* P", may be read, variously, as "P is not provable", or "the falsity of P is assumable", or "the falsity of P is abducible", or "there is no evidence for P", or "there is no argument for P". Default negation allows us to deal with lack of information, a common situation in the real world. It introduces non-monotonicity into knowledge representation. Indeed, conclusions might not be solid because the rules leading to them may be defeasible. Legal texts, regulations, and courts employ this form of negation abundantly, as they perforce deal with open worlds.

For instance, we don't normally have explicit information about who is or is not the lover of whom, though that kind of information may arrive unexpectedly. Thus we write:

$$faithful(H, K) \leftarrow married(H, K), not\ lover(H, L)$$

I.e., if we have no evidence to conclude $lover(H, L)$ for some L given H , we can assume it false for all L given H.

The issue arises because often, particularly in data and knowledge bases, the Closed World Assumption (CWA) is enforced: everything that is not explicitly represented as positive is considered as negative. However, this introduces an asymmetry in knowledge representation, since negative information is only representable as the negation of positive one, we are not able to explicitly represent negative information that we may have obtained from the surrounding world. Therefore, a new form of negation, called *explicit negation* [PA92] and represented with $\neg$, is needed in order to restore the symmetry.

In some cases, we may have no factual or derivable either positive or negative information and we'd like to be able to say that both are false epistemically, i.e. from the "knowledge we possess" point of view. Accordingly, the excluded middle postulate, stating that any predication is either true or false, is unacceptable because some predication and its explicit negation may be false simultaneously. Therefore, explicit negation '$\neg$' differs from classical negation because it does not comply with the excluded middle postulate.

By means of this form of negation we are also able to adopt the CWA in a symmetrical way, i.e., to assume as true what is not explicitly represented as false.

For example, we are able to write

$$\neg faithful(H, K) \leftarrow married(H, K), not\ \neg lover(H, L)$$

to model instead a world where people are unfaithful by default or custom, and where it is required to explicitly prove that someone does not take any lover before concluding that person not unfaithful. Here $not\ \neg lover(H, L)$ is true by CWA unless $\neg lover(H, L)$ is proven true. This can be understood as assuming $lover(H, L)$ true.

More precisely, we can state the CWA for just those predicates P or $\neg Q$ we wish, simply by writing:

$$\neg P \leftarrow not\ P \quad \text{or} \quad Q \leftarrow not\ \neg Q$$

Alternatively, use of $not\ P$ or of $not\ \neg Q$ assumption literals may be made at just those predicate occurrences so requiring it.

Let us next examine the need for revising assumptions and of introducing a third truth-value, call it "undefined", into our framework.

When we combine the viewpoints of the two above worlds we become confused:

$$faithful(H, K) \leftarrow married(H, K), not\ lover(H, L)$$

$$\neg faithful(H, K) \leftarrow married(H, K), not\ \neg lover(H, L)$$

If we have no evidence for $lover(H, L)$ nor for $\neg lover(H, L)$, we could assume both of them as false. However, supposing that $married(H, K)$ is true for some H and K, it now appears that both

$$faithful(H,K) \quad and \quad \neg faithful(H,K)$$

are contradictorily true.

In this case the assumptions of falsity of $lover(H, L)$ and for $\neg lover(H, L)$ has led to a contradiction. But when an assumption leads to contradiction one should retract it. It is the venerable principle of *reductio ad absurdum*, or "reasoning by contradiction".

In our case, the two assumptions that led to the contradiction are on equal footing. Given no other, possibly preferential information, we retract both because we cannot decide between them. That is, we assume neither $lover(H, L)$ nor $\neg lover(H, L)$ false. Since neither is provably true either, we make each *undefined*, i.e., we introduce a third truth-value to better characterize this lack of information about some lovers' situation. This imposition of undefinedness can be achieved simply, by adding to our knowledge:

$$\neg lover(H, L) \leftarrow not\ lover(H, L)$$

$$lover(H, L) \leftarrow not\ \neg lover(H, L)$$

Given no other information, we can prove neither of $lover(H, L)$ nor $\neg lover(H, L)$ true, or false. Any attempt to do it runs into a self-referential circle involving default negation. Thus, $lover(H, L)$ and $\neg lover(H, L)$ are assigned the truth-value undefined and, as a consequence, $faithful(H, K)$ and $\neg faithful(H, K)$ are undefined too.

However, if we hypothesize one of $lover(H, L)$ nor $\neg lover(H, L)$ true, the other ipso facto becomes false, and vice-versa. These two possible situations are thus not ruled out. But the safest, skeptical, third option is to take no side in this marital dispute, and abstain from believing either.

Indeed, the *WFSX* [AP96, DPP97] semantics assigns to the literals in the above two clauses the truth value *undefined*, in its knowledge skeptical well-founded model, but allows also for the other two, non truth-minimal, more credulous models.

When dealing with non-provability one really needs a third truth-value to express our epistemic inability to come up with information.

In any case, we are in wont of a third logical value for other reasons. As we build up our real-world imperfect knowledge base, we may very well create, unwittingly and unawares, circular dependencies as above. For example, the Legislator may well enact conflicting, circular, laws. Still, we want to be able to carry on reasoning, whether or not such circularities legitimately express what they model.

5.2.2 Extended Logic Programs

An *extended logic program* is a finite set of rules of the form:

$$L_0 \leftarrow L_1, \ldots, L_n$$

with $n \geq 0$, where L_0 is an objective literal, $L_1, \ldots, L_n$ are literals and each rule stands for the sets of its ground instances. *Objective literals* are of the form A or $\neg A$, where A is an atom, while a *literal* is either an objective literal L or its default negation *not* L. $\neg A$ is said the *opposite* literal of A (and vice versa), where $\neg\neg A = A$, and *not* A the *complementary* literal of A (and vice versa). By *not* $\{a_1, \ldots, a_n\}$ we mean $\{not\ a_1, \ldots, not\ a_n\}$. The set of all objective literals of a program P is called its *extended Herbrand base* and is represented as $H^E(P)$. An *interpretation* I of an extended program P is denoted by $T \cup not\ F$, where T and F are disjoint subsets of $H^E(P)$. Objective literals in T are said to be *true* in I, objective literals in F are said to be *false* in I and in $H^E(P) - I$ *undefined* in I. We introduce in the language the proposition $\mathbf{u}$ that is undefined in every interpretation I.

The *WFSX* extends the well founded semantics (*WFS*) [VGRS91] for normal logic programs to the case of extended logic programs. *WFSX* is obtained from *WFS* by adding the coherence principle relating the two forms of negation: "*if L is an objective literal and $\neg L$ belongs to the model of a program, then also not L belongs to the model*".

The definition of *WFSX* that follows is taken from [ADP94] and is based on the alternating fix points of Gelfond-Lifschitz Γ-like operators.

Definition 53 (The Γ-operator) *Let P be an extended logic program and let I be an interpretation. $\Gamma_P(I)$ is the program obtained from P by performing in the sequence the following four operations:*

- *Remove from P all rules containing a default literal $L = not\ A$ such that $A \in I$.*
- *Remove from P all rules containing in the body an objective literal L such that $\neg L \in I$.*
- *Remove from all remaining rules of P their default literals $L = not\ A$ such that $not\ A \in I$.*
- *Replace all the remaining default literals by proposition* $\mathbf{u}$.

In order to impose the coherence requirement, we need the following definition.

Definition 54 (Seminormal Version of a Program) *The seminormal version of a program P is the program P_s obtained from P by adding to the (possibly empty) Body of each rule $L \leftarrow Body$ the default literal $not \neg L$, where $\neg L$ is the complement of L with respect to explicit negation.*

In the following, we will use the following abbreviations: $\Gamma(S)$ for $\Gamma_P(S)$ and $\Gamma_s(S)$ for $\Gamma_{P_s}(S)$.

Definition 55 (Partial Stable Model) *An interpretation $T \cup not\ F$ is called a* partial stable model *of P iff $T = \Gamma\Gamma_s T$ and $F = H^E(P) - \Gamma_s T$.*

Partial stable models are an extension of *stable models* [GL88] for extended logic programs and a three-valued semantics. Not all programs have a partial stable model (e.g. $P = \{a, \neg a\}$) and programs without a partial stable model are called *contradictory*.

Theorem 56 (*WFSX* Semantics) *Every non-contradictory program P has a least (with respect to $\subseteq$) partial stable model, the well-founded model of P denoted by $WFM(P)$.*

To obtain an iterative "bottom-up" definition for $WFM(P)$ we define the following transfinite sequence $\{I_\alpha\}$:

$$I_0 = \{\}; \quad I_{\alpha+1} = \Gamma\Gamma_S I_\alpha; \quad I_\delta = \bigcup\{I_\alpha | \alpha < \delta\}$$

where δ is a limit ordinal. There exists a smallest ordinal λ for the sequence above, such that I_λ is the smallest fix point of $\Gamma\Gamma_S$. Then, $WFM(P) = I_\lambda \cup not\ (H^E(P) - \Gamma_S I_\lambda)$.

Let us now show an example of the *WFSX* semantics in the case of a simple program.

Example 57 *Consider the following extended logic program:*

$$\begin{array}{lcl} \neg a & \leftarrow & . \\ a & \leftarrow & b. \end{array} \qquad\qquad b \leftarrow not\ b.$$

A WFSX *model of this program is* $M = \{\neg a, not\ \neg b, not\ a\}$*:* $\neg a$ *is true,* a *is false (i.e., both* $\neg a$ *and* $not\ a$ *are in the well-founded model),* $\neg b$ *is false (there are no rules for* $\neg b$*) and* b *is undefined. Notice that* $not\ a$ *is in the model since it is implied by* $\neg a$ *via the coherence principle.*

One of the most important characteristic of *WFSX* is that it provides a semantics for an important class of extended logic programs: the set of *non-stratified* programs, i.e., the set of programs that contain recursion through default negation. An extended logic program is *non-stratified* if its *dependency graph* does not contain any cycle with an arc labelled with $-$. The *dependency graph* of a program P is a labelled graph with a node for each predicate of P and an arc from a predicate p to a predicate q if q appears in the body of clauses with p in the head. The arc is labelled with $+$ if q appears in an objective literal in the body and with $-$ if it appears in a default literal.

Non-stratified programs are very useful for knowledge representation because the *WFSX* semantics assigns the truth value undefined to the literals involved in the recursive cycle through negation, as shown above for $lover(H, L)$ and $\neg lover(H, L)$. In section 5.5 we will employ non stratified programs in order to resolve possible contradictions.

WFSX was chosen among the other semantics for extended logic programs, answer-sets [GL90] and three-valued strong negation [APP98], because none of the others enjoys the property of relevance [AP96, APP98] for non-stratified programs, i.e., they cannot have top-down querying procedures for non-stratified programs. Instead, for *WFSX* there exists a top-down proof procedure *SLX* [AP96], which is correct with respect to the semantics[1]. Cumulativity is also enjoyed by *WFSX*, i.e., if you add a lemma then the semantics does not change. This property is important for speeding-up the implementation. By memorizing intermediate lemmas through tabling, the implementation of *SLX* greatly improves. Answer-set semantics, however, is not cumulative for non-stratified programs and thus cannot use tabling.

The *SLX* top-down procedure for *WFSX* relies on two independent kinds of derivations: T-derivations, proving truth, and TU-derivations proving non-falsity, i.e., truth or undefinedness. Shifting from one to the other is required for proving a default literal $not\ L$: the T-derivation of $not\ L$ succeeds if the TU-derivation of L fails; the TU-derivation of $not\ L$ succeeds if the T-derivation of L fails. Moreover, the T-derivation of $not\ L$ also succeeds if the T-derivation of $\neg L$ succeeds, and the TU-derivation of L fails if the T-derivation of $\neg L$ succeeds (thus taking into account the *coherence principle*).

The *SLX* procedure is amenable to a simple pre-processing implementation, by mapping *WFSX* programs into *WFS* programs through the T-TU transformation [DP97]. This transformation is linear and essentially doubles the number of program clauses. Then, the

[1]Though *WFSX* is not truth-functional (i.e., the truth-value of any formula does not depend only on the truth-value of its subformulas as expressed by the truth table of the logical connectives) any extended logic program under *WFSX* can be transformed into an equivalent program under *WFS* through the T-TU transformation [DP97, AP96] which is truth-functional. This transformation is used for the implementation.

transformed program can be executed in *XSB* [SSW+97], an efficient Logic Programming system which implements the *WFS* with tabling, and subsumes Prolog. Tabling in *XSB* consists in memorizing intermediate lemmas, and in properly dealing with non-stratification according to *WFS*. Tabling is important in learning, where computations are often repeated for testing the coverage or otherwise of examples.

5.3 Learning in a Three-valued Setting

In real-world problems, complete information about the world is impossible to achieve and it is necessary to reason and act on the basis of the available partial information. In situations of incomplete knowledge, it is important to distinguish between what is true, what is false, and what is unknown or undefined.

Such situation occurs, for example, when an agent incrementally gathers information from the surrounding world and has to select its own actions on the basis of such acquired knowledge. If the agent learns in a two-valued setting, it can encounter the problems that have been highlighted in [DRB90]. When learning in a specific to general way, it will learn a cautious definition for the target concept and it will not be able to distinguish what is false from what is not yet known (see figure 5.1a). Supposing the target predicate represents the allowed actions, then the agent will not distinguish forbidden actions from actions with an outcome and this can restrict the agent acting power. If the agent learns in a general to specific way, instead, it will not know the difference between what is true and what is unknown (figure 5.1b) and, therefore, it can try actions with an unknown outcome. Rather, by learning in a three-valued setting, it will be able to distinguish between allowed actions, forbidden actions, and actions with an unknown outcome (figure 5.1c). In this way, the agent will know which part of the domain needs to be further explored and will not try actions with an unknown outcome unless it is trying to expand its knowledge.

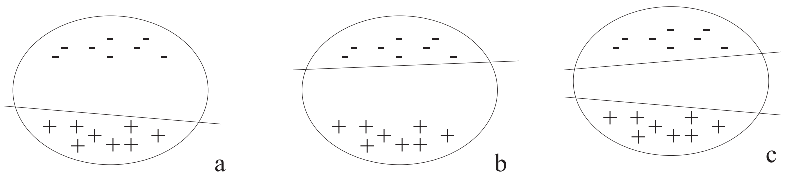

Figure 5.1: (taken from [DRB90])(a,b): two-valued setting, (c): three-valued setting

Learning in a three-valued setting requires the adoption of a more expressive class of programs to be learned. This class can be represented, we have seen, by means of extended logic programs under the well-founded semantics extended with explicit negation *WFSX* [AP96, APP98, PA92].

We consider a new learning problem where we want to learn an extended logic program from a background knowledge that is itself an extended logic program and from a set of positive and a set of negative examples in the form of ground facts for the target predicates. A learning problem for extended logic programs was first introduced in [IK97] where the notion of coverage was defined by means of truth in the answer-set semantics. Here the problem definition is modified to consider coverage as truth in the *WFSX* semantics

Definition 58 (Learning Extended Logic Programs)
Given:

- *a set $\mathcal{P}$ of possible (extended logic) programs*
- *a set E^+ of positive examples (ground facts)*
- *a set E^- of negative examples (ground facts)*
- *a consistent extended logic program B (*background knowledge*)*

Find:

- *an extended logic program $P \in \mathcal{P}$ such that*
  - $\forall e \in E^+, \neg E^-,\ B \cup P \models_{WFSX} e$ *(*completeness*)*
  - $\forall e \in \neg E^+, E^-,\ B \cup P \not\models_{WFSX} e$ *(*consistency*)*

  where $\neg E = \{\neg e | e \in E\}$.

The theory that is learned will contain rules of the following form:

$$p(\vec{X}) \leftarrow Body^+(\vec{X})$$

$$\neg p(\vec{X}) \leftarrow Body^-(\vec{X})$$

for every target predicate p, where $\vec{X}$ stands for a tuple of arguments. In order to satisfy the completeness requirement, the rules for p will entail all positive examples while the rules for $\neg p$ will entail all (explicitly negated) negative examples. The consistency requirement is satisfied by ensuring that both sets of rules do not entail instances of the opposite element in either of the training sets.

Note that, in the case of extended logic programs, the consistency with respect to the training set is equivalent to the requirement that the program is non-contradictory on the examples. This requirement is enlarged to require that the program be consistent also for unseen atoms, i.e., $B \cup P \not\models L \wedge \neg L$ for every atom L of the target predicates.

We say that an example e is *covered* by program P if $P \models_{WFSX} e$. Since the *SLX* procedure is correct with respect to *WFSX*, even for contradictory programs, coverage of examples is tested by verifying whether $P \vdash_{SLX} e$.

Our approach to learning with extended logic programs consists in initially applying conventional ILP techniques to learn a positive definition from E^+ and E^- and a negative definition from E^- and E^+. In these techniques, the *SLX* procedure substitutes the standard proof procedure of Logic Programming to test the coverage of examples.

The ILP techniques to be used depend on the level of generality that we want to have for the two definitions: we can look for the Least General Solution (LGS) or the Most General Solution (MGS) of the problem of learning each concept and its complement (see section 3.2.7 for a definition of LGS and MGS).

LGSs can be found by adopting one of the bottom-up methods such as relative least general generalization (*rlgg*) [Plo70] (see section 3.2.5) and the GOLEM system [MF90] (see section 3.4.1), inverse resolution [MB92] or inverse entailment [LM92]. Conversely, MGSs can be found by adopting a top-down refining method (see section 3.2.6) and a system such as FOIL [Qui90b] (see section 3.4.2) or Progol [Mug95a].

5.4 Strategies for Combining Different Generalizations

The generality of concepts to be learned is an important issue when learning in a three-valued setting. In a two-valued setting, once the generality of the definition is chosen, the extension (i.e., the generality) of the set of false atoms is, we've seen, undesirably automatically decided, because it is the complement of the true atoms set. In a three-valued setting, rather, the extension of the set of false atoms depends on the generality of the definition learned for the negative concept. Therefore, the corresponding level of generality may be chosen independently for the two definitions, thus affording four epistemological cases.

Furthermore, the generality of the solutions learned for the positive and negative concepts clearly influences the interaction between the definitions. If we learn the MGS for both a concept and its opposite, the probability that their intersection is non-empty is higher than if we learn the LGS for both. Accordingly, the decision as to which type of solution to learn should take into account the possibility of interaction as well: if we want to reduce this possibility, we have to learn two LGS, if we do not care about interaction, we can learn two MGS. In general, we may learn different generalizations and combine them in distinct ways for different strategic purposes within the same application problem.

The choice of the level of generality should be made on the basis of available knowledge about the domain. Two of the criteria that can be taken into account are the damage or risk that may arise from an erroneous classification of an unseen object, and the confidence we have in the training set as to its correctness and representativeness.

When classifying an as yet unseen object as belonging to a concept, we may later discover that the object belongs to the opposite concept. The more we generalize a concept, the higher is the number of unseen atoms covered by the definition and the higher is the risk of an erroneous classification. Depending on the damage that may derive from such a mistake, we may decide to take a more cautious or a more confident approach. If the possible damage from an over extensive concept is high, then one should learn the LGS for that concept, if the possible damage is low then one can generalize the most and learn the MGS. The overall risk will depend too on the use of the learned concepts within other rules.

As regards the confidence in the training set, we can prefer to learn the MGS for a concept if we are confident that examples for the opposite concept are correct and representative of the concept. In fact, in top-down methods, negative examples are used in order to delimit the generality of the solution. Otherwise, if we think that examples for the opposite concept are not reliable, then we should learn the LGS.

In the following, we present a realistic example of the kind of reasoning that can be used to choose and specify the preferred level of generality, and discuss how to strategically combine the different levels by employing the extended Logic Programming approach to learning.

Example 59 *Consider a person living in a bad neighbourhood in Los Angeles. He is an honest man and to survive he needs two concepts, one about who is likely to attack him, on the basis of appearance, gang membership, age, past dealings, etc. Since he wants to take a cautious approach, he maximizes attacker and minimizes $\neg attacker$, so that his $attacker1$ concept allows him to avoid dangerous situations.*

$attacker1(X) \leftarrow attacker_{MGS}(X)$

$\neg attacker1(X) \leftarrow \neg attacker_{LGS}(X)$

Another concept he needs is the type of beggars he should give money to (he is a good man) that actually seem to deserve it, on the basis of appearance, health, age, etc. Since he is not rich and does not like to be tricked, he learns a $beggar1$ *concept by minimizing* $beggar$ *and maximizing* $\neg beggar$*, so that his beggar concept allows him to give money strictly to those appearing to need it without faking.*

$beggar1(X) \leftarrow beggar_{LGS}(X)$

$\neg beggar1(X) \leftarrow \neg beggar_{MGS}(X)$

However rejected beggars, especially malicious ones, may turn into attackers, in this very bad neighbourhood. Consequently, if he thinks a beggar might attack him he had better be more permissive about who is a beggar and placate him with money. In other words, he should maximize $beggar$ *and minimize* $\neg beggar$ *in a* $beggar2$ *concept.*

$beggar2(X) \leftarrow beggar_{MGS}(X)$

$\neg beggar2(X) \leftarrow \neg beggar_{LGS}(X)$

These concepts can be used in order to minimize his risk taking when he carries, by his standards, a lot of money and meets someone who is likely to be an attacker, with the following kind of reasoning:

$run(X) \leftarrow lot\_of\_money(X), meets(X,Y), attacker1(Y), not\ beggar2(Y)$

$\neg run(X) \leftarrow lot\_of\_money(X), give\_money(X,Y)$

$give\_money(X,Y) \leftarrow meets(X,Y), beggar1(Y)$

$give\_money(X,Y) \leftarrow meets(X,Y), attacker1(Y), beggar2(Y)$

If he does not have a lot of money on him, he may prefer not to run as he risks being beaten up. In this case he has to relax his attacker concept into $attacker2$*, but not relax it so much that he would use* $\neg attacker_{MGS}$.

$\neg run(X) \leftarrow little\_money(X), meets(X,Y), attacker2(Y)$

$attacker2(X) \leftarrow attacker_{LGS}(X)$

$\neg attacker2(X) \leftarrow \neg attacker_{LGS}(X)$

The various notions of attacker and beggar are learnt on the basis of previous experience the man has had. In the following, we show, through a simple background knowledge and training set, how such concepts can be learned.

Example 60 *(cont'd) Consider the case in which we have a background knowledge containing the following general rules:*

$animal(X) \leftarrow person(X)$	$person(X) \leftarrow man(X)$
$animal(X) \leftarrow dog(X)$	$person(X) \leftarrow woman(X)$

in addition to which we know some facts about a number of instances (male or female, person or dog) we have encountered in the past that have been classified as $attackers$ *or non* $attackers$*, and as* $beggars$ *or non* $beggars$.

$man(1)$	$\neg good\_appearance(1)$		$gang\_member(1)$
$man(2)$	$\neg good\_appearance(2)$	$age(2, adult)$	
$man(3)$	$\neg good\_appearance(3)$	$age(3, old)$	
$woman(4)$	$\neg good\_appearance(4)$	$age(4, old)$	
$man(5)$	$good\_appearance(5)$	$age(5, adult)$	$\neg healthy(5)$
$man(6)$	$\neg good\_appearance(6)$	$age(6, youth)$	
$man(7)$	$good\_appearance(7)$	$age(7, adult)$	

$woman(8)$	$good\_appearance(8)$	$age(8, old)$	
$man(9)$		$age(9, youth)$	
$woman(10)$		$age(10, youth)$	
$dog(11)$	$\neg good\_appearance(11)$	$age(11, old)$	
$woman(12)$	$\neg good\_appearance(12)$	$age(12, adult)$	$\neg healthy(12)$
$man(13)$		$age(13, old)$	$healthy(13)$
$man(14)$	$good\_appearance(14)$	$age(14, adolescent)$	
$man(15)$	$\neg good\_appearance(15)$		$gang\_member(15)$
$man(16)$	$\neg good\_appearance(16)$	$age(16, adult)$	
$man(17)$	$good\_appearance(17)$	$age(17, adult)$	$\neg healthy(17)$
$man(18)$	$good\_appearance(18)$	$age(18, adult)$	
$dog(19)$	$\neg good\_appearance(19)$	$age(19, old)$	
$man(20)$		$age(20, old)$	$healthy(20)$
$woman(21)$	$good\_appearance(21)$	$age(21, adolescent)$	

Let the training set for the attacker and beggar concepts be:

$$
\begin{aligned}
E^+ &= \{\ attacker(1), attacker(2), attacker(15), attacker(16),\\
&\quad\ beggar(3), beggar(4), beggar(5), beggar(17), beggar(12)\}\\
E^- &= \{\ attacker(3), attacker(4), attacker(7), attacker(18), attacker(8),\\
&\quad\ attacker(9), attacker(10), attacker(11), attacker(19),\\
&\quad\ beggar(11), beggar(19), beggar(13), beggar(14), beggar(20), beggar(21)\}
\end{aligned}
$$

Then, most general and least general solutions can be computed. By using the system GOLEM [MF90], we obtained the following results. For the positive and negative concepts of attacker:

$$
\begin{aligned}
attacker_{MGS}(X) &\leftarrow gang\_member(X)\\
attacker_{MGS}(X) &\leftarrow \neg good\_appearance(X), age(X, adult)\\[1ex]
attacker_{LGS}(X) &\leftarrow gang\_member(X), man(X), animal(X),\\
&\qquad person(X), \neg good\_appearance(X)\\
attacker_{LGS}(X) &\leftarrow \neg good\_appearance(X), man(X), animal(X),\\
&\qquad person(X), age(X, adult)\\[1ex]
\neg attacker_{MGS}(X) &\leftarrow good\_appearance(X)\\
\neg attacker_{MGS}(X) &\leftarrow age(X, youth)\\
\neg attacker_{MGS}(X) &\leftarrow age(X, old)\\[1ex]
\neg attacker_{LGS}(X) &\leftarrow age(X, adult), man(X), animal(X), person(X),\\
&\qquad good\_appearance(X)\\
\neg attacker_{LGS}(X) &\leftarrow age(X, youth), animal(X), person(X)\\
\neg attacker_{LGS}(X) &\leftarrow age(X, old), animal(X)
\end{aligned}
$$

and for those of beggar:

$$
\begin{aligned}
beggar_{MGS}(X) &\leftarrow age(X, adult)\\
beggar_{MGS}(X) &\leftarrow person(X), \neg good\_appearance(X)
\end{aligned}
$$

$$
\begin{array}{lcl}
beggar_{LGS}(X) & \leftarrow & age(X, adult), man(X), animal(X), person(X), \\
 & & \neg healthy(X), good\_appearance(X) \\
beggar_{LGS}(X) & \leftarrow & person(X), \neg good\_appearance(X), age(X, B), animal(X)
\end{array}
$$

$$
\begin{array}{lcl}
\neg beggar_{MGS}(X) & \leftarrow & dog(X) \\
\neg beggar_{MGS}(X) & \leftarrow & healthy(X) \\
\neg beggar_{MGS}(X) & \leftarrow & age(X, adolescent)
\end{array}
$$

$$
\begin{array}{lcl}
\neg beggar_{LGS}(X) & \leftarrow & dog(X), age(X, old), animal(X), \neg good\_appearance(X) \\
\neg beggar_{LGS}(X) & \leftarrow & healthy(X), age(X, old), man(X), animal(X), person(X) \\
\neg beggar_{LGS}(X) & \leftarrow & age(X, adolescent), good\_appearance(X), animal(X), person(X)
\end{array}
$$

Notice that the positive and negative versions of a concept (despite the algorithm used to learn a definition for it) never overlap on training set instances, but they might overlap for atoms not belonging to the training set. The latter situation requires program refining in order to eliminate contradictions, as shown next.

5.5 Strategies for Eliminating Learned Contradictions

Both in single and multi-agent learning, we shall see, the definitions of the positive and negative concepts may overlap. Conflicting rules for a predicate and its explicit negation may originate in the same knowledge source, or in combining rules obtained from distinct knowledge sources or on distinct occasions. In the sequel, we deal with the problem of removing contradiction in such cases.

5.5.1 Single Source Contradiction

Even for a single agent, the definitions of the positive and negative concepts may overlap. In this case, we have a contradictory classification for the objective literals in the intersection. In order to resolve the conflict, we must distinguish two types of literals in the intersection: those that belong to the training set and those that do not, also dubbed *unseen* atoms (see figure 5.2).

Example 61 *(cont'd) Let the person living in Los Angeles be now travelling to Brazil, where youth gangs are known for attacks on tourists. For the unseen instance:*

$$man(22) \qquad \neg good\_appearance(22) \qquad age(22, youth) \qquad gang\_member(22)$$

the person concludes both that instance 22 is an attacker and a non attacker as well, since $attacker_{LGS}(22)$ *(alternatively* $attacker_{MGS}(22)$*) and* $\neg attacker_{LGS}(22)$ *(alternatively* $\neg attacker_{MGS}(22)$*) are true. Thus, contradiction arises for* $attacker$ *and* $\neg attacker$*.*

In the following, we discuss how to resolve the conflict in the case of unseen literals and of literals in the training set. We first consider the case in which the training sets are disjoint and we later extend the scope to the case where there is a non-empty intersection of the training sets, when they are less than perfect. From now onwards, $\vec{X}$ stands for a tuple of arguments.

Contradiction on Unseen Literals For unseen literals, the conflict is resolved by classifying them as undefined, since the arguments supporting the two classifications are equally strong. Instead, for literals in the training set, the conflict is resolved by giving priority to the classification stipulated by the training set. In other words, literals in a training set that are covered by the opposite definition are made as *exceptions* to that definition. For unseen literals in the intersection, the undefined classification is obtained by making

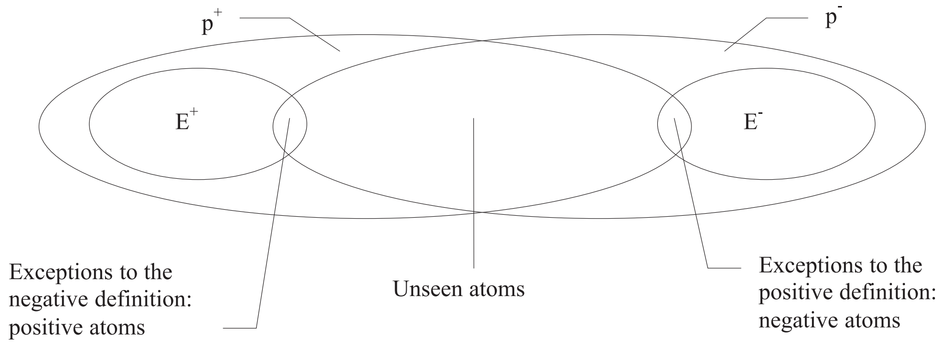

Figure 5.2: Interaction of the positive and negative definitions on exceptions.

opposite rules mutually defeasible, or "non-deterministic" (see [BG94a, AP96]). The target theory is consequently expressed in the following way:

$$\begin{array}{rcl} p(\vec{X}) & \leftarrow & p^{+}(\vec{X}), not\ \neg p(\vec{X}) \\ \neg p(\vec{X}) & \leftarrow & p^{-}(\vec{X}), not\ p(\vec{X}) \end{array}$$

where $p^{+}(\vec{X})$ and $p^{-}(\vec{X})$ are, respectively, the definitions learned for the positive and the negative concept, obtained by renaming the positive predicate by p^{+} and its explicit negation by p^{-}. From now onwards, we will indicate with these superscripts the definitions learned separately for the positive and negative concepts.

We want $p(\vec{X})$ and $\neg p(\vec{X})$ each to act as an exception to the other. In case of contradiction, this will introduce mutual circularity, and hence undefinedness according to *WFSX*. For each literal in the intersection of p^{+} and p^{-}, there are two stable models, one containing the literal in its three-valued version, the other containing the opposite literal. According to *WFSX*, there is a third (partial) stable model where both literals are undefined, i.e., no literal $p(\vec{X})$, $\neg p(\vec{X})$, *not* $p(\vec{X})$ or *not* $\neg p(\vec{X})$ belongs to the well-founded (or least partial stable) model. The resulting program contains a recursion through negation (i.e., it is non-stratified) but the top-down *SLX* procedure does not go into a loop because it comprises mechanisms for loop detection and treatment, which are implemented by XSB through tabling.

Example 62 *Let us consider the Example of section 5.4. In order to avoid contradictions on unseen atoms, the learned definitions must be:*

$$\begin{array}{lcl} attacker1(X) & \leftarrow & attacker^{+}_{MGS}(X), not\ \neg attacker1(X) \\ \neg attacker1(X) & \leftarrow & attacker^{-}_{LGS}(X), not\ attacker1(X) \end{array}$$

$$
\begin{array}{lll}
beggar1(X) & \leftarrow & beggar^{+}_{LGS}(X), not\ \neg beggar1(X) \\
\neg beggar1(X) & \leftarrow & beggar^{-}_{MGS}(X), not\ beggar1(X) \\
beggar2(X) & \leftarrow & beggar^{+}_{MGS}(X), not\ \neg beggar2(X) \\
\neg beggar2(X) & \leftarrow & beggar^{-}_{LGS}(X), not\ beggar2(X) \\
attacker2(X) & \leftarrow & attacker^{+}_{LGS}(X), not\ \neg attacker2(X) \\
\neg attacker2(X) & \leftarrow & attacker^{-}_{LGS}(X), not\ attacker2(X)
\end{array}
$$

Note that $p^+(\vec{X})$ and $p^-(\vec{X})$ can display as well the undefined truth value, either because the original background is non-stratified or because they rely on some definition learned for another target predicate, which is of the form above and therefore non-stratified. In this case, three-valued semantics can produce literals with the value "undefined", and one or both of $p^+(\vec{X})$ and $p^-(\vec{X})$ may be undefined. If one is undefined and the other is true, then the rules above make both p and $\neg p$ undefined, since the negation by default of an undefined literal is still undefined. However, this is counter-intuitive: a defined value should prevail over an undefined one.

In order to handle this case, we suppose that a system predicate $undefined(X)$ is available[2], that succeeds if and only if the literal X is undefined. So we add the following two rules to the definitions for p and $\neg p$:

$$
\begin{array}{lll}
p(\vec{X}) & \leftarrow & p^+(\vec{X}), undefined(p^-(\vec{X})) \\
\neg p(\vec{X}) & \leftarrow & p^-(\vec{X}), undefined(p^+(\vec{X}))
\end{array}
$$

According to these clauses, $p(\vec{X})$ is true when $p^+(\vec{X})$ is true and $p^-(\vec{X})$ is undefined, and conversely.

Contradiction on Examples Theories are tested for consistency on all the literals of the training set, so we should not have a conflict on them. However, in some cases, it is useful to relax the consistency requirement and learn clauses that cover a small amount of counter examples. This is advantageous when it would be otherwise impossible to learn a definition for the concept, because no clause is contained in the language bias that is consistent, or when an overspecific definition would be learned, composed of very many specific clauses instead of a few general ones. In such cases, the definitions of the positive and negative concepts may cover examples of the opposite training set. These must then be considered exceptions and treated as abnormalities.

Exceptions may also be due to noise in the collection of data, or to abnormalities in the opposite concept. In the latter case, if exceptions form a class, it may be possible to learn a definition for it, provided that we have data on their common properties and the language bias so allows.

Let us start with the case where some literals covered by a definition belong to the opposite training set. We want of course to classify these according to the classification given by the training set, by making such literals *exceptions*. To handle exceptions to classification rules, we add a negative default literal of the form *not* $abnorm_p(\vec{X})$ (resp. *not* $abnorm_{\neg p}(\vec{X})$) to the rule for $p(\vec{X})$ (resp. $\neg p(\vec{X})$), to express possible abnormalities

[2] The *undefined* predicate can be implemented through negation NOT under CWA ($NOT\ P$ means that P is false whereas *not* means that P is false or undefined), i.e., $undefined(P) \leftarrow NOT\ P, NOT(not\ P)$.

arising from exceptions. Then, for every exception $p(\vec{t})$, an individual fact of the form $abnorm_p(\vec{t})$ (resp. $abnorm_{\neg p}(\vec{t})$) is asserted so that the rule for $p(\vec{X})$ (resp. $\neg p(\vec{X})$) does not cover the exception, while the opposite definition still covers it. In this way, exceptions will figure in the model of the theory with the correct truth value. The learned theory thus takes the form:

$$p(\vec{X}) \leftarrow p^+(\vec{X}), not\ abnorm_p(\vec{X}), not\ \neg p(\vec{X}) \tag{5.1}$$

$$\neg p(\vec{X}) \leftarrow p^-(\vec{X}), not\ abnorm_{\neg p}(\vec{X}), not\ p(\vec{X}) \tag{5.2}$$

$$p(\vec{X}) \leftarrow p^+(\vec{X}), undefined(p^-(\vec{X})) \tag{5.3}$$

$$\neg p(\vec{X}) \leftarrow p^-(\vec{X}), undefined(p^+(\vec{X})) \tag{5.4}$$

Abnormality literals have not been added to the rules for the undefined case because a literal which is an exception is also an example, and so must be covered by its respective definition; therefore it cannot be undefined.

Individual facts of the form $abnorm_p(\vec{X})$ are then used as examples for learning a definition for $abnorm_p$ and $abnorm_{\neg p}$, as in [IK97, LMMR97]. In turn, exceptions to the definitions of $abnorm_p$ and $abnorm_{\neg p}$ may be found and so on, thus leading to a hierarchy of exceptions.

Example 63 *Consider a domain containing entities a, b, c, d, e, f and suppose the target concept is $flies$. Let the background knowledge be:*

$bird(a)$	$has\_wings(a)$	
$jet(b)$	$has\_wings(b)$	
$angel(c)$	$has\_wings(c)$	$has\_limbs(c)$
$penguin(d)$	$has\_wings(d)$	$has\_limbs(d)$
$dog(e)$	$has\_limbs(e)$	
$cat(f)$	$has\_limbs(f)$	

and let the training set be:

$$E^+ = \{flies(a)\} \qquad E^- = \{flies(d), flies(e)\}$$

The learned theory is:

$$\begin{array}{rcl} flies(X) & \leftarrow & flies^+(X), not\ abnormal_{flies}(X), not\ \neg flies1(X) \\ \neg flies(X) & \leftarrow & flies^-(X), not\ flies1(X) \\ flies(X) & \leftarrow & flies^+(X), undefined(flies^-(X)) \\ \neg flies(X) & \leftarrow & flies^-(X), undefined(flies^+(X)) \\ abnormal_{flies}(d) & & \end{array}$$

where $flies^+(X) \leftarrow has\_wings(X)$ and $flies(X)^- \leftarrow has\_limbs(X)$ Moreover, the abnormality fact $abnormal_{flies}(d)$ can be generalized to obtain
$abnormal_{flies}(X) \leftarrow penguin(X)$

The example above and figure 5.3 show all the various cases for a literal when learning in a three-valued setting. a and e are examples that are consistently covered by the definitions. b

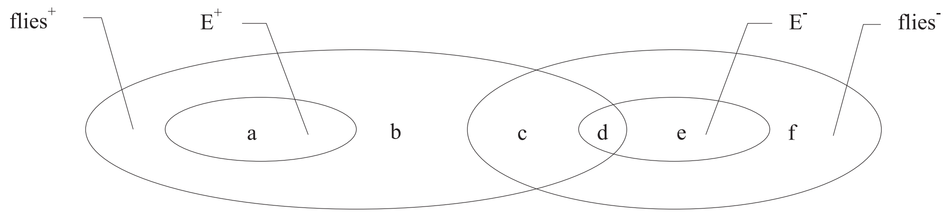

Figure 5.3: Coverage of definitions for opposite concepts

and f are unseen literals on which there is no contradiction. c and d are literals where there is contradiction, but c is classified as undefined whereas d is considered as an exception to the positive definition and is classified as negative.

extended logic programs can be used as well to represent n disjoint classes $p_1, \ldots, p_n$. When one has to learn n disjoint classes, the training set contains a number of facts for a number of predicates $p_1, \ldots, p_n$. Let p_i^+ be a definition learned by using, as positive examples, the literals in the training set classified as belonging to p_i and, as negative examples, all the literals for the other classes. Then the following rules ensure consistency on unseen literals and on exceptions:

$$
\begin{array}{lcl}
p_1(\vec{X}) & \leftarrow & p_1^+(\vec{X}), not\ abnormal_{p_1}(\vec{X}), not\ p_2(\vec{X}), \ldots, not\ p_n(\vec{X}) \\
p_2(\vec{X}) & \leftarrow & p_2^+(\vec{X}), not\ abnormal_{p_2}(\vec{X}), not\ p_1(\vec{X}), not\ p_3(\vec{X}), \ldots, not\ p_n(\vec{X}) \\
\cdots & \leftarrow & \cdots \\
p_n(\vec{X}) & \leftarrow & p_n^+(\vec{X}), not\ abnormal_{p_n}(\vec{X}), not\ p_1(\vec{X}), \ldots, not\ p_{n-1}(\vec{X})
\end{array}
$$

$$
\begin{array}{lcl}
p_1(\vec{X}) & \leftarrow & p_1^+(\vec{X}), undefined(p_2^+(\vec{X})), \ldots, undefined(p_n^+(\vec{X})) \\
p_2(\vec{X}) & \leftarrow & p_2^+(\vec{X}), undefined(p_1^+(\vec{X})), undefined(p_3^+(\vec{X})), \ldots, undefined(p_n^+(\vec{X})) \\
\cdots & \leftarrow & \cdots \\
p_n(\vec{X}) & \leftarrow & p_n^+(\vec{X}), undefined(p_1^+(\vec{X})), \ldots, undefined(p_{n-1}^+(\vec{X}))
\end{array}
$$

regardless of the algorithm used for learning the p_i^+.

Noisy Training Set Consider the case in which the training sets are not disjoint. Then, literals in the intersection of the training sets will be abnormal exceptions for both definitions. For an atom $p(\vec{X})$, both $p(\vec{X})$ and $\neg p(\vec{X})$ will result false in the three-valued model of the theory. Therefore, these literals differ from unseen ones, for which the truth value of $p(\vec{X})$ and $\neg p(\vec{X})$ is undefined.

5.5.2 Multiple Source Contradiction

In the single source case above, we showed how to deal with contradictions arising from learning conflicting rules for a predicate and its explicit negation, originating in the same knowledge source. Here we consider and handle contradictions arising from combining rules obtained from distinct knowledge sources or on distinct occasions. Let us dub it *multiple source contradiction.* This kind of situation may occur in the settings of:

- multiple, separately learning agents with distinct background knowledge, or multiple, cloned, agents with the same background knowledge;
- one agent learning separate rules from heterogenous data sources;
- one agent learning rules from uniform but separate data sets, (either because of their size, or in order to benefit from parallelism, or both);
- one agent learning separate sets of rules on different occasions;
- one agent learning separate sets of rules by employing multiple strategies or systems;
- a combination of these settings.

Example 64 *Consider, for instance, the case of two persons living in Los Angeles (say i and j). Both have an interest in identifying attackers (and non attackers) but each of them has had different experiences (i.e., different training sets). Let, for instance, the background knowledge and training set of person i be those reported in the example in section 5.4. Let person j know what is known by person i, but also that youth gangs can attack persons, ever since he visited Brazil:*

$$man(22) \quad \neg good\_appearance(22) \quad age(22, youth) \quad gang\_member(22)$$

Let the training set for j be the same of example 60 plus a new positive example $attacker(22)$. Then, the program clauses induced by person j are as follows (here we consider only the most general solutions learned by GOLEM):

$$\begin{array}{lcl}
attacker^+_{MGS}(X) & \leftarrow & gang_member(X) \\
attacker^+_{MGS}(X) & \leftarrow & \neg good_appearance(X), age(X, adult)
\end{array}$$

$$\begin{array}{lcl}
attacker^-_{MGS}(9) & & \\
attacker^-_{MGS}(10) & & \\
attacker^-_{MGS}(X) & \leftarrow & good_appearance(X) \\
attacker^-_{MGS}(X) & \leftarrow & age(X, old)
\end{array}$$

Then, when these two persons meet one another and exchange experience about their notions of attacker, a contradiction arises because person i classifies the unseen instance 22 as a non attacker whereas person j classifies it as an attacker.

Generalizing the Single Source Technique The single source technique of section 5.5.1 can be easily generalized to multiple sources for learning p and $\neg p$. Let there be s sources for p and $\neg p$. We now have clauses 5.1-5.4 previously introduced, and for i from 1 to s:

$$p^+(\vec{X}) \quad \leftarrow \quad p_i^+(\vec{X}) \tag{5.5}$$

$$p^-(\vec{X}) \quad \leftarrow \quad p_i^-(\vec{X}) \tag{5.6}$$

$$abnorm_p(\vec{X}) \quad \leftarrow \quad abnorm_{p_i^+}(\vec{X}) \tag{5.7}$$

$$abnorm_{\neg p}(\vec{X}) \quad \leftarrow \quad abnorm_{p_j^-}(\vec{X}) \tag{5.8}$$

This means that whenever any two sources conflict on p for $\vec{X}$, both $p(\vec{X})$ and $\neg p(\vec{X})$ become undefined. Also, any abnormality found by one source is, *ipso facto*, an abnormality for them all. Note that some sources may provide information only about positive or negative information, thus the definition for only one of p_i^+ or p_i^- may be available.

Conflicts and Preferences However, a new situation may now arise which could not do so in the single source case: we may prefer one knowledge source over another, e.g., we may trust one source all the more because of its learning method, or because it has more recent or more trustworthy information. In example 64, for instance, the preference might be given to the person which has had in his past life the greatest number of experiences (i.e., known instances and classified instances).

To achieve this, and inspired by the program update method of [ALP+98], we generalize clause 5.5 and 5.6 above to the *combination rules*:

$$\begin{aligned} p^+(\vec{X}) &\leftarrow p_i^+(\vec{X}), not\ reject(p_i^+(\vec{X})) \\ p^-(\vec{X}) &\leftarrow p_i^-(\vec{X}), not\ reject(p_i^-(\vec{X})) \end{aligned}$$

Predicate *reject* expresses when one knowledge source, say i, is rejected by another, say j, with respect to p, through the *reject rules*[3]:

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_j^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_j^+(\vec{X}) \end{aligned}$$

It may as well be the case that the positive and negative information provided by source i are rejected by two different sources k and l.

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_k^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_l^+(\vec{X}) \end{aligned}$$

It can also be the case that only one or even none of these clauses is present for source i, in the case in which no source is preferred to i.

But, naturally, rejection may be made to occur for a variety of reasons, and the bodies of clauses for *reject* will then observe the corresponding conditions.

As for the case of a single source, two or more knowledge sources may reject one another's conflicting conclusions. Instead of treating mutually contradictory information as undefined, as done by means of clauses 5.1-5.4, we can treat mutually contradictory information as false by means of appropriate *reject* rules, both in the single source case and in the multiple source case. Preferring *false* to *undefined* in removing a contradiction amounts to ignoring the clause instances leading to it, so that the usual CWA is adopted symmetrically with respect to positive and negative information [APP98].

Conflicting conclusions of two knowledge sources i and j can be made mutually *false* instead of *undefined* by means of the following instances of *reject* rules:

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_j^-(\vec{X}) \qquad & reject(p_j^+(\vec{X})) &\leftarrow p_i^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_j^+(\vec{X}) \qquad & reject(p_j^-(\vec{X})) &\leftarrow p_i^+(\vec{X}) \end{aligned}$$

[3]If we want rejection to be as strong as what is rejected we may qualify these rules by appealing to the non undefinedness of the rejector.

If symmetry is not desired, one can remove self-contradiction by opting for only some of these clauses.

Let us now consider an example where a knowledge source is preferred over another.

Example 65 *Suppose k is the boss of i, and that they may have distinct, separately learnt, opinions about p. We may combine together their knowledge, by adding*

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_k^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_k^+(\vec{X}) \end{aligned}$$

to ensure that a conclusion arrived at by the boss wins over that of a contrary conclusion by the subordinate.

For the case of a colleague j of i, we may choose to eliminate all mutual contradictions, by means of:

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_j^-(\vec{X}) & \qquad reject(p_j^+(\vec{X})) &\leftarrow p_i^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_j^+(\vec{X}) & \qquad reject(p_j^-(\vec{X})) &\leftarrow p_i^+(\vec{X}) \end{aligned}$$

Notice that, when learning, an agent as access only to its background knowledge but, when the knowledge is combined, it may access as well the definitions of background or target predicates of other agents. In some cases it may happen that a contradiction arises exactly because, after the combination of the learned rules, an agent may use the knowledge learned by another agent as background knowledge.

Example 66 *Suppose agent i has non-contradictorily learned from examples that*

$$\begin{aligned} p_i^+(\vec{X}) &\leftarrow a(\vec{X}) \\ p_i^-(\vec{X}) &\leftarrow b(\vec{X}) \end{aligned}$$

Recall that, before knowledge sources are combined, only access to self knowledge is possible.

Further, suppose next that j has learned the rules

$$\begin{aligned} a_j(\vec{X}) &\leftarrow \neg c(\vec{X}) \\ b_j(\vec{X}) &\leftarrow \neg c(\vec{X}) \end{aligned}$$

and that the background acknowledges the fact

$$\neg c(golem)$$

When the rules from i and j are combined, i and j may access each conclusion and the background knowledge too. Now a contradiction arises in the knowledge of i regarding $p_i^+(golem)$ and $p_i^-(golem)$. If we want to resolve this contradiction by preferring false over undefined, we can use the following reject rules

$$\begin{aligned} reject(p_i^+(\vec{X})) &\leftarrow p_i^-(\vec{X}) \\ reject(p_i^-(\vec{X})) &\leftarrow p_i^+(\vec{X}) \end{aligned}$$

5.6 Strategies for Theory Refinement

When learning a definition for a concept p and its opposite $\neg p$ (separately or not), it can be the case that some contradiction arises for an *unseen* literal. Figure 5.4 depicts various cases which may occur. Identifying such contradictions is useful in interactive theory revision, where the system can ask an oracle to classify the literal(s) leading to contradiction, and accordingly revise the least or most general solutions for p and for $\neg p$. Detecting uncovered literals points to theory extension.

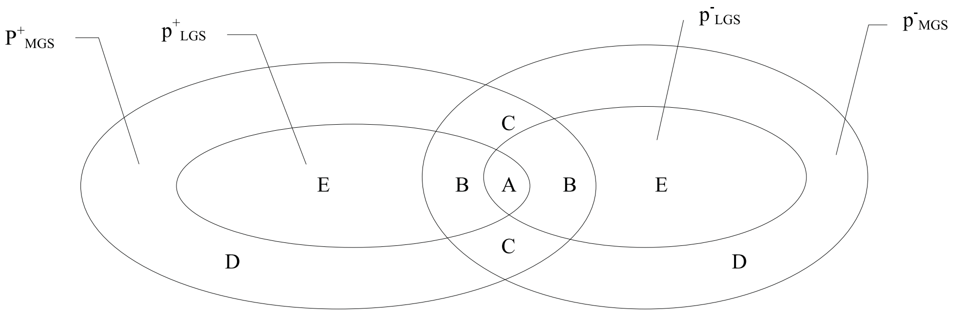

Figure 5.4: Intersection of Learnt Solutions

Refinement Further information on unseen contradictory literals for the various cases can help in improving learnt rules.

Area A represents contradictions between the two least general solutions, for a concept p and its opposite $\neg p$, i.e., it represents unseen literals satisfying the conjunction $p^+_{LGS}(\vec{X}), p^-_{LGS}(\vec{X})$. This is the strongest form of contradiction, and unseen literals in region A should be given priority when querying the oracle.

Example 67 *Consider, for instance, examples 59-60, and the unseen literal* 22 *which belongs to the intersection of the learned definitions for* $attacker^+_{LGS}$ *and* $attacker^-_{LGS}$ *(i.e., area A in figure 5.4). Knowing that* 22 *is an attacker helps in specializing the learned definitions, in this case* $attacker^+_{LGS}$ *and* $attacker^+_{MGS}$.

Areas B represent contradictions between most general solutions for concept p^+ and p^- which are outside the least general solution for one concept, but inside the least general solution for the other. I.e., they represent unseen literals satisfying the conjunction $p^+_{MGS}(\vec{X}), not\ p^+_{LGS}(\vec{X}), p^-_{LGS}(\vec{X})$ or the conjunction $p^-_{MGS}(\vec{X}), not\ p^-_{LGS}(\vec{X}), p^+_{LGS}(\vec{X})$. Identifying such contradictions can be useful in refining knowledge and, in particular, the most and least general solutions for a concept. For literals satisfying the first conjunction, the system has to revise most general solution for p^+ if the oracle classifies the literal as negative and the least and most general solution for p^- if the oracle classifies the literal as positive, and vice-versa for the literals satisfying the second conjunction.

Though less strongly contradictory than area A, areas B are more strongly so than areas C, and so merit attention next when querying the oracle.

algorithm LIVE(
  inputs :E^+, E^-: training sets,
    B: background theory,
  outputs : H : learned theory)
LearnHierarchy($E^+, E^-, B; H_p$)
LearnHierarchy($E^-, E^+, B; H_{\neg p}$)
Obtain H by:
  transforming H_p, $H_{\neg p}$ into "non-deterministic" rules
  adding the clauses for the undefined case
output H

Figure 5.5: Algorithm LIVE

Areas C represent contradictions between most general solutions for concept p and its opposite which are outside both the least general solutions. I.e., it represents literals satisfying the conjunction $p^+_{MGS}(\vec{X}), not\ p^+_{LGS}(\vec{X}), p^-_{MGS}(\vec{X}), not\ p^-_{LGS}(\vec{X})$. Identifying such contradictions can be useful in refining knowledge and bridging the gap between most and least general solutions for a concept. The system has to revise the most general solution for p if the oracle classifies the atom as negative and for $\neg p$ if the oracle classifies the atom as positive, and vice-versa.

Finally, it is worth mentioning that other regions where a contradiction does not arise, namely D and E, can be useful in guiding knowledge acquisition. New information about an unseen atom always increases knowledge, and thus eventually requires knowledge refinement or knowledge extension. However, among unseen literals not leading to contradiction, we can identify class D which can be more useful than E in bridging the gap between the least and the most general solution. This area represents instances which satisfy the conjunction $p^+_{MGS}(\vec{X}), not\ p^+_{LGS}(\vec{X}), not\ p^-_{MGS}(\vec{X})$ or $p^-_{MGS}(\vec{X}), not\ p^-_{LGS}(\vec{X}), not\ p^+_{MGS}(\vec{X})$. If a literal satisfying the former condition is classified as negative by a oracle, then the most general solution for p has to be revised, whereas, if a literal satisfying the latter condition is classified as positive by an oracle, then the most general solution for $\neg p$ has to be revised.

It may be that learnt rules do not cover atoms and their negations for legitimate argument tuples. Accordingly, a further area exists (the one outside the areas in figure 5.4) which pinpoints cases of interest, leading to theory extension (and subsequent refinement where contradictions emerge).

5.7 An Algorithm for Learning Extended Logic Programs

The algorithm LIVE (Learning In a 3-Valued Environment) learns extended logic programs containing non-deterministic rules for a concept and its opposite that may allow a hierarchy of exceptions.

Figure 5.5 shows the main procedure of the algorithm. It calls a procedure LearnHierarchy (see figure 5.6) that, given a set of positive, a set of negative examples and a background knowledge, returns a definition for the positive concept, consisting of default rules, together

procedure LearnHierarchy(
inputs : E^+: positive examples,
E^-: negative examples, B: background theory,
outputs : H: learned theory)
Learn$(E^+, E^-, B; H_p)$
$H := H_p$
for each rule r in H_p **do**
Find the sets E_r^+, E_r^- of positive and negative examples covered by r
if E_r^- is not empty **then**
Add the literal $not\_abnormal_r(\vec{X})$ to r
Obtain $E^+_{abnormal_r}$, $E^-_{abnormal_r}$ from E_r^-, E_r^+ by
transforming each $p(\vec{t})$ into $abnormal_r(\vec{t})$
LearnHierarchy$(E^+_{abnormal_r}, E^-_{abnormal_r}, B; H_r)$
$H := H \cup H_r$
endif
enfor
output H

Figure 5.6: Procedure LearnHierarchy

with definitions for the eventual abnormality literals. The procedure LearnHierarchy is called twice, once for the positive concept and once for the negative concept. When it is called for the negative concept, E^- is used as the positive training set and E^+ as the negative one.

LearnHierarchy first calls a procedure Learn$(E^+, E^-, B; H_p)$ that learns a definition H_p for the target concept p. Learn consists of an ordinary ILP algorithm, either bottom-up or top-down, modified to adopt the *SLX* interpreter for testing the coverage of examples and to relax the consistency requirement of the solution. The procedure thus returns a theory that may cover some negative examples. These negative examples are then treated as exceptions, by adding a default literal to the inconsistent rules and learning a definition for the abnormality predicate. In particular, for each rule $r = p(\vec{X}) \leftarrow Body(\vec{X})$ in H_p covering some negative examples, a new non-abnormality literal $not\ abnormal_r(\vec{X})$ is added to r and a definition for $abnormal_r(\vec{X})$ is learned by recursively calling LearnHierarchy. Examples for $abnormal_r$ are obtained from examples for p by observing that, in order to cover a positive (uncover a negative) example $p(\vec{X})$ for p, the atom $abnormal_r(\vec{X})$ must be false (true). Therefore, positive (negative) examples for $abnormal_r$ are obtained from the set E_r^- of negative (E_r^+ of positive) examples covered by the rule. When learning a definition for $abnormal_r$, in turn, LearnHierarchy may find exceptions to exceptions and call itself recursively again. In this way, we are able to learn a hierarchy of exceptions.

Let us now discuss in more details the algorithm that implements the Learn procedure. Depending on the generality of solution that we want to learn, different algorithms must be employed: a top-down algorithm for learning the MGS, a bottom-up algorithm for the LGS. In both cases, the algorithm must be such that, if a consistent solution cannot be found, it returns a theory that covers the least number of negative examples.

When learning with a top-down algorithm, the consistency necessity stopping criterion must be relaxed to allows clauses that are inconsistent with a small number of negative examples to be learned, for example by adopting one of the heuristic necessity stopping criteria proposed in ILP to handle noise, such as the encoding length restriction [Qui90b] of FOIL (see section 3.4.2) or the significancy test of mFOIL [Dže91] (see section 3.4.3). In this way, we are able to learn definitions of concepts with exceptions: when a clause must be specialized too much in order to make it consistent, we prefer to transform it into a default rule and consider the covered negative examples as exceptions.

The simplest criterion that can be adopted is to stop specializing the clause when no literal from the language bias can be added that reduces the coverage of negative examples.

When learning with a bottom-up algorithm, we can learn using positive examples only by using the $rlgg$ operator: since the clause is not tested on negative examples, it may cover some of them. This approach is realized by using the system GOLEM (see section 3.4.1), as in [IK97].

In order to show the behaviour of the algorithm when learning exceptions and to compare it with those of the system LELP [IK97], we will consider the learning problem that is described in example 3.4 in [IK97] where the definition of the concept $flies$ is learned.

Example 68 *Consider the following background knowledge and training sets:*

$penguin(1)$ $penguin(2)$
$bird(3)$ $bird(4)$ $bird(5)$
$animal(6)$ $animal(7)$ $animal(8)$
$animal(9)$ $animal(10)$ $animal(11)$
$animal(12)$
$animal(X) \leftarrow bird(X)$
$bird(X) \leftarrow penguin(X)$

$E^+ = \{flies(3), flies(4), flies(5)\}$
$E^- = \{flies(1), flies(2), flies(6), flies(7), flies(8), flies(9), flies(10), flies(11), flies(12)\}$

We consider the case in which a top-down method is adopted for the procedure Learn. The stopping criterion used is the simplest, i.e., we stop when no literal can be added to reduce the number of covered negative examples (suppose that the language bias allows any literal built on predicates of the background knowledge to appear in the body of clauses). The algorithm learns the rules

(1) $flies^+(X) \leftarrow bird(X), not\ abnormal_1(X)$
(2) $abnormal_1(X) \leftarrow penguin(X)$
(3) $flies^-(X) \leftarrow animal(X), not\ abnormal_3(X)$
(4) $abnormal_3(X) \leftarrow bird(X), not\ abnormal_4(X)$
(5) $abnormal_4(X) \leftarrow penguin(X)$

Then, the algorithm builds the clauses for $flies$ *and* $\neg flies$ *and make them non-deterministic, adds the clauses for the undefined case and terminates.*

5.8 Implementation

In order to learn the most general solutions, a top-down ILP algorithm (cf. section 3.2.6) has been integrated with the procedure *SLX* for testing the coverage. The specialization loop of the top-down system consists of a beam search in the space of possible clauses. At each step of the loop, the system removes the best clause from the beam and generates all its refinement. They are then evaluated according to an accuracy heuristic function, and their refinements covering at least one positive example are added to the beam. The best clause found so far is also separately stored: this clause is compared with each refinement and is replaced if the refinement is better. The specialization loop stops when either the best clause in the beam is consistent or the beam becomes empty. Then, the system returns the best clause found so far. The beam may become empty before a consistent clause is found and in this case the system will return an inconsistent clause.

In order to find least general solutions, the GOLEM ([MF90], also described in section 3.4.1) system is employed. The finite well-founded model is computed, through *SLX*, and it is transformed by replacing literals of the form $\neg A$ with new predicate symbols of the form *neg_A*. Then GOLEM is called with the computed model as background knowledge. The output of GOLEM is then parse in order to extract the clauses generated by *rlgg* before they are post-processed by dropping literals. Thus, the clauses that are extracted belong to the least general solution. In fact, they are obtained by randomly picking couples of examples, computing their *rlgg* and choosing the consistent one that covers the bigger number of positive examples. This clause is further generalized by choosing randomly new positive examples and computing the *rlgg* of the previously generated clause and each of the examples. The consistent generalization that covers more examples is chosen and further generalized until the clause starts covering some negative examples. An inverse model transformation is then applied to the rules thus obtained by substituting each literal of the form *neg_A* with the literal $\neg A$.

Prolog was chosen for the implementation of LIVE for the same reasons for which it was chosen for ACL1 that are mentioned in section 4.4.1: Prolog is particularly suitable for the elaboration of logic programs due to the uniformity of code and data, to meta-level predicates for accessing programs and to the availability of lists as primitive data structures.

LIVE code is composed of the following main procedures. As for ACL1, `i(File)` is the command to be given at the Prolog prompt for starting the induction. It reads the files that contains the input data, it calls the procedure `learn_ELP(Rules)` and writes the output to a file.

`learn_ELP(Rules)` implements the main procedure of LIVE (see figure 5.5) and calls twice the procedure `learn_hierarchy(Eplus,Eminus,Rules,Gen)` for learning the positive and the negative concept. The argument `Gen` can assume the values `lgs` or `mgs` and is used in order to indicate the generality of the solutions specified by the user.

The procedure `learn(Eplus,Eminus,Rules,Gen)` is called by `learn_hierarchy` and, depending on the value of `Gen`, either calls a procedure `call_golem(Eplus,Eminus,Rules)`, that invokes GOLEM, or calls the procedure `covering_loop(Eplus,Eminus,[],Rules)` that implements the top-down algorithm.

`covering_loop(Eplus,Eminus,RulesIn,RulesOut)` first initializes the beam by including in it a clause with an empty body for every target predicate and then starts the specialization loop by calling `specialize(BeamIn,BeamOut,Eplus,Eminus,N)`. The parameter `N` is used in order to put a limit on the maximum number of specialization steps.

The predicate `evaluate(Value,Clause,Eplus,Epluscovered,Eminus,Eminuscovered,` `Nplus,Nminus)` is used in order to evaluate clauses. It takes as input the clause to be evaluated `Clause` and the current training set `Eplus, Eminus`, and returns the values of the heuristic function `Value` together with the sets of covered examples `Epluscovered,` `Eminuscovered`.

LIVE was implemented in *XSB* Prolog [SSW$^+$97] and the code of the system can be found at the following address:
`http://www-lia.deis.unibo.it/Software/LIVE/`.

5.9 Related Work

The adoption of a three-valued logic in learning has been investigated by many authors. Many propositional learning systems learn a definition for both the concept and its opposite. For example, systems that learn decision trees, as c4.5 [Qui93], or decision rules, as the AQ family of systems [Mic73], are able to solve the problem of learning a definition for n classes, that generalizes the problem of learning a concept and its opposite. However, in most cases the definitions learned are assumed to cover the whole universe of discourse: no undefined classification is produced, any instance is always classified as belonging to one of the classes. Instead, we classify as undefined the instances for which the learned definitions do not give an unanimous response.

When learning multiple concepts, it may be the case that the descriptions learned are overlapping. We have considered this case as non-desirable: this is reasonable when learning a concept and its opposite but it may not be the case when learning more than two concepts. As it has been pointed out by [Mic84], in some cases, it is useful to produce more than one classification for an instance: for example if a patient has two diseases, his symptoms should satisfy the descriptions of both diseases. Subject for future work will be to consider classes of paraconsistent logic programs where the overlapping of definitions for p and $\neg p$ (and, in general, multiple concepts) is allowed.

The problems raised by negation and uncertainty in concept-learning, and Inductive Logic Programming in particular, were pointed out in some previous work (e.g., [BM92, DRB90, DR92]). For concept learning, the use of the CWA for target predicates is no longer acceptable because it does not allow to distinguish between what is false and what is undefined. De Raedt and Bruynooghe [DRB90] proposed to use a three-valued logic (later on formally defined in [DR92]) and an explicit definition of the negated concept in concept learning. This technique has been integrated within the CLINT system, an interactive concept-learner. In the resulting system, both a positive and a negative definition are learned for a concept (predicate) p, stating, respectively, the conditions under which p is true and those under which it is false. The definitions are learned so that they do not produce an inconsistency on the examples. Differently from this system, we take also care that the two definitions do not produce inconsistency on unseen atoms and we are able to learn definitions for exceptions to both concepts. Furthermore, we are able to cope with two kinds of negation, the explicit one used to state what is false, and the default (defeasible) one used to state what can be assumed false.

The system LELP (Learning Extended Logic Programs) [IK97] learns extended logic programs under answer-set semantics. As our algorithm, LELP is able to learn non-deterministic default rules with a hierarchy of exceptions. From the point of view of the learning problems that the two algorithms can solve, they are equivalent when the background is a stratified extended logic program. All the examples shown in [IK97] are stratified and therefore they can be learned by our algorithm and, viceversa, example in section 5.5.1 can be learned by LELP. However, when the background is a non-stratified extended logic program, the adoption of a well-founded semantics gives a number of advantages with respect to the answer-set semantics. For non-stratified background theories, answer-sets semantics does not enjoy the structural property of relevance [Dix95], like our *WFSX* does, and so they cannot employ any top-down proof procedure. Furthermore, answer-set semantics is not cumulative [Dix95], i.e., if you add a lemma then the semantics can change, and thus the improvement in efficiency given by tabling cannot be obtained. Moreover, by means of *WFSX*, we have introduced a method to choose one concept when the other is undefined which they cannot replicate because in the answer-set semantics one has to compute eventually all answer-sets to find out if a literal is undefined.

The structure of the two algorithms is similar: LELP first generates candidate rules from a concept using an ordinary ILP framework. Then exceptions are identified (as covered examples of the opposite set) and rules specialized through negation as default and abnormality literals, which are then assumed to prevent the coverage of exceptions. These assumptions can be, in their turn, generalized to generate hierarchical default rules.

One of the differences between us and [IK97] is in the level of generality of the definitions we can learn. LELP learn a definition for a concept only from positive examples of that concept and therefore it can only employ a bottom-up ILP technique and learns the LGS. Instead, we can choose whether to adopt a bottom-up or a top-down algorithm and we can learn theories of different generality for different target concepts.

Another difference consists in the fact that LELP learns a definition only for the concept that has the highest number of examples in the training set. It learns both positive and negative concepts only when the number of positive examples is close to that of negative ones (in 60 %-40 % range), while we always learn both concepts.

LELP also differs from our approach because it adds to the theory a clause for the negative concept given in terms of the abnormality literals for the positive concept. For example, in the case of example 63, LELP would produce the following theory:

$C_1 = flies^+(X) \leftarrow has\_wings(X), not\ abnormal_1(X)$
$C_2 = abnormal_1(X) \leftarrow penguin(X)$
$C_3 = flies^-(X) \leftarrow has\_limbs(X)$
$C_4 = flies^-(X) \leftarrow abnormal_1(X)$

We do not generate clause C_4 since, when learning a definition for both $flies$ and $\neg flies$, the examples it covers are already covered by clause C_3 and therefore such a clause is redundant.

Several other authors have also addressed the task of learning rules with exceptions [DK95]. In these frameworks, non monotonicity and exceptions are dealt with by learning logic programs with negation. In [DK95] the authors rely on a language which uses a limited form of "classical" (or, better, syntactic) negation together with a priority relation among the sentences of the program [KMD94]. The expressive power of this formalism is however more restricted than the one of extended logic programs since, theories expressed in this language can be mapped into normal logic program.

Non-abnormality literals can also be viewed as new abducible predicates, as done for instance in [LMMR97, EFL+98, Ino98]. In particular, in [LMMR97, EFL+98] the authors have considered the integration and cooperation of induction and abduction in order to learn Abductive Logic Programs (ALP) from (possibly) incomplete background knowledge expressed as ALP in its turn. In order to make a rule for a target predicate p consistent, the rule is specialized by adding a new abducible literal $not\_abnorm_i(\vec{X})$ and exceptions are ruled out by abducing $abnorm_i(\vec{X})$ for them. These assumptions are then used to learn a definition for $abnorm_i$ that describes the class of exceptions. In this way, they are able to learn hierarchies of exceptions. Since there exists an implementation of SLX with abduction (called SLXA [AP98]) this points to future extensions of LIVE with abduction too.

5.10 Conclusions

The two-valued setting that has been considered in most work on ILP and Inductive Concept Learning in general is not sufficient in many cases where we need to represent real world data. This is for example the case of an agent that has to learn the effect of the actions it can perform on the domain by performing experiments. Such an agent needs to learn a definition for allowed actions, forbidden actions and actions with an unknown outcome and therefore it needs to learn in a richer three-valued setting.

In order to adopt such a setting in ILP, the class of extended logic programs under the well-founded semantics with explicit negation ($WFSX$) is adopted as the representation language. This language allows two kinds of negation, default plus a second form of negation called explicit, that is used in order to represent explicitly negative information. Adopting extended logic programs in ILP prosecutes the general trend in Machine Learning of extending the representation language in order to overcome the limits of existing systems.

The programs that are learned will contain a definition for the concept and its opposite, where the opposite concept is expressed by means of explicit negation. When learning in a three-valued settings, a number of issues have to be taken into account. Standard ILP techniques can be adopted to separately learn the definitions for the concept and its opposite. Depending on the adopted technique, one can learn the most general or the least general definition. Accordingly, four epistemological varieties occur, resulting from the mutual combination of most general and least general solutions for the positive and negative concept. The choice of one of these epistemological variety should be done on the basis of a number of conditions that hold in the learning situation, such as the damage that can derive from an erroneous classification of an unseen object or the confidence we have in the training set.

The two definition learned may overlap and the inconsistency is resolved in a different way for atoms in the training set and for unseen atoms: atoms in the training set are considered as exceptions, while unseen atoms are considered as unknown. The different behaviour is obtained by employing negation by default in the definitions: default abnormality literals are used in order to consider exceptions to rules, while non-deterministic rules are used in order to obtain an unknown value for unseen atoms. Exceptions to a positive concept are identified from negative examples, whereas exceptions to a negative concept are identified from positive examples. A definition for the class of exceptions may then be learned and may include new exceptions. The process is then iterated thus possibly producing a hierarchy of exceptions. This way of coping with contradiction can be generalized for multiple source learning, and modified in order to take into account preferences among multiple learning agents or information sources. Moreover, we discuss how detecting different kinds of uncovered atoms points to different opportunities for theory extension.

The system LIVE (Learning in a three-Valued Environment) has been developed that implements the above mentioned techniques. In particular, the system learns a definition for both the concept and its opposite and is able to identify exceptions and to learn a hierarchical definition for them. The system is parametric in the procedure used for learning each definition: it can adopt either a top-down algorithm, using beam-search and heuristic necessity stopping criterion, or a bottom-up algorithm, that exploits the GOLEM system.

Chapter 6

Conclusions

The aim of this thesis was to demonstrate how some of the limits of existing learning techniques in ILP can be overcome by adopting extensions of Logic Programming. Increasing the expressiveness of the representation language in Machine Learning is a general trend that has allowed to solve more and more complex learning problems. The language used to represent concepts and examples has gone from analytical expressions to attribute-value formalisms and finally to first order logic languages and Logic Programming in particular. Adopting extensions of Logic Programming is thus a natural prosecution of this trend.

Two problems where current ILP systems perform poorly are presented. The first problem consists in learning from an incomplete background knowledge. To this purpose, abductive logic programs are used that allow to perform hypothetical reasoning from incomplete knowledge. A new learning problem is defined where both the background and target theories are abductive theories and abductive entailment is used as the coverage relation.

The system ACL (Abductive Concept Learning) has been developed that is able to learn in this new framework. An abductive theory is learned by first learning the program part, by means of a top-down algorithm (called ACL1) adopting an abductive proof procedure for testing the coverage, and then learning the constraint part by employing the system ICL. Experiments have been performed in a variety of domains where the knowledge is incomplete. The results have been compared with those of the systems ICL-Sat, mFOIL and FOIL that adopt special techniques for handling imperfect data. In the multiplexer experiment, the accuracy of the theory learned by ACL1 has been superior to the one of theories learned by ICL-Sat and mFOIL. In the father experiment, ACL1 without constraints learned a complete and consistent theory for all levels of incompleteness apart from 80% and 40%, while mFOIL learned a complete theory for all levels of incompleteness but a consistent one only for the case of no incompleteness. When considering as well integrity constraints in the background, ACL1 learned a complete and consistent theory for all levels of incompleteness. An experiment has been performed as well on real world data from the domain of marketing where the incompleteness occours naturally as unanswered questions and the theory learned by ACL1 has been judged to be very meaningful by experts. For all three experiments, the constraint learning phase was also performed, obtaining in all cases constraints that could be useful for classifying incompletely specified unseen atoms.

The framework of learning abductive logic programs can be very useful as well for learning multiple predicates. A system for learning multiple predicates called M-ACL has been implemented that is able to solve some of the problems of ILP systems when learning multiple predicates. The system was able to learn programs containing the definitions of multiple predicates such as a very simple definite clause grammar for the English language, the mutually recursive predicates *even* and *odd* and multiple family relations.

The other problem that has been considered consists in learning in a three-valued logical setting. Most work on inductive concept learning has considered a two-valued setting, however this is not sufficient in many learning situations, such as the one of an autonomous agent that has to learn general rules about the outcome of its actions on the surrounding world. In this case, the agent wants to learn when an action has a positive outcome, when it has a negative outcome and distinguish them from actions with an unknown outcome. To this purpose, the class of extended logic programs under the well founded semantics with explicit negation *WFSX* ([AP96]) is used as the representation language. The language allows two forms of negation, default negation plus explicit negation that is used in order to explicitly represent negative information, and the semantics allows three logical values for atoms. The programs that are learned will contain a definition for the concept and its opposite.

The system LIVE (Learning In a three-Valued Environment) has been developed that is able to learn extended logic programs containing a definition for the concept and its opposite. The system takes into account a number of issues that arise when learning in a three-valued settings. Contradiction among the definitions for the concept and its opposite may arise. The contradiction is resolved differently depending on whether the atom on which there is contradiction is an unseen atom or belongs to the training set. In the first case, the contradiction is resolved by assigning the unknown truth value to the atom, in the second case by assigning the truth value given by the training set. Contradiction is handled by employing representation techniques offered by extended logic programs. Similar techniques can also be used in order to handle contradiction among different sources of information. The system is parametric in the learning technique adopted for learning the concept and its opposite: if a bottom-up technique is used, then a least general solution is found, if a top-down technique is used, then a most general solution is found. Various criteria have been studied for choosing between the least general solution or the most general solution for the concepts. The theory learned by LIVE may allow exceptions. A definition for exceptions is then learned that, on its turn, may also allow exceptions. In this way hierarchies of exceptions can be learned.

LIVE is compared with the system LELP that is also able to learn extended logic programs containing a definition for the concept and its opposite that allow exceptions. Differently from LIVE, LELP adopts a two-valued semantics and thus is not able to classify as unknown unseen atoms in the intersection of definitions. Moreover, LELP is not parametric in the learning technique employed, thus it can not learn solutions of different generality.

The two systems proposed have been tested on a number of artificial datasets in various domains. ACL was also tested on real world data in the marketing domain. Further testing on real world data is needed in order to provide more evidence of the effectiveness of the techniques in practice. In particular, the system ACL will be applied to perform Data Mining tasks in domains where incompleteness occours naturally in the data, as in the marketing domain. LIVE instead will be applied to perform knowledge acquisition by agents that have to automatically explore the surrounding world. Such agents could be, for example, robots sent to unknown environments to perform a specific task.

The study of the two proposed extension of Logic Programming is a first step towards the development of a system that is able to learn from incomplete information in a three-valued settings.

In the research field of Logic Programming an extension of the language has been studied that considers abductive theories containing two kinds of negation, default and explicit. A semantics for this class of programs was given in [BLMM97] and a proof procedure for it was given in [AP98]. The learning techniques adopted in the two proposed systems can be combined for obtaining a system that learns using this extended class of programs as the representation language. Such a system would be able to learn definitions for both a concept and its opposite starting from an incomplete background knowledge.

Appendix A

Appendixes to Chapter 4

A.1 Proof of Theorem 44 on Equivalence of ACL with ACL1 and ACL2

Theorem 17 *Let $T_{ACL1} = \langle P \cup P', A, I\rangle$, Δ^+ and Δ^- be the solution of ACL1 given training sets E^+ and E^-, background theory $T = \langle P, A, I\rangle$ and space of possible programs $\mathcal{P}$. Moreover, let $T' = \langle P \cup P', A, I \cup I'\rangle$ be the solution to ACL2 given the previous solution of ACL1 and space of possible constraints $\mathcal{I}$. Then T' is a solution to the ACL problem that has E^+ and E^- as training sets, T as background theory and $\mathcal{P}$ and $\mathcal{I}$ as spaces of possible programs and constraints.*

Proof: We first prove that $T' \models_A E^+$ and then that $\forall e^- \in E^-$, $T' \not\models_A e^-$.
Proof of $T' \models_A E^+$: from ACL1 we have that $M_{P\cup P'}(\Delta^+) \models E^+$. From ACL1 and ACL2 we have, respectively, that $M_{P\cup P'}(\Delta^+) \models I$ and $M_{P\cup P'}(\Delta^+) \models I'$, therefore $M_{P\cup P'}(\Delta^+) \models I \cup I'$. This, together with $M_{P\cup P'}(\Delta^+) \models E^+$, proves that Δ^+ is an abductive explanation for E^+ in T'.
Proof of $\forall e^- \in E^-$, $T' \not\models_A e^-$: from ACL1 we have that $T_{ACL1} \models_A not\_E^-$ with Δ^-. From the definition of strong abductive explanation of a conjunction of goals (definition 35) Δ^- is also a strong abductive explanation for not_e^- for every $e^- \in E^-$. Therefore, from property 36 in section 4.2 we have

$$\forall \Delta_{e^-} : T_{ACL1} \models_A e^- \text{ with } \Delta_{e^-}, \quad \exists \bar{l} \in \Delta_{e^-} : l \in \Delta^-$$

Since the integrity constraints in T' are a superset of those in T_{ACL1} and the rule part is the same, the set of explanations for e^- in T' is a subset of those for e^- in T_{ACL1}.

The constraints I' generated by ACL2 make inconsistent each of the complements in Δ^- and hence for every such Δ_{e^-} there exists an $\bar{l} \in \Delta_{e^-}$ such that $\{\bar{l}\}$ is inconsistent with I'. From the restricted form of the integrity constraints in I', any superset of $\{\bar{l}\}$, in particular Δ_{e^-}, cannot satisfy the integrity constraints. Therefore, any Δ_{e^-} is not a consistent extension of T' and hence $T' \not\models_A e^-$ as required.

□

A.2 Proof of Theorem 48 on Soundness of ACL

Let us first give the proof of proposition 38 that will be needed for proving theorem 48.

Proposition 11 *Let $T = \langle P, A, I \rangle$ be an abductive theory in its three-valued version and let Δ_1 and Δ_2 be two strong abductive explanations of, respectively, G_1 and G_2, where G_1 and G_2 can be either positive or negative goals. If $\Delta_1 \cup \Delta_2$ is self-consistent, then $\Delta_1 \cup \Delta_2$ is a strong abductive explanation for both G_1 and G_2.*

Proof: We first consider the case where G_1 and G_2 are two positive goals. We need to verify the two conditions of definition 34.

Let us first prove that $M(\Delta_1 \cup \Delta_2)$ is a generalized model. Consider Δ_2 as a self-consistent extension of Δ_1. Since Δ_1 is a strong abductive explanation, any self-consistent extension Δ' of Δ_1 for which $M(\Delta') \models I$, is such that $M(\Delta_1 \cup \Delta') \models I$. Taking $\Delta' = \Delta_2$, since Δ_2 is an abductive explanation, $M(\Delta_2) \models I$ holds and so $M(\Delta_1 \cup \Delta_2) \models I$. Therefore $M(\Delta_1 \cup \Delta_2)$ is a generalized model. Since $P \cup \Delta_1 \cup \Delta_2$ is a definite logic program, $M(\Delta_1 \cup \Delta_2) \supseteq M(\Delta_1)$ and $M(\Delta_1 \cup \Delta_2) \supseteq M(\Delta_2)$ therefore $M(\Delta_1 \cup \Delta_2) \models G_1$ and $M(\Delta_1 \cup \Delta_2) \models G_2$, so $\Delta_1 \cup \Delta_2$ is an abductive explanation for both G_1 and G_2.

To show that $\Delta_1 \cup \Delta_2$ satisfies the second condition of definition 34 consider a set Δ' such that $\Delta' \cup \Delta_1 \cup \Delta_2$ is self-consistent and $M(\Delta') \models I$. We need to prove that $M(\Delta' \cup \Delta_1 \cup \Delta_2) \models I$. Consider the set $\Delta'' = \Delta' \cup \Delta_2$. Since Δ_1 is strong and $\Delta'' \cup \Delta_1$ is self-consistent, if $M(\Delta'') \models I$ then $M(\Delta_1 \cup \Delta'') \models I$ would follow. But $M(\Delta'') \models I$ is true since Δ_2 is strong, $\Delta' \cup \Delta_2$ is self-consistent and $M(\Delta') \models I$.

Consider now the case where we have two negative goals $G_1 = not\_O_1$ and $G_2 = not\_O_2$. In order for $\Delta_1 \cup \Delta_2$ to be a strong abductive explanation for G_1 and G_2, we need to show that the conditions of definition 35 are satisfied. The fact that $M(\Delta_1 \cup \Delta_2)$ is a strong generalized model can be proved in the same way as for positive goals. To show that $M(\Delta_1 \cup \Delta_2) \not\models O_1$ (and similarly $M(\Delta_1 \cup \Delta_2) \not\models O_2$) we note that Δ_1 is a strong abductive explanation for not_O_1 and hence if Δ_2 were an abductive explanation for O_1, then $\Delta_1 \cup \Delta_2$ would not be self-consistent which contradicts the hypothesis of the statement. Next we show the second condition of definition 35, i.e., that for every Δ' that is an abductive explanation for O_1 (or O_2), then $(\Delta_1 \cup \Delta_2) \cup \Delta'$ is not self-consistent. This follows directly from the fact that if Δ' is an explanation for O_1 (O_2), since Δ_1 (Δ_2) is strong, then $\Delta' \cup \Delta_1$ ($\Delta' \cup \Delta_2$) is not self-consistent and hence $\Delta_1 \cup \Delta_2 \cup \Delta'$ is not self-consistent. The other case where one of the goals is positive and the other is negative can be shown similarly. □

Theorem 22 (Soundness) *The ACL algorithm is sound.*

Proof: ACL finds a solution T' of ACL by solving the ACL1 and ACL2 subproblems in sequence. Theorem 44 states that the combination of the solutions of ACL1 and ACL2 gives a solution for ACL. Therefore, to prove the soundness of ACL, it is sufficient to prove that the solutions found by the algorithms for ACL1 and ACL2 satisfy their respective subproblem definitions.

For the second phase of ACL2, this is guaranteed by the correctness of the ICL [DRL95] algorithm or of any other sound method used for discriminating between positive and negative interpretations. It remains therefore to prove that the procedure ACL1 is sound with respect to the ACL1 definition, i.e. that, given the background theory $T = \langle P, A, I \rangle$ and training sets E^+ and E^-, the program $T_{ACL1} = \langle P \cup P', A, I \rangle$ and the sets Δ^+ and Δ^- that are generated by the algorithm are such that

$$T_{ACL1} \models_A E^+ \text{ with } \Delta^+ \tag{A.1}$$

$$T_{ACL1} \models_A not\_E^- \text{ with } \Delta^- \tag{A.2}$$

$$\Delta^+ \cup \Delta^- \text{ is self-consistent} \tag{A.3}$$

ACL1 learns the program T_{ACL1} by iteratively adding a new clause to the current hypothesis, initially empty. Each clause is tested by trying an abductive derivation for each positive and for each (negated) negative example.

Suppose that clauses are learned in the following order: $C_1, \ldots, C_l$. Let $H_1, \ldots, H_l$ be the successive partial hypotheses, with $H_0 = \emptyset$ and $H_k = H_{k-1} \cup \{C_k\}$, and let $T_k = \langle P \cup H_k, A, I \rangle$. Let also $E_k^+ = \{e_{k,1}^+, \ldots, e_{k,n_k}^+\}$ be the set of positive examples whose conjunction is covered by clause C_k and let $E^+ = \{e_1^+, \ldots, e_n^+\}$, $E^- = \{e_1^-, \ldots, e_m^-\}$ be the complete sets of positive and negative examples.

For each clause C_k, we define two sets of abductive assumptions Δ_k^{in} and Δ_k^{out}. Δ_k^{in} is the initial set of assumptions under which the testing of examples with this clause starts. Δ_k^{out} is the final set of assumptions produced in the derivations of all the examples in E_k^+ and in E^-. The input sets Δ_k^{in} are defined recursively via $\Delta_k^{in} = \Delta_{k-1}^{in} \cup \Delta_{k-1}^{out}$ for $k = 2, \ldots, l$, with $\Delta_1^{in} = \emptyset$. The output sets Δ_k^{out} are given by $\Delta_k^{out} = \Delta_k^+ \cup \Delta_k^-$ with

$$\Delta_k^+ = \bigcup_{i=1,\ldots,n_k} \Delta_{e_{k,i}^+}$$

$$\Delta_k^- = \bigcup_{j=1,\ldots,m} \Delta_{k,not\_e_j^-}$$

where $\Delta_{e_{k,i}^+}$ is the explanation for example $e_{k,i}^+$ and $\Delta_{k,not\_e_j^-}$ is the explanation for $not\_e_j^-$ in the theory $T_k = \langle P \cup H_k, A, I \rangle$.

We will show that each abductive explanation $\Delta_{e_{k,i}^+}$ and $\Delta_{k,not\_e_j^-}$ is a strong abductive explanation in the theory T_k. These explanations are constructed successively with the explanation for each example forming part of the input for the next example. Therefore, if the input sets Δ_k^{in} are strong, then also the individual explanations are strong, by the correctness (with respect to definition 39) of the abductive derivation used by the algorithm and the property of proposition 38 that the union of strong explanations is strong. Note also that the successive test of the examples by the abductive derivation in the algorithm ensures that these individual explanations are self-consistent with each other required for the application of proposition 38.

Hence we need to show that Δ_k^{in} are strong abductive extensions in T_k, for $k = 1, \ldots, l$. We do this by induction on k. For $k = 1$, $\Delta_1^{in} = \emptyset$ which is a strong abductive extension because, by the assumptions on the hypothesis spaces of the integrity constraints and programs, it always satisfies any set of constraints and it trivially satisfies the strong property in definition 34. Suppose that Δ_k^{in} is strong in T_k, we have to prove that Δ_{k+1}^{in} is strong in T_{k+1}. We first prove that Δ_{k+1}^{in} is strong in T_k. $\Delta_{k+1}^{in} = \Delta_k^{in} \cup \Delta_k^{out}$ is the union of strong abductive extensions of T_k: Δ_k^{in} is strong by the inductive hypothesis and Δ_k^{out} is strong because is the union of strong explanations computed successively by the correct abductive derivations of the algorithm starting from the strong extension Δ_k^{in}. Also the derivations ensures that all these explanations are self-consistent with each other. Therefore, by proposition 38, Δ_{k+1}^{in} is a strong abductive extension of T_k.

We still need to show that Δ_{k+1}^{in} is a strong extension of T_{k+1}. This can be done by directly verifying the conditions in the definition 34 of strong abductive extension. Since the integrity constraints I and the background program P do not contain any target predicate, their satisfaction is independent from the addition of any clause for the target predicates. Therefore, as Δ_{k+1}^{in} satisfies I in T_k, it does so in T_{k+1} as well. We also need to show that, for any set Δ' such that $\Delta_{k+1}^{in} \cup \Delta'$ is self-consistent and Δ' satisfies I in T_{k+1}, $\Delta_{k+1}^{in} \cup \Delta'$ must also do so in T_{k+1}. From the independence of I and P from the target predicates, Δ' satisfies I in T_{k+1} implies that Δ' satisfies I in T_k. Since Δ_{k+1}^{in} is strong in T_k, then $\Delta_{k+1}^{in} \cup \Delta'$ satisfies I in T_k. Again, the independence of I and P from the target predicates gives that $\Delta_{k+1}^{in} \cup \Delta'$ satisfies I in T_{k+1}.

We can now show the ACL1 conditions with

$$\Delta^+ = \bigcup_{k=1,\ldots,l} \bigcup_{i=1,\ldots,n_k} \Delta_{e_{k,i}^+}$$

$$\Delta^- = \bigcup_{k=1,\ldots,l \;\; j=1,\ldots,m} \Delta_{k,not\ e_j^-}$$

which by construction are the final sets returned by the ACL1 algorithm. We first show that all the explanations for the individual examples are strong abductive explanations in the final theory $T_l = T_{ACL1}$ from the fact that they are strong in their respective theories T_k. This follows in the same way as we have shown above that Δ_{k+1}^{in} is strong in T_{k+1} from the fact that it is strong in T_k.

We also know that all these individual explanations are self-consistent with each other. This follows directly from their successive construction in the algorithm satisfying the abductive derivability of definition 39. Hence $\Delta^+ \cup \Delta^-$ is self-consistent and the third condition (3) of ACL1 is satisfied. Moreover, by proposition 38, the union Δ^+ is then also a strong abductive explanation of $E_1^+, \ldots, E_l^+$ in T_{ACL1}. From the sufficiency stopping criterion (see figure 4.1) we have that $E_1^+ \cup \ldots \cup E_l^+ = E^+$, therefore Δ^+ is a strong abductive explanation of E^+ in T_{ACL1} and condition A.1 is satisfied. Similarly, by proposition 38, the union Δ^- is a strong abductive explanation of E^- in T_{ACL1} and condition A.2 is satisfied.

□

A.3 Abductive Proof Procedure

In the following we recall the abductive proof procedure for ALP, taken from [KM90c], used as a basis for the abductive coverage procedure in the ACL1 algorithm.

This ALP procedure is applied to abductive theories $T = \langle P, A, I \rangle$ in their three-valued version. Thus the abducibles A contain predicates ($a \in A$) for positive assumptions and predicates ($not\_a \in A$) for negative assumptions. The integrity constraints in I are restricted to have a denial form, $\neg(B_1 \wedge \ldots \wedge B_m \wedge \neg A_1 \wedge \ldots \wedge \neg A_k)$ (written here in Logic Programming style as goals $\leftarrow (B_1, \ldots, B_m, \neg A_1, \ldots, \neg A_k)$, with at least one abducible with no definition in P appearing in $B_1, \ldots, B_m$. Integrity constraints in the range-restricted clausal form,

$A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_m$, are first transformed into the equivalent denial above before they are used by the abductive procedure.

This procedure also assumes that the program P of T contains no definitions for the abducible predicates ie. no rule (or fact) in P has in its head an abducible predicate. When the program contains such definitions the abductive theory $T = \langle P, A, I \rangle$ can be first transformed so that no such definitions exist. For each abducible predicate p that constains that has a partial definition in P we add a new abducible δ_p to the set of abducibles A, we remove p from A and we add the rule $p(\vec{X}) \leftarrow \delta_p(\vec{X})$ to the program P. In this way, if $p(\vec{c})$ can not be derived using the partial definition for p, it can be derived by abducing $\delta_p(\vec{c})$ thus effectively abducing p.

The procedure is composed of two phases: abductive derivation and consistency derivation.

Abductive derivation

An abductive derivation from $(G_1\ \Delta_1)$ to $(G_n\ \Delta_n)$ in $\langle P, A, I \rangle$ via a safe selection rule R, of a literal[1] from a goal, is a sequence

$$(G_1\ \Delta_1), (G_2\ \Delta_2), \ldots, (G_n\ \Delta_n)$$

such that each G_i has the form $\leftarrow L_1, \ldots, L_k$, $R(G_i) = L_j$ and $(G_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

(1) If L_j is not abducible, then $G_{i+1} = C$ and $\Delta_{i+1} = \Delta_i$ where C is the resolvent of some clause in P with G_i on the selected literal L_j;

(2) If L_j is abducible and $L_j \in \Delta_i$, then $G_{i+1} = \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k$ and $\Delta_{i+1} = \Delta_i$;

(3) If L_j is a ground abducible, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$ and there exists a *consistency derivation* from $(\{L_j\}\ \Delta_i \cup \{L_j\})$ to $(\{\}\ \Delta')$ then $G_{i+1} = \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k$ and $\Delta_{i+1} = \Delta'$.

Steps (1) and (2) are SLD-resolution steps with the rules of P and abductive assumptions already computed, respectively. In step (3) a new abductive assumption is required and it is added to the current set of assumptions provided it is consistent.

Consistency derivation

A consistency derivation for an abducible α from $(\alpha,\ \Delta_1)$ to $(F_n\ \Delta_n)$ in $\langle P, A, I \rangle$ is a sequence

$$(\alpha\ \Delta_1), (F_1\ \Delta_1), (F_2\ \Delta_2), \ldots, (F_n\ \Delta_n)$$

where :

(i) F_1 is the union of all goals of the form $\leftarrow L_1, \ldots, L_n$ obtained by resolving the abducible α with the denials in I with no such goal been empty;

(ii) for each $i > 1$, F_i has the form $\{\leftarrow L_1, \ldots, L_k\} \cup F'_i$, for some $j = 1, \ldots, k$ L_j is selected and $(F_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

  (C1) If L_j is not abducible, then $F_{i+1} = C' \cup F'_i$ where C' is the set of all resolvents of clauses in P with $\leftarrow L_1, \ldots, L_k$ on the literal L_j and the empty goal $[] \notin C'$, and $\Delta_{i+1} = \Delta_i$;

  (C2) If L_j is abducible, $L_j \in \Delta_i$ and $k > 1$, then
  $F_{i+1} = \{\leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k\} \cup F'_i$
  and $\Delta_{i+1} = \Delta_i$;

[1] We use the term literal despite the fact that goals contain only positive conditions due to the presence of negative abducible conditions of the form *not_a*.

(C3) If L_j is abducible, $\overline{L_j} \in \Delta_i$ then $F_{i+1} = F'_i$ and $\Delta_{i+1} = \Delta_i$;

(C4) If L_j is a ground abducible, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$, and there exists an *abductive derivation* from $(\leftarrow \overline{L_j}\ \Delta_i)$ to $([]\ \Delta')$ then $F_{i+1} = F'_i$ and $\Delta_{i+1} = \Delta'$;

(C5) If L_j is equal to $\neg A$ with A a ground atom and there exists an *abductive derivation* from $(\leftarrow A\ \Delta_i)$ to $([]\ \Delta')$ then $F_{i+1} = F'_i$ and $\Delta_{i+1} = \Delta'$.

In case (C1) the current branch splits into as many branches as the number of resolvents of $\leftarrow L_1, \ldots, L_k$ with the clauses in P on L_j. If the empty clause is one of such resolvents the whole consistency check fails. In case (C2) the goal under consideration is made simpler if literal L_j belongs to the current set of assumptions Δ_i. In case (C3) the current branch is already consistent under the assumptions in Δ_i, and this branch is dropped from the consistency checking In case (C4) the current branch of the consistency search space can be dropped provided $\leftarrow \overline{L_j}$ is abductively provable. In case (C5), like (C4) the current branch fails and can be dropped provided that we can show that the atom A holds.

Given an initial goal (query) G, the procedure succeeds, and returns the set of abducibles Δ iff there exists an abductive derivation from $(G\ \{\})$ to $([]\ \Delta)$. In this case, we also say that the abductive derivation succeeds.

A.4 Examples 51 and 52

M-ACL was tested on examples 51 and 52 in order to verify its ability to backtrack from a wrong clause and to use negative examples generated from abduction to avoid overgeneralization.

For example 51, the following output was generated:

```
/* Execution time 0.940000 seconds. Generated rules */

rule(ancestor(A,B),[parent(A,B)],c2)
GC: yes, LC: yes
Covered positive examples: [ancestor(b,c),ancestor(a,b)]
Covered positive abduced examples: []
Covered negative abduced examples: []
Abduced literals: []


rule(father(A,B),[male(A),ancestor(A,B)],c13)
GC: yes, LC: yes
Covered positive examples: [father(a,b)]
Covered positive abduced examples: []
Covered negative abduced examples: []
Abduced literals: [[not(ancestor(a,c)),c13]]


rule(ancestor(A,B),[parent(A,C),ancestor(C,B)],c21)
GC: yes, LC: yes
Covered positive examples: [ancestor(d,c)]
Covered positive abduced examples: []
```

```
Covered negative abduced examples: [ancestor(a,c)]
Abduced literals: [[not(ancestor(c,d)),c21],[not(ancestor(b,d)),c21],
        [not(ancestor(c,a)),c21]]

Backtracking: retracting clauses
rule(father(A,B),[male(A),ancestor(A,B)],
        c13,[father(a,b)],[])

rule(father(A,B),[male(A),parent(A,B)],c32)
GC: yes, LC: yes
Covered positive examples: [father(a,b)]
Covered positive abduced examples: []
Covered negative abduced examples: []
Abduced literals: []
```

For example 52, the following output was generated:

```
/* Execution time 0.690000 seconds. Generated rules */

rule(grandfather(A,B),[parent(C,B),father(A,C)],c16)
GC: yes, LC: yes
Covered positive examples: [grandfather(david,jim),grandfather(john,ellen)]
Covered positive abduced examples: []
Covered negative abduced examples: []
Abduced literals: [[not(father(mary,ellen)),c16],[father(david,steve),c16]]


rule(father(A,B),[parent(A,B),male(A)],c44)
GC: yes, LC: yes
Covered positive examples: [father(john,mary)]
Covered positive abduced examples: [father(david,steve)]
Covered negative abduced examples: []
Abduced literals: []
```

Bibliography

[Abe98] A. Abe. The relation between abductive hypotheses and inductive hypotheses. In Flach and Kakas [FK98].

[AD94] H. Adé and M. Denecker. RUTH: An ILP theory revision system. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, 1994.

[AD95] H. Adé and M. Denecker. AILP: Abductive inductive logic programming. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995.

[ADP94] J. J. Alferes, C. V. Damásio, and L. M. Pereira. SLX - A top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *Proc. Int. Symp. on Logic Programming*. The MIT Press, 1994.

[ADRB95] H. Adé, L. De Raedt, and M. Bruynooghe. Declarative bias for specific-to-general ILP systems. *Machine Learning*, 20(1/2):119–154, 1995.

[ALP+98] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming. In *Sixth International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kauffman, 1998.

[AP96] J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111 of *LNAI*. Springer-Verlag, 1996.

[AP98] J. J. Alferes and L. M. Pereira. Tabling abduction. In *Proceedings of the First International Workshop on Tabulation in Parsing and Deduction, TAPD98*, pages 75–82, Paris, France, April 1998.

[APP98] J. J. Alferes, L. M. Pereira, and T. C. Przymusinski. "Classical" negation in non-monotonic reasoning and logic programming. *Journal of Automated Reasoning*, 20:107–142, 1998.

[BDR96] H. Blockeel and L. De Raedt. Inductive database design. In *Proceedings of the 10th International Symposium on Methodologies for Intelligent Systems*, volume 1079 of *Lecture Notes in Artificial Intelligence*, pages 376–385. Springer-Verlag, 1996.

[BG93] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In R. Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1044–1049. Morgan Kaufmann, 1993.

[BG94a] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[BG94b] F. Bergadano and D. Gunetti. Learning clauses by tracing derivations. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 11–30. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[BG95] F. Bergadano and D. Gunetti. *Inductive Logic Programming.* MIT press, 1995.

[BGS88] A. Bergadano, A. Giordana, and L. Saitta. Automated concept acquisition in noisy environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):555–578, 1988.

[BLMM97] A. Brogi, E. Lamma, P. Mancarella, and P. Mello. A unifying view for logic programming with non-monotonic reasoning. *Theoretical Computer Science*, 184:1–59, 1997.

[BM91] M. Bain and S. Muggleton. Non-monotonic learning. In J.E. Hayes-Michie and E. Tyugu, editors, *Machine Intelligence*, volume 12. Oxford University Press, 1991.

[BM92] M. Bain and S. Muggleton. Non-monotonic learning. In S. Muggleton, editor, *Inductive Logic Programming*, pages 145–161. Academic Press, 1992.

[Car89] J. G. Carbonell. Introduction: Paradigms for machine learning. *Artificial Intelligence*, 40(1-3):1–9, 1989.

[CB89] P. Clark and R. Boswell. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, 1989.

[Ces90] B. Cestnik. Estimating probabilities: A crucial task in machine learning. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 147–149, London, 1990. Pitman.

[CKB87] B. Cestnik, I. Knononenko, and I. Bratko. ASSISTANT 86: A knowledge elicitation tool for sophisticated users. In I. Bratko and N. Lavrač, editors, *Progress in Machine Learning*, pages 31–45. Sigma Press, Wilmslow, UK, 1987.

[CKRP73] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un systeme de communication homme-machine en Françis. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[Cla78] K. L. Clark. Negation as failure. In *Logic and Databases*. Plenum Press, 1978.

[CMM83] J. G. Carbonell, R. S. Michalski, and T. M. Mitchell. An overview of machine learning. In Michalski et al. [MCM83], pages 3–24.

[Coh92] W. W. Cohen. Abductive explanation-based learning: A solution to the multiple inconsistent explanation problem. *Machine Learning*, 8:167–219, 1992.

[DB92] S. Džeroski and I. Bratko. Handling noise in inductive logic programming. In S. Muggleton, editor, *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, Report ICOT TM-1182, 1992.

[DDS92] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for normal abductive programs. In K. R. Apt, editor, *Proceedings of the International Joint Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[Dix95] J. Dix. A classification-theory of semantics of normal logic programs: I. & II. *Fundamenta Informaticae*, XXII(3):227–255 and 257–288, 1995.

[DK95] Y. Dimopoulos and A. C. Kakas. Learning non-monotonic logic programs: Learning exceptions. In *Proceedings of the 8th European Conference on Machine Learning*, 1995.

[DK96] Y. Dimopoulos and A. C. Kakas. Abduction and inductive learning. In *Advances in Inductive Logic Programming*. IOS Press, 1996.

[DP97] C. V. Damásio and L. M. Pereira. Abduction on 3-valued extended logic programs. In V. W. Marek, A. Nerode, and M. Trusczynski, editors, *Logic Programming and Non-Monotonic Reasoning - Proc. of 3rd International Conference LPNMR'95*, volume 925 of *LNAI*, pages 29–42, Germany, 1997. Springer-Verlag.

[DP98] C. V. Damásio and L. M. Pereira. A survey on paraconsistent semantics for extended logic programs. In Gabbay and Smets, editors, *Handbook on Quantitative Reasoning: Negation and Inconsistency*, volume 2. Oxford University Press, 1998.

[DPP97] J. Dix, L. M. Pereira, and T. Przymusinski. Prolegomena to logic programming and non-monotonic reasoning. In J. Dix, L. M. Pereira, and T. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming - Selected papers from NMELP'96*, number 1216 in LNAI, pages 1–36, Germany, 1997. Springer-Verlag.

[DR92] L. De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, 1992.

[DR97] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95(1):187–201, 1997.

[DRB89] L. De Raedt and M. Bruynooghe. Towards friendly concept-learners. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 849–856. Morgan Kaufmann, 1989.

[DRB90] L. De Raedt and M. Bruynooghe. On negation and three-valued logic in interactive concept learning. In *Proceedings of the 9th European Conference on Artificial Intelligence*, 1990.

[DRB92a] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.

[DRB92b] L. De Raedt and M. Bruynooghe. Interactive concept learning and constructive induction by analogy. *Machine Learning*, 8(2):107–150, 1992.

[DRB93] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993.

[DRBM91] L. De Raedt, M. Bruynooghe, and B. Martens. Integrity constraints and interactive concept-learning. In L. Birnbaum and G. Collins, editors, *Proceedings of the 8th International Workshop on Machine Learning*, pages 394–398. Morgan Kaufmann, 1991.

[DRD94] L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[DRD96a] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 1996. (To appear).

[DRD96b] L. De Raedt and L. Dehaspe. DLAB a declarative language bias for concept learning and knowledge discovery engines. Technical Report CW214, Katholieke Universiteit Leuven, 1996.

[DRD96c] L. De Raedt and L. Dehaspe. Learning from satisfiability. Technical report, Katholieke Universiteit Leuven, 1996.

[DRL95] L. De Raedt and W. Van Lear. Inductive constraint logic. In *Proceedings of the 5th International Workshop on Algorithmic Learning Theory*, 1995.

[DRLD93] L. De Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 221–240. J. Stefan Institute, 1993.

[Dun91] P. M. Dung. Negation as hypothesis: An abductive foundation for logic programming. In *Proceedings of the Eigth Int. Conf. on Logic Programming, ICLP91*, pages 3–17. The MIT Press, 1991.

[dV89] W. Van de Velde. IDL, or taming the multiplexer problem. In Morik K., editor, *Proceedings of the 4th European Working Session on Learning*. Pittman, 1989.

[Dže91] S. Džeroski. Handling noise in inductive logic programming. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, 1991.

[EFL+98] F. Esposito, S. Ferilli, E. Lamma, P. Mello, M. Milano, F. Riguzzi, and G. Semeraro. Cooperation of abduction and induction in logic programming. In P. A. Flach and A. C. Kakas, editors, *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer, 1998. Submitted for publication.

[EK89] K. Eshghi and R. A. Kowalski. Abduction compared with Negation by Failure. In *Proceedings of the 6th International Conference on Logic Programming*, 1989.

[ELM+96] F. Esposito, E. Lamma, D. Malerba, P. Mello, M. Milano, F. Riguzzi, and G. Semeraro. Learning abductive logic programs. In Flach and Kakas [FK96]. Available on-line at `http://www.cs.bris.ac.uk/~flach/ECAI96/`.

[FK96] P. A. Flach and A. C. Kakas, editors. *Proceedings of the ECAI'96 Workshop on Abductive and Inductive Reasoning*, 1996. Available on-line at `http://www.cs.bris.ac.uk/~flach/ECAI96/`.

[FK97] P. A. Flach and A. C. Kakas, editors. *Proceedings of the IJCAI'97 Workshop on Abductive and Inductive Reasoning*, 1997. Available on-line at `http://www.cs.bris.ac.uk/~flach/IJCAI97/`.

[FK98] P. A. Flach and A. C. Kakas, editors. *Abductive and Inductive Reasoning*. Pure and Applied Logic. Kluwer, 1998.

[Fla92] P. A. Flach. Logical approaches to machine learning - An overview. *THINK*, 1(2):25–36, 1992.

[Fla95] P. A. Flach. *Conjectures: An Inquiry Concerning the Logic of Induction*. PhD thesis, Katholieke Universiteit Brabant, 1995.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th Int. Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GL90] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming ICLP90*, pages 579–597. The MIT Press, 1990.

[GM90] L. Giordano and A. Martelli. Generalized stable model semantics, truth maintenance and conflict resolution. In *Proceedings of the 7th International Conference on Logic Programming*, pages 427–411, Jerusalem, 1990. MIT Press.

[Hel89] N. Helft. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–156. Morgan Kaufmann, 1989.

[IK97] K. Inoue and Y. Kudoh. Learning extended logic programs. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 176–181. Morgan Kaufmann, 1997.

[IKI+96] N. Inuzuka, M. Kamo, N. Ishii, H. Seki, and H. Itoh. Top-down induction of logic programs from incomplete samples. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, pages 119–136. Stockholm University, Royal Institute of Technology, 1996.

[Ino94] K. Inoue. Hypothetical reasoning in logic programs. *Journal of Logic Programming*, 18:191–227, 1994.

[Ino98] K. Inoue. Learning abductive and nonmonotonic logic programs. In P. A. Flach and A. C. Kakas, editors, *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer, 1998. Submitted for publication.

[IS94] K. Inoue and C. Sakama. On the equivalence between disjunctive and abductive logic programs. In *In proceedings of ICLP94*, pages 489–503, 1994.

[IS95] K. Inoue and C. Sakama. Abductive framework for nonmonotonic theory change. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 204–210, 1995.

[Kal79] J. Kalbfleish. *Probability and Statistical Inference*, volume II. Springer, New York, 1979.

[KK98] T. Kanai and S. Kunifuji. Extending inductive generalization with abduction. In Flach and Kakas [FK98].

[KKT93] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1993.

[KKT97] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 5, pages 233–306. Oxford University Press, 1997.

[KM90a] A. C. Kakas and P. Mancarella. Database updates through abduction. In R. Sacks-Davis D. McLeod and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Databases, VLDB-90*, pages 650–661. Morgan Kaufmann, 1990.

[KM90b] A. C. Kakas and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proceedings of the 9th European Conference on Artificial Intelligence*, 1990.

[KM90c] A. C. Kakas and P. Mancarella. On the relation between truth maintenance and abduction. In *Proceedings of the 2nd Pacific Rim International Conference on Artificial Intelligence*, 1990.

[KMD94] A. C. Kakas, P. Mancarella, and P. M. Dung. The acceptability semantics for logic programs. In *Proceedings of the 11th International Conference on Logic Programming*, 1994.

[Kow74] R. A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP74*, pages 569–574. North Holland Publishing Co., 1974.

[KR97] A. C. Kakas and F. Riguzzi. Learning with abduction. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, 1997.

[KR98] A. C. Kakas and F. Riguzzi. Learning with abduction. submitted for publication, 1998.

[LD92] N. Lavrač and S. Džeroski. Inductive learning of relations from noisy examples. In S. Muggleton, editor, *Inductive Logic Programming*, pages 495–516. Academic Press, 1992.

[LD94] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.

[LDB96] N. Lavrač, S. Džeroski, and I. Bratko. Handling imperfect data in inductive logic programming. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 48–64. IOS Press, 1996.

[LDG91a] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with LINUS. In Y. Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 265–281. Springer-Verlag, 1991.

[LDG91b] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with LINUS. In Y. Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 265–281. Springer-Verlag, 1991.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, Berlin, second edition, 1987.

[LM92] S. Lapointe and S. Matwin. Sub-unification: A tool for efficient induction of recursive programs. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Workshop on Machine Learning*, pages 273–281. Morgan Kaufmann, 1992.

[LMMR97] E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating induction and abduction in logic programming. In P. P. Wang, editor, *Proceedings of the Third Joint Conference on Information Sciences*, volume 2, pages 203–206, 1997.

[LMMR98] E. Lamma, P. Mello, M. Milano, and F. Riguzzi. Integrating induction and abduction in logic programming. To appear on Information Sciences, 1998.

[LP98] J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Collected Papers from Workshop on Logic Programming and Knowledge Representation LPKR'97*, number 1471 in LNAI. Springer-Verlag, 1998.

[LRP88a] E. Lamma, F. Riguzzi, and L. M. Pereira. Learning in a three-valued setting. In *Proceedings of the Fourth International Workshop on Multistrategy Learning*, 1988.

[LRP88b] E. Lamma, F. Riguzzi, and L. M. Pereira. Learning with extended logic programs. In *Proceedings of the Logic Programming track of the Seventh International Workshop on Nonmonotonic Reasoning (LP-NMR98)*, 1988.

[LRP88c] E. Lamma, F. Riguzzi, and L. M. Pereira. Strategies in combined learning via logic programs. Technical report, DEIS - University of Bologna, 1988.

[MB92] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In S. Muggleton, editor, *Inductive Logic Programming*, pages 261–280. Academic Press, 1992.

[MCM83] R. Michalski, J. G. Carbonell, and T. M. Mitchell, editors. *Machine Learning - An Artificial Intelligence Approach.* Springer-Verlag, 1983.

[MCM86] R. Michalski, J. G. Carbonell, and T. M. Mitchell, editors. *Machine Learning - An Artificial Intelligence Approach Vol. II.* Morgan Kaufmann, 1986.

[MDR94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[Mic73] R. Michalski. Discovery classification rules using variable-valued logic system VL1. In *Proceedings of the Third International Conference on Artificial Intelligence*, pages 162–172. Stanford University, 1973.

[Mic80] R.S. Michalski. Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.

[Mic84] R. Michalski. A theory and methodology of inductive learning. In R. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach*, volume 1, pages 83–134. Springer-Verlag, 1984.

[Mic86] R. S. Michalski. Understanding the nature of learning: Issues and research directions. In Michalski et al. [MCM86], pages 3–26.

[Mit82] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[Mit97] T. M. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[Moo98] R. Mooney. Integrating abduction and induction in machine learning. In Flach and Kakas [FK98].

[Mor91] K. Morik. Balanced cooperative modelling. In *Proc. First Int. Workshop on Multistrategy Learning*, Fairfax, VA, 1991. George Mason University.

[Mug95a] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[Mug95b] S. Muggleton. Inverting entailment and Progol. In *Machine Intelligence*, volume 14, pages 133–188. Oxford University Press, 1995.

[MV95a] L. Martin and C. Vrain. MULT_ICN: An empirical multiple predicate learner. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 129–144. Department of Computer Science, Katholieke Universiteit Leuven, 1995.

[MV95b] L. Martin and C. Vrain. A three-valued framework for the induction of general program. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 109–128. Department of Computer Science, Katholieke Universiteit Leuven, 1995.

[NB86] T. Niblett and I. Bratko. Learning decision rules in noisy domains. In M. Bramer, editor, *Research and Development in Expert Systems III*, pages 24–25. Cambridge University Press, 1986.

[O'R94] P. O'Rourke. Abduction and explanation-based learning: Case studies in diverse domains. *Computational Intelligence*, 10:295–330, 1994.

[PA92] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In *Proceedings of the European Conference on Artificial Intelligenece ECAI92*, pages 102–106. John Wiley and Sons, 1992.

[PGA87] D. Poole, R. G. Goebel, and Aleliunas. Theorist: a logical reasoning system for default and diagnosis. In Cercone and McCalla, editors, *The Knowledge Fronteer: Essays in the Representation of Knowledge*, Lecture Notes in Computer Science, pages 331–352. Springer-Verlag, 1987.

[PK92] M.J. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.

[Plo70] G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[Plo71] G.D. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.

[Qui90a] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Qui90b] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Qui91] J. R. Quinlan. Unknown attribute values in induction. In *Proceedings of the Sixth International Machine Learning Workshop*, pages 164–168, San Mateo, CA, 1991. Morgan Kaufmann.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[Rei78] R. Reiter. On closed-word data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rus89] S. Russell. *The Use of Knowledge in Analogy and Induction*. Pitman, London, 1989.

[Sak98] C. Sakama. Computing induction through abduction. In Flach and Kakas [FK98].

[SB86] C. Sammut and R. Banerji. Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Volume 2*, pages 167–191. Morgan Kaufmann, 1986.

[Sha83] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[SI92] K. Satoh and N. Iwayama. A query evaluation method for abductive logic programming. In *In proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, pages 671–685, 1992.

[Sim83] H. A. Simon. Why should machines learn. In Michalski et al. [MCM83], pages 25–37.

[SSW+97] K. F. Sagonas, T. Swift, D. S. Warren, J. Freire, and P. Rao. *The XSB Programmer's Manual Version 1.7.1*, 1997.

[Swe97] Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual*, 1997.

[TM94] C. Thompson and R. Mooney. Inductive learning for abductive diagnosis. In *Proceedings of the 12th National Conference on Artificial Intelligence*, 1994.

[VGRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[VLDDR94] W. Van Laer, L. Dehaspe, and L. De Raedt. Applications of a logical discovery engine. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 263–274, 1994.

[WD95] S. Wrobel and S. Džeroski. The ILP description learning problem: Towards a genearl model-leve definition of data mining in ILP. In *Proceedings of the Fachgruppentreffen Maschinelles Lernen*, 1995.

[Wro88] S. Wrobel. Automatic representation adjustment in an observational discovery system. In D. Sleeman, editor, *Proceedings of the 3rd European Working Session on Learning*, pages 253–262. Pitman, 1988.