Luc De Raedt

Logical and Relational Learning

Springer

Cognitive Technologies

H. Prendinger, M. Ishizuka (Eds.)
Life-Like Characters
Tools, Affective Functions, and Applications
IX, 477 pages. 2004

H. Helbig
Knowledge Representation and the Semantics of Natural Language
XVIII, 646 pages. 2006

P.M. Nugues
An Introduction to Language Processing with Perl and Prolog
An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German
XX, 513 pages. 2006

W. Wahlster (Ed.)
SmartKom: Foundations of Multimodal Dialogue Systems
XVIII, 644 pages. 2006

B. Goertzel, C. Pennachin (Eds.)
Artificial General Intelligence
XVI, 509 pages. 2007

O. Stock, M. Zancanaro (Eds.)
PEACH — Intelligent Interfaces for Museum Visits
XVIII, 316 pages. 2007

V. Torra, Y. Narukawa
Modeling Decisions: Information Fusion and Aggregation Operators
XIV, 284 pages. 2007

P. Manoonpong
Neural Preprocessing and Control of Reactive Walking Machines
Towards Versatile Artificial Perception–Action Systems
XVI, 185 pages. 2007

S. Patnaik
Robot Cognition and Navigation
An Experiment with Mobile Robots
XVI, 290 pages. 2007

M. Cord, P. Cunningham (Eds.)
Machine Learning Techniques for Multimedia
Case Studies on Organization and Retrieval
XVI, 290 pages. 2008

L. De Raedt
Logical and Relational Learning
XVI, 388 pages. 2008

Cognitive Technologies

Luc De Raedt

Logical and Relational Learning

With 77 Figures and 10 Tables

Springer

Author:

Prof. Dr. Luc De Raedt
Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Heverlee
Belgium
luc.deraedt@cs.kuleuven.be

ISBN: 978-3-540-20040-6 e-ISBN: 978-3-540-68856-3

Cognitive Technologies ISSN: 1611-2482

Library of Congress Control Number: 2008933706

ACM Computing Classification (1998): I.2.6, I.2.4, H.2.8, I.5.1

Cover design: KuenkelLopka GmbH

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

To Lieve, Maarten and Soetkin

Preface

I use the term *logical and relational learning* to refer to the subfield of artificial intelligence, machine learning and data mining that is concerned with learning in expressive logical or relational representations. It is the union of inductive logic programming, (statistical) relational learning and multi-relational data mining, which all have contributed techniques for learning from data in relational form. Even though some early contributions to logical and relational learning are about forty years old now, it was only with the advent of inductive logic programming in the early 1990s that the field became popular. Whereas initial work was often concerned with logical (or logic programming) issues, the focus has rapidly changed to the discovery of new and interpretable knowledge from structured data, often in the form of rules, and soon important successes in applications in domains such as bio- and chemo-informatics and computational linguistics were realized. Today, the challenges and opportunities of dealing with structured data and knowledge have been taken up by the artificial intelligence community at large and form the motivation for a lot of ongoing research. Indeed, graph, network and multi-relational data mining are now popular themes in data mining, and statistical relational learning is receiving a lot of attention in the machine learning and uncertainty in artificial intelligence communities. In addition, the range of tasks for which logical and relational techniques have been developed now covers almost all machine learning and data mining tasks. On the one hand these developments have resulted in a new role and novel views on logical and relational learning, but on the other hand have also made it increasingly difficult to obtain an overview of the field as a whole.

This book wants to address these needs by providing a new synthesis of logical and relational learning. It constitutes an attempt to summarize some of the key results about logical and relational learning, it covers a wide range of topics and techniques, and it describes them in a uniform and accessible manner. While the author has tried to select a representative set of topics and techniques from the field of logical and relational learning, he also realizes that he is probably biased by his own research interests and views on the field. Furthermore, rather than providing detailed accounts of the many specific systems and techniques, the book focuses on the underlying principles, which should enable the reader to easily get access to and understand the relevant literature on logical and relational learning. Actually, at the end of each chapter, suggestions for further reading are provided.

The book is intended for graduate students and researchers in artificial intelligence and computer science, especially those in machine learning, data mining, uncertainty in artificial intelligence, and computational logic, with an interest in learning from relational data. The book is the first textbook on logical and relational learning and is suitable for use in graduate courses, though it can also be used for self-study and as a reference. It contains many different examples and exercises. Teaching material will become available from the author's website. The author would also appreciate receiving feedback, suggestions for improvement and needed corrections by email to luc.deraedt@cs.kuleuven.be.

The book starts with an introductory chapter clarifying the nature, motivations and history of logical and relational learning. Chapter 2 provides a gentle introduction to logic and logic programming, which will be used throughout the book as the representation language. Chapter 3 introduces the idea of learning as search and provides a detailed account of some fundamental machine learning algorithms that will play an important role in later chapters. In Chapter 4, a detailed study of a hierarchy of different representations that are used in machine learning and data mining is given, and two techniques (propositionalization and aggregation) for transforming expressive representations into simpler ones are introduced. Chapter 5 is concerned with the theoretical basis of the field. It studies the generality relation in logic, the relation between induction and deduction, and introduces the most important framework and operators for generality. In Chapter 6, a methodology for developing logical and relational learning systems is presented and illustrated using a number of well-known case studies that learn relational rules, decision trees and frequent queries. The methodology starts from existing learning approaches and upgrades them towards the use of rich representations. Whereas the first six chapters are concerned with the foundations of logical and relational learning, the chapters that follow introduce more advanced techniques. Chapter 7 focuses on learning the definition of multiple relations, that is, on learning theories. This chapter covers abductive reasoning, using integrity constraints, program synthesis, and the use of an oracle. Chapter 8 covers statistical relational learning, which combines probabilistic models with logical and relational learning. The chapter starts with a gentle introduction to graphical models before turning towards probabilistic logics. The use of kernels and distances for logical and relational learning is addressed in Chapter 9, and in Chapter 10 computational issues such as efficiency considerations and learnability results are discussed. Finally, Chapter 11 summarizes the most important lessons learned about logical and relational learning. The author suggests to read it early on, possibly even directly after Chapter 1.

An introductory course to logical and relational learning covers most of the materials in Chapters 1 to 4, Sects. 5.1 – 5.4, 5.9, and Chapters 6 and 11. The other chapters do not depend on one another, and, hence, further chapters can be selected according to the interests and the preferences of the reader. Given the interests in statistical relational learning, the author certainly recommends Chapter 8. Advanced sections and exercises are marked with a * or even with **. They are more challenging, but can be skipped without loss of continuity.

This book could not have been written without the help and encouragement of many persons. The author is indebted to a number of co-workers who contributed ideas, techniques, surveys and views that have found their way into this book, including: Maurice Bruynooghe for influencing the use of logic in this book and numerous suggestions for improvement; Hendrik Blockeel, Luc Dehaspe and Wim Van Laer for contributions to the upgrading methodology described in Chapter 6, Kristian Kersting for joint work on statistical relational learning presented in Chapter 8, and Jan Ramon for his work on distances in Chapter 9. This book has also taken inspiration from a number of joint overview papers and tutorials that the author delivered in collaboration with Hendrik Blockeel, Sašo Džeroski, Kristian Kersting, Nada Lavrač and Stephen Muggleton. The author would also like to thank the editor at Springer, Ronan Nugent, for his patience, help, and support during all phases of this book-writing project.

The author is grateful for the feedback and encouragement on the many earlier versions of this book he received. He would like to thank especially: the reading clubs at the University of Bristol (headed by Peter Flach, and involving Kerstin Eder, Robert Egginton, Steve Gregory, Susanne Hoche, Simon Price, Simon Rawles and Ksenia Shalonova) and at the Katholieke Universiteit Leuven (Hendrik Blockeel, Björn Bringmann, Maurice Bruynooghe, Fabrizio Costa, Tom Croonenborghs, Anton Dries, Kurt Driessens, Daan Fierens, Christophe Costa Florencio, Elisa Fromont, Robby Goetschalkx, Bernd Gutmann, Angelika Kimmig, Niels Landwehr, Wannes Meert, Siegfried Nijssen, Stefan Raeymaekers, Leander Schietgat, Jan Struyf, Ingo Thon, Anneleen van Assche, Joaquin Vanschoren, Celine Vens, and Albrecht Zimmermann), several colleagues who gave feedback or discussed ideas in this book (James Cussens, Paolo Frasconi, Thomas Gaertner, Tamas Horvath, Manfred Jaeger, Martijn van Otterlo), and the students in Freiburg who used several previous versions of this book.

Last but not least I would like to thank my wife, Lieve, and my children, Soetkin and Maarten, for their patience and love during the many years it took to write this book. I dedicate this book to them.

Destelbergen, May 2008 *Luc De Raedt*

Contents

1

Introduction

The field of logical and relational learning, which is introduced in this chapter, is motivated by the limitations of traditional symbolic machine learning and data mining systems that largely work with propositional representations. These limitations are clarified using three case studies: predicting the activity of chemical compounds from their structure; link mining, where properties of websites are discovered; and learning a simple natural language interface for a database. After sketching how logical and relational learning works, the chapter ends with a short sketch of the history of this subfield of machine learning and data mining as well as a brief overview of the rest of this book.

1.1 What Is Logical and Relational Learning?

Artificial intelligence has many different subfields. Logical and relational learning combines principles and ideas of two of the most important subfields of artificial intelligence: machine learning and knowledge representation. Machine learning is the study of systems that improve their behavior over time with experience. In many cases, especially in a data mining context, the experience consists of a set of observations or examples in which one searches for patterns, regularities or classification rules that provide valuable new insights into the data and that should ideally be readily interpretable by the user. The learning process then typically involves a search through various generalizations of the examples.

In the past fifty years, a wide variety of machine learning techniques have been developed (see Mitchell [1997] or Langley [1996] for an overview). However, most of the early techniques were severely limited from a knowledge representation perspective. Indeed, most of the early techniques (such as decision trees [Quinlan, 1986], Bayesian networks [Pearl, 1988], perceptrons [Nilsson, 1990], or association rules [Agrawal et al., 1993]) could only handle data and generalizations in a limited representation language, which was essentially propositional. Propositional representations (based on boolean or propositional logic) cannot elegantly represent domains involving multiple entities as well as the relationships amongst them. One such domain is that of social networks where the persons are the entities and their social interactions are characterized by their relationships. The representational limitations of the early machine learning systems carried over to the resulting learning and data mining techniques, which were in turn severely limited in their application domain. Various machine learning researchers, such as Ryszard Michalski [1983] and Gordon Plotkin [1970], soon realized these limitations and started to employ more expressive knowledge representation frameworks for learning. Research focused on using frameworks that were able to represent a variable number of entities as well as the relationships that hold amongst them. Such representations are called *relational*. When they are grounded in or derived from first-order logic they are called *logical* representations. The interest in learning using these expressive representation formalisms soon resulted in the emergence of a new subfield of artificial intelligence that I now describe as logical and relational learning.

Logical and relational learning is thus viewed in this book as the study of machine learning and data mining within expressive knowledge representation formalisms encompassing relational or first-order logic. It specifically targets learning problems involving multiple entities and the relationships amongst them. Throughout the book we shall mostly be using logic as a representation language for describing data and generalizations, because logic is inherently relational, it is expressive, understandable, and interpretable, and it is well understood. It provides solid theoretical foundations for many developments within artificial intelligence and knowledge representation. At the same time, it enables one to specify and employ background knowledge about the domain, which is often also a key factor determining success in many applications of artificial intelligence.

1.2 Why Is Logical and Relational Learning Important?

To answer this question, let us look at three important applications of logical and relational learning. The first is concerned with learning to classify a set of compounds as active or inactive [Srinivasan et al., 1996], the second with analyzing a website [Craven and Slattery, 2001], and the third with learning a simple natural language interface to a database system [Mooney, 2000]. In these applications there are typically a variable number of entities as well as relationships amongst them. This makes it very hard, if not impossible, to use more traditional machine learning methods that work with fixed feature vectors or attribute-value representations. Using relational or logical representations, these problems can be alleviated, as we will show throughout the rest of this book.

1.2.1 Structure Activity Relationship Prediction

Consider the compounds shown in Fig. 1.1. Two of the molecules are active and two are inactive. The learning task now is to find a pattern that discriminates the actives from the inactives. This type of task is an important task in computational chemistry. It is often called *structure activity relationship prediction* (SAR), and it forms an essential step in understanding various processes related to drug design and discovery [Srinivasan and King, 1999b], toxicology [Helma, 2005], and so on. The figure also shows a so-called *structural alert*, which allows one to distinguish the actives from the inactives because the structural alert is a substructure (subgraph) that matches both of the actives but none of the inactives. At the same time, the structural alert is readily interpretable and provides useful insights into the factors determining the activity.

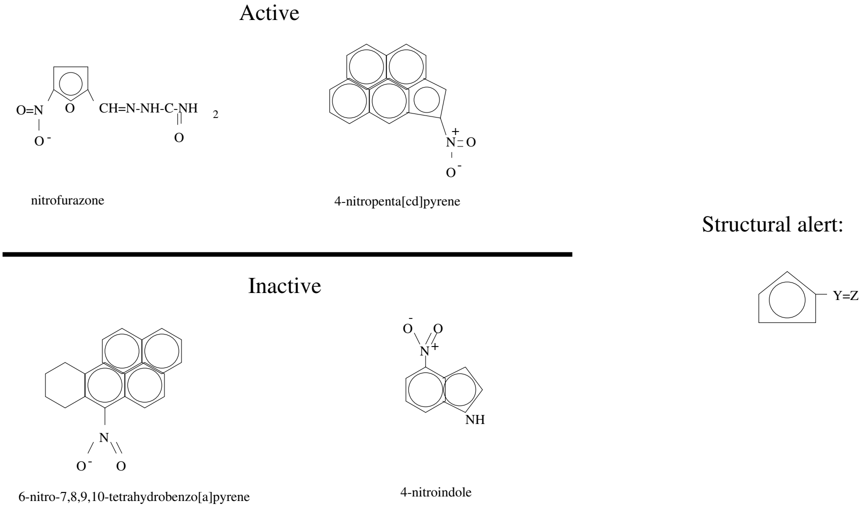


Fig. 1.1. Predicting mutagenicity. Reprinted from [Srinivasan et al., 1996], page 288, ©1996, with permission from Elsevier

Traditional machine learning methods employ the *single-tuple single-table assumption*, which assumes that the data can be represented using attribute-value pairs. Within this representation, each example (or compound) corresponds to a single row or tuple in a table, and each feature or attribute to a single column; cf. Table 1.1. For each example and attribute, the cells of the table specify the value of the attribute for the specific example, for instance, whether or not a benzene ring is present. We shall call representations that can easily be mapped into the single-tuple single-table format *propositional*.

Table 1.1. A table-based representation

Compound	Attribute1	Attribute2	Attribute3	Class
1	true	false	true	active
2	true	true	true	active
3	false	false	true	inactive
4	true	false	true	inactive

From a user perspective, the difficulty with this type of representation is the mismatch between the graphical and structured two-dimensional representation in the molecules and the flat representation in the table. In order to use the flat representation, the user must first determine the features or attributes of interest. In structure activity relationship prediction, these are sometimes called *fingerprints*. Finding these features is in itself a non-trivial task as there exist a vast number of potentially interesting features. Furthermore, the result of the learning process will critically depend on the quality of the employed features. Even though there exist a number of specialized tools to tackle this kind of task (involving the use of libraries of fingerprints), the question arises as to whether there exist general-purpose machine learning systems able to cope with such structured representations directly. The answer to this question is affirmative, as logical and relational learners directly deal with structured data.

Example 1.1. The graphical structure of one of the compounds can be represented by means of the following tuples, which we call *facts*:

$$\begin{array}{ll}
\mathsf{active}(\mathsf{f1}) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_1, \mathsf{f1}_2, 7) \leftarrow \\
\mathsf{logmutag}(\mathsf{f1}, 0.64) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_2, \mathsf{f1}_3, 7) \leftarrow \\
\mathsf{lumo}(\mathsf{f1}, -1.785) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_3, \mathsf{f1}_4, 7) \leftarrow \\
\mathsf{logp}(\mathsf{f1}, 1.01) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_4, \mathsf{f1}_5, 7) \leftarrow \\
\mathsf{atom}(\mathsf{f1}, \mathsf{f1}_1, \mathsf{c}, 21, 0.187) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_8, \mathsf{f1}_9, 2) \leftarrow \\
\mathsf{atom}(\mathsf{f1}, \mathsf{f1}_2, \mathsf{c}, 21, -0.143) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_8, \mathsf{f1}_{10}, 2) \leftarrow \\
\mathsf{atom}(\mathsf{f1}, \mathsf{f1}_3, \mathsf{c}, 21, -0.143) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_1, \mathsf{f1}_{11}, 1) \leftarrow \\
\mathsf{atom}(\mathsf{f1}, \mathsf{f1}_4, \mathsf{c}, 21, -0.013) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_{11}, \mathsf{f1}_{12}, 2) \leftarrow \\
\mathsf{atom}(\mathsf{f1}, \mathsf{f1}_5, \mathsf{o}, 52, -0.043) \leftarrow & \mathsf{bond}(\mathsf{f1}, \mathsf{f1}_{11}, \mathsf{f1}_{13}, 1) \leftarrow \\
\ldots &
\end{array}$$

In this encoding, each entity is given a name and the relationships among the entities are captured. For instance, in the above example, the compound is named $\mathsf{f1}$ and its atoms $\mathsf{f1}_1, \mathsf{f1}_2, \ldots$. Furthermore, the relation $\mathsf{atom}/5$ of arity 5 states properties of the atoms: the molecule they occur in (e.g., $\mathsf{f1}$), the element (e.g., c denoting a carbon) and the type (e.g., 21) as well as the charge (e.g., 0.187). The relationships amongst the atoms are then captured by the relation bond/3, which represents the bindings amongst the atoms. Finally, there are also overall properties or attributes of the molecule, such as their *logp* and *lumo* values. Further properties of the compounds could be mentioned, such as the functional groups or ring structures they contain:

$$\begin{array}{l} \mathsf{ring\_size\_5}(\mathsf{f1}, [\mathsf{f1}_5, \mathsf{f1}_1, \mathsf{f1}_2, \mathsf{f1}_3, \mathsf{f1}_4]) \leftarrow \\ \mathsf{hetero\_aromatic\_5\_ring}(\mathsf{f1}, [\mathsf{f1}_5, \mathsf{f1}_1, \mathsf{f1}_2, \mathsf{f1}_3, \mathsf{f1}_4]) \leftarrow \\ \ldots \end{array}$$

The first tuple states that there is a ring of size 5 in the compound f1 that involves the atoms $\mathsf{f1}_5, \mathsf{f1}_1, \mathsf{f1}_2, \mathsf{f1}_3$ and $\mathsf{f1}_4$ in molecule f1; the second one states that this is a heteroaromatic ring.

Using this representation it is possible to describe the structural alert in the form of a rule

$$\mathsf{active}(\mathsf{M}) \leftarrow \mathsf{ring\_size\_5}(\mathsf{M}, \mathsf{R}), \mathsf{element}(\mathsf{A1}, \mathsf{R}), \mathsf{bond}(\mathsf{M}, \mathsf{A1}, \mathsf{A2}, 2)$$

which actually reads as[1]:

> Molecule M is active IF it contains a ring of size 5 called R and atoms A1 and A2 that are connected by a double (2) bond such that A1 also belongs to the ring R.

The previous example illustrates the use of logical representations for data mining. It is actually based on the well-known mutagenicity application of relational learning due to Srinivasan et al. [1996], where the structural alert was discovered using the inductive logic programming system Progol [Muggleton, 1995] and the representation employed above. The importance of this type of application is clear when considering that the results were published in the scientific literature in the application domain [King and Srinivasan, 1996], that they were obtained using a general-purpose inductive logic programming algorithm and were transparent to the experts in the domain. The combination of these factors has seldom been achieved in artificial intelligence.

1.2.2 A Web Mining Example

While structure activity relationship prediction involves the mining of a (potentially large) set of small graphs, *link mining* and discovery is concerned with the analysis of a single large graph or network [Getoor, 2003, Getoor and Dielh, 2005]. To illustrate link mining, we consider the best known example of a network, that is, the Internet, even though link mining is applicable in

[1] This form of description is sometimes called 'Sternberg' English in inductive logic programming, after the computational biologist Michael Sternberg, who has been involved in several pioneering scientific applications of logical learning.

other contexts as well, for instance, in social networks, protein networks, and bibliographic databases. The following example is inspired by the influential WebKB example of Craven and Slattery [2001].

Example 1.2. Consider the website of a typical university. It contains several web pages that each describe a wide variety of entities. These entities belong to a wide variety of classes, such as student, faculty, staff, department, course, project, and publication. Let us assume that for each such web page, there is a corresponding tuple in our relational database. Consider, for instance, the following facts (where the urls denote particular URLs):

faculty(url1, stephen) ← faculty(url2, john) ←
course(url3, logic_for_learning) ← project(url4, april2) ←
department(url5, computer_science) ← student(url6, hiroaki) ←
. . .

In addition, there are relationships among these entities. For instance, various pages that refer to one another, such as the link from url6 to url1, denote a particular relationship, in this case the relationship between the student hiroaki and his adviser stephen. This can again be modeled as facts in a relational database:

adviser(stephen, hiroaki) ←
teaches(john, logic_for_learning) ←
belongsTo(stephen, computer_science) ←
follows(hiroaki, logic_for_learning) ←
. . .

Again, the structure of the problem can elegantly be represented using a relational database. This representation can easily be extended with additional background knowledge in the form of rules:

studentOf(Lect, Stud) ← teaches(Lect, Course), follows(Stud, Course)

which expresses that

> Stud is a studentOf Lect IF Lect teaches a Course and Stud follows the Course.

Further information could be contained in the data set, such as extracts of the text appearing on the different links or pages. There are several interesting link mining tasks in this domain. It is for instance possible to learn to predict the classes of web pages or the nature of the relationships encoded by links between web pages; cf. [Getoor, 2003, Craven and Slattery, 2001, Chakrabarti, 2002, Baldi et al., 2003]. To address such tasks using logical or relational learning, one has to start from a set of examples of relations that are known to hold. For instance, the fact that stephen is the adviser of hiroaki is a (positive) example stating that the link from hiroaki to stephen belongs to the relation adviser. If for a given university website, say the University of Freiburg, all hyperlinks would be labeled (by hand) with the corresponding relationships, one could then learn general rules using logical and relational learning that would allow one to predict the labels of unseen hyperlinks. These rules could then be applied to determine the labels of the hyperlinks at another university website, say that of the University of Leuven. An example of a rule that might be discovered in this context is

$$\begin{aligned}&\mathsf{adviser}(\mathsf{Prof},\mathsf{Stud}) \leftarrow\\ &\qquad \mathsf{webpage}(\mathsf{Stud},\mathsf{Url}),\mathsf{student}(\mathsf{Url}),\\ &\qquad \mathsf{contains}(\mathsf{Url},\mathsf{advisor}),\mathsf{contains}(\mathsf{Url},\mathsf{Prof})\end{aligned}$$

which expresses that

> Prof is an adviser of Stud IF Stud has a webpage with Url of type student that contains the words adviser and Prof.

To tackle such problems, one often combines logical and relational learning with probabilistic models. This topic will be introduced in Chapter 8.

Link mining problems in general, and the above example in particular, cannot easily be represented using the single-tuple single-table assumption (as in Table 1.1) without losing information.

Exercise 1.3. Try to represent the link mining example within the single-tuple single-table assumption and identify the problems with this approach.

Exercise 1.4. Sketch other application domains that cannot be modeled under this assumption.

1.2.3 A Language Learning Example

A third illustration of an application domain that requires dealing with knowledge as well as structured data is natural language processing. Empirical natural language processing is now a major trend within the computational linguistics community [Manning and Schütze, 1999], and several logical and relational learning scientists have contributed interesting techniques and applications; cf. [Cussens and Džeroski, 2000, Mooney, 2000]. As an illustration of this line of work, let us look at one of the applications of Raymond Mooney's group, which pioneered the use of logical and relational learning techniques for language learning. More specifically, we sketch how to learn to parse database queries in natural language, closely following Zelle and Mooney [1996].[2] The induced semantic parser is the central component of a question-answering system.

Example 1.5. Assume you are given a relational database containing information about geography. The database contains information about various basic

[2] For ease of exposition, we use a slightly simplified notation.

entities such as countries, cities, states, rivers and places. In addition, it contains facts about the relationships among them, for instance, $\mathsf{capital}(\mathsf{C}, \mathsf{Y})$, which specifies that C is the capital of Y, $\mathsf{loc}(\mathsf{X}, \mathsf{Y})$, which states that X is located in Y, $\mathsf{nextTo}(\mathsf{X}, \mathsf{Y})$, which states that X is located next to Y, and many other relationships.

The task could then be to translate queries formulated in natural language to database queries that can be executed by the underlying database system. For instance, the query in natural language

What are the major cities in Kansas?

could be translated to the database query

$$\mathsf{answer}(\mathsf{C}, (\mathsf{major}(\mathsf{C}), \mathsf{city}(\mathsf{C}), \mathsf{loc}(\mathsf{C}, \mathsf{S}), \mathsf{equal}(\mathsf{S}, \mathsf{kansas})))$$

This last query can then be passed on to the database system and executed. The database system then generates all entities C that are major, a city, and $\mathsf{located}$ in kansas.

Zelle and Mooney's learning system [1996] starts from examples, which consist of queries in natural language and in database format, from an elementary shift-reduce parser, and from some background knowledge about the domain. The task is then to learn control knowledge for the parser. Essentially, the parser has to learn the conditions under which to apply the different operators in the shift-reduce parser. The control knowledge is represented using a set of clauses (IF rules) as in the previous two case studies. The control rules need to take into account knowledge about the stack used by the parser, the structure of the database, and the semantics of the language. This is again hard (if not impossible) to represent under the single-tuple single-table assumption.

1.3 How Does Relational and Logical Learning Work?

Symbolic machine learning and data mining techniques essentially search a space of possible patterns, models or regularities. Depending on the task, different search algorithms and principles apply. For instance, consider the structure activity relationship prediction task and assume that one is searching for *all* structural alerts that occur in at least 20% of the actives and at most 2% of the inactives. In this case, a complete search strategy is applicable. On the other hand, if one is looking for a structural alert that separates the actives from the inactives and could be used for classification, a heuristic search method such as hill climbing is more appropriate.

Data mining is often viewed as the process of computing the set of patterns $Th(Q, D, \mathcal{L})$ [Mannila and Toivonen, 1997], which can be defined as follows (cf. also Chapter 3). The search space consists of all patterns expressible within a language of patterns $\mathcal{L}$. For logical and relational learning this will typically be a set of rules or clauses of the type we encountered in the case studies of the previous section; the data set D consists of the examples that need to be generalized; and, finally, the constraint Q specifies which patterns are of interest. The constraint typically depends on the data mining task tackled, for instance, finding a single structural alert in a classification setting, or finding all alerts satisfying particular frequency thresholds. So, the set $Th(Q, D, \mathcal{L})$ can be defined as the set of all patterns $h \in \mathcal{L}$ that satisfy the constraint $Q(h, D)$ with respect to the data set D.

A slightly different perspective is given by the machine learning view, which is often formulated as that of finding a particular function h (again belonging to a language of possible functions $\mathcal{L}$) that minimizes a *loss* function $l(h, D)$ on the data. Using this view, the natural language application of the previous subsection can be modeled more easily, as the goal is to learn a function mapping statements in natural language to database queries. An adequate loss function is the accuracy of the function, that is, the fraction of database queries that is correctly predicted. The machine learning and data mining views can be reconciled, for instance, by requiring that the constraint $Q(h, D)$ succeeds only when $l(h, D)$ is minimal; cf. Chapter 3.

Central in the definition of the constraint $Q(h, D)$ or the loss function $l(h, D)$ is the *covers* relation between the data and the rules. It specifies when a rule covers an example, or, equivalently, when an example satisfies a particular rule. There are various possible ways to represent examples and rules and these result in different possible choices for the covers relation (cf. Chapter 4). The most popular choice is that of *learning from entailment*. It is also the setting employed in the case studies above.

Example 1.6. To illustrate the notion of coverage, let us reconsider Ex. 1.1 and let us also simplify it a bit. An example could now be represented by the rule:

$$\begin{array}{l}\mathsf{active(m1)} \leftarrow \\ \qquad \mathsf{atom(m1, m11, c)}, \ldots, \mathsf{atom(m1, m1n, c)}, \\ \qquad \mathsf{bond(m1, m11, m12, 2)}, \ldots, \mathsf{bond(m1, m11, m13, 1)}, \\ \qquad \mathsf{ring\_size\_5(m1, [m15, m11, m12, m13, m14])}, \ldots\end{array}$$

Consider now the rule

$$\mathsf{active(M)} \leftarrow \mathsf{ring\_size\_5(M, R), atom(M, M1, c)}$$

which actually states that

> Molecule M is active IF it contains a ring of size 5 called R and an atom M1 that is a carbon (c).

The rule covers the example because the conditions in the rule are satisfied by the example when setting $\mathsf{M} = \mathsf{m1}$, $\mathsf{R} = [\mathsf{m15}, \mathsf{m11}, \mathsf{m12}, \mathsf{m13}, \mathsf{m14}]$ and $\mathsf{M1} = \mathsf{m11}$. The reader familiar with logic (see also Chapter 2) will recognize that the example e is a logical consequence of the rule r, which is sometimes written as $r \models e$.

Now that we know *what* to compute, we can look at *how* this can be realized. The computation of the solutions proceeds typically by searching the space of possible patterns or hypotheses $\mathcal{L}$. One way of realizing this is to employ a generate-and-test algorithm, though this is too naive to be efficient. Therefore symbolic machine learning and data mining techniques typically structure the space $\mathcal{L}$ according to *generality.* One pattern or hypothesis is *more general* than another if all examples that are covered by the latter pattern are also covered by the former.

Example 1.7. For instance, the rule

$$\mathsf{active(M)} \leftarrow \mathsf{ring\_size\_5(M, R), element(A1, R), bond(M, A1, A2, 2)}$$

is more general than the rule

$$\begin{aligned}&\mathsf{active(M)} \leftarrow\\ &\qquad \mathsf{ring\_size\_5(M, R), element(A1, R),}\\ &\qquad \mathsf{bond(M, A1, A2, 2), atom(M, A2, o, 52, C)}\end{aligned}$$

which reads as

> Molecule M is active IF it contains a ring of size 5 called R and atoms A1 and A2 that are connected by a double (2) bond such that A1 also belongs to the ring R and atom A2 is an oxygen of type 52.

The former rule is more general (or, equivalently, the latter one is more specific) because the latter one requires also that the atom connected to the ring of size 5 be an oxygen of atom-type 52. Therefore, all molecules satisfying the latter rule will also satisfy the former one.

The generality relation is quite central during the search for solutions. The reason is that the generality relation can often be used 1) to prune the search space, and 2) to guide the search towards the more promising parts of the space. The generality relation is employed by the large majority of logical and relational learning systems, which often search the space in a *general-to-specific* fashion. This type of system starts from the most general rule (the unconditional rule, which states that *all* molecules are active in our running example), and then repeatedly specializes it using a so-called refinement operator. Refinement operators map rules onto a set of specializations; cf. Chapters 3 and 5.

Example 1.8. Consider the rule

$$\mathsf{active(Mol)} \leftarrow \mathsf{atom(Mol, Atom, c, Type, Charge)},$$

which states that a molecule is active if it contains a carbon atom. Refinements of this rule include:

$\mathsf{active(Mol)} \leftarrow \mathsf{atom(Mol, Atom, c, 21, Charge)}$
$\mathsf{active(Mol)} \leftarrow \mathsf{atom(Mol, Atom, c, T, Charge), atom(Mol, Atom2, h, T2, Charge2)}$
$\mathsf{active(Mol)} \leftarrow \mathsf{atom(Mol, Atom, c, T, Charge), ring\_size\_5(Mol, Ring)}$
...

The first refinement states that the carbon atom must be of type 21, the second one requires that there be carbon as well as hydrogen atoms, and the third one that there be a carbon atom and a ring of size 5. Many more specializations are possible, and, in general, the operator depends on the description language and generality relation used.

The generality relation can be used to prune the search. Indeed, assume that we are looking for rules that cover at least 20% of the active molecules and at most 1% of the inactive ones. If our current rule (say the second one in Ex. 1.7) only covers 18% of the actives, then we can prune away all specializations of that rule because specialization can only decrease the number of covered examples. Conversely, if our current rule covers 2% of the inactives, then all generalizations of the rule cover at least as many inactives (as generalization can only increase the number of covered examples), and therefore these generalizations can safely be pruned away; cf. Chapter 3 for more details.

Using logical description languages for learning provides us not only with a very expressive representation, but also with an excellent theoretical foundation for the field. This becomes clear when looking at the generality relation. It turns out that the generality relation coincides with logical entailment. Indeed, the above examples of the generality relation clearly show that the more general rule logically entails the more specific one.[3] So, the more specific rule is a logical consequence of the more general one, or, formulated differently, the more general rule logically entails the more specific one. Consider the simpler example: $\mathsf{flies(X)} \leftarrow \mathsf{bird(X)}$ (if X is a bird, then X flies), which logically entails and which is clearly more general than the rule $\mathsf{flies(X)} \leftarrow \mathsf{bird(X), normal(X)}$ (only normal birds fly). This property of the generalization relation provides us with an excellent formal basis for studying inference operators for learning. Indeed, because one rule is more general than another if the former entails the latter, deduction is closely related to specialization as deductive operators can be used as specialization operators. At the same time, as we will see in detail in Chapter 5, one can obtain generalization (or inductive inference) operators by inverting deductive inference operators.

1.4 A Brief History

Logical and relational learning typically employ a form of reasoning known as *inductive inference*. This form of reasoning generalizes specific facts into gen-

[3] This property holds when learning from entailment. In other settings, such as learning from interpretations, this property is reversed; cf. Chapter 5. The more specific hypothesis then entails the more general one.

eral laws. It is commonly applied within the natural sciences, and therefore has been studied in the philosophy of science by several philosophers since Aristotle. For instance, Francis Bacon investigated an inductive methodology for scientific inquiry. The idea is that knowledge can be obtained by careful experimenting, observing, generalizing and testing of hypotheses. This is also known as empiricism, and various aspects have been studied by many other philosophers including Hume, Mill, Peirce, Popper and Carnap. Inductive reasoning is fundamentally different from deductive reasoning in that the conclusions of inductive reasoning do not follow logically from their premises (the observations) but are always cogent; that is, they can only be true with a certain probability. The reader may notice that this method is actually very close in spirit to that of logical and relational learning today. The key difference seems to be that logical and relational learning investigates *computational* approaches to inductive reasoning.

Computational models of inductive reasoning and scientific discovery have been investigated since the very beginning of artificial intelligence. Several cognitive scientists, such as a team involving the Nobel prize winner Herbert A. Simon (see [Langley et al., 1987] for an overview), developed several models that explain how specific scientific theories could be obtained. Around the same time, other scientists (including Bruce Buchanan, Nobel prize winner Joshua Lederberg, Ed Feigenbaum, and Tom Mitchell [Buchanan and Mitchell, 1978]) started to develop learning systems that could assist scientists in discovering new scientific laws. Their system META-DENDRAL produced some new results in chemistry and were amongst the first scientific discoveries made by an artificial intelligence system that were published in the scientific literature of the application domain. These two lines of research have actually motivated many developments in logical and relational learning albeit there is also a crucial difference between them. Whereas the mentioned approaches were *domain-specific*, the goal of logical and relational learning is to develop *general-purpose* inductive reasoning systems that can be applied across different application domains. The example concerning structure activity relationship is a perfect illustration of the results of these developments in logical and relational learning. An interesting philosophical account of the relationship between these developments is given by Gillies [1996].

Supporting the scientific discovery process across different domains requires a solution to two important computational problems. First, as scientific theories are complex by their very nature, an expressive formalism is needed to represent them. Second, the inductive reasoning process should be able to employ the available background knowledge to obtain meaningful hypotheses. These two problems can to a large extent be solved by using logical representations for learning.

The insight that various types of logical and relational representations can be useful for inductive reasoning and machine learning can be considered as an outgrowth of two parallel developments in computer science and artificial intelligence. First, and most importantly, since the mid-1960s a number of researchers proposed to use (variants of) predicate logic as a formalism for studying machine learning problems. This was motivated by severe limitations of the early machine learning systems that essentially worked with propositional representations. Ranan Banerji [1964] was amongst the earliest advocates of the use of logic for machine learning. The logic he proposed was motivated by a pattern recognition task. Banerji's work on logical descriptions provided inspiration for developing logical learning systems such as CONFUCIUS [Cohen and Sammut, 1982] and MARVIN [Sammut and Banerji, 1986], which already incorporated the first inverse resolution operators; cf. Chapter 5. These systems learned incrementally and were able to employ the already learned concepts during further learning tasks.

Around the same time, Ryszard Michalski [1983] developed his influential AQ and INDUCE systems that address the traditional classification task that made machine learning so successful. Michalski's work stressed the importance of both learning readable descriptions and using background knowledge. He developed his own variant of a logical description language, the Variable Valued Logic, which is able to deal with structured data and relations. At the same time, within VVL, he suggested that induction be viewed as the inverse of deduction and proposed several inference rules for realizing this. This view can be traced back in the philosophy of science [Jevons, 1874] and still forms the basis for much of the theory of generalization, which is extensively discussed in Chapter 5.

Theoretical properties of generalization and specialization were also studied by researchers such as Plotkin [1970], Reynolds [1970], Vere [1975] and Buntine [1988]. Especially, Plotkin's Ph.D. work on θ-subsumption and relative subsumption, two generalization relations for clausal logic, has been very influential and still constitutes the main framework for generalization in logical learning. It will be extensively studied in Chapter 5. Second, there is the work on automatic programming [Biermann et al., 1984] that was concerned with synthesizing programs from examples of their input-output behavior, where researchers such as Biermann and Feldman [1972], Summers [1977] and Shapiro [1983] contributed very influential systems and approaches. Whereas Alan Biermann's work was concerned with synthesizing Turing machines, Phil Summers studied functional programs (LISP) and Ehud Shapiro studied the induction of logic programs and hence contributed an inductive logic programming system avant la lettre. Shapiro's MODEL INFERENCE SYSTEM is still one of the most powerful program synthesis and inductive inference systems today. It will be studied in Chapter 7.

In the mid-1980s, various researchers including Bergadano et al. [1988], Emde et al. [1983], Morik et al. [1993], Buntine [1987] and Ganascia and Kodratoff [1986] contributed inductive learning systems that used relational or logical description languages. Claude Sammut [1993] gives an interesting account of this period in logical and relational learning.

A breakthrough occurred when researchers started to realize that both problems (in automatic programming and machine learning) could be studied simultaneously within the framework of computational logic. It was the contribution of Stephen Muggleton [1991] to define the new research field of *inductive logic programming* (ILP) [Muggleton and De Raedt, 1994, Lavrač and Džeroski, 1994] as the intersection of inductive concept learning and logic programming and to bring together researchers in these areas in the inductive logic programming workshops [Muggleton, 1992b] that have been organized annually since 1991. A new subfield of machine learning was born and attracted many scientists, especially in Japan, Australia and Europe (where two European projects on inductive logic programming were quite influential [De Raedt, 1996]). Characteristic for the early 1990s was that inductive logic programming was developing firm theoretical foundations, built on logic programming concepts, for logical learning. In parallel, various well-known inductive logic programming systems were developed, including Foil [Quinlan, 1990], Golem [Muggleton and Feng, 1992], Progol [Muggleton, 1995], Claudien [De Raedt and Dehaspe, 1997], Mobal [Morik et al., 1993], Linus [Lavrač and Džeroski, 1994]. Also, the first successes in real-life applications of inductive logic programming were realized by Ross King, Stephen Muggleton, Ashwin Srinivasan, and Michael Sternberg [King et al., 1992, King and Srinivasan, 1996, King et al., 1995, Muggleton et al., 1992]; see [Džeroski, 2001] for an overview. Due to the success of these applications and the difficulties in true progress in program synthesis, the field soon focused on machine learning and data mining rather than on automatic programming. A wide variety of systems and techniques were being developed that *upgraded* traditional machine learning systems towards the use of logic; cf. Chapter 6.

During the mid-1990s, both the data mining and the uncertainty in artificial intelligence communities started to realize the limitations of the key representation formalism they were using. Within the data mining community, the item-sets representation employed in association rules [Agrawal et al., 1993] corresponds essentially to a boolean or propositional logic, and the Bayesian network formalism [Pearl, 1988] defines a probability distribution over propositional worlds. These limitations motivated researchers to look again at more expressive representations derived from relational or first-order logic. Indeed, within the data mining community, the work on Warmr [Dehaspe et al., 1998], which discovers frequent queries and relational association rules from a relational database, was quite influential. It was successfully applied on a structure activity relationship prediction task, and motivated several researchers to look into graph mining [Washio et al., 2005, Inokuchi et al., 2003, Washio and Motoda, 2003]. Researchers in data mining soon started to talk about *(multi-)relational data mining* (MRDM) (cf. [Džeroski and Lavrač, 2001]), and an annual series of workshops on multi-relational data mining was initiated [Džeroski et al., 2002, 2003, Džeroski and Blockeel, 2004, 2005].

A similar development took place in the uncertainty in artificial intelligence community. Researchers started to develop expressive probabilistic logics [Poole, 1993a, Breese et al., 1994, Haddawy, 1994, Muggleton, 1996], and started to study learning [Sato, 1995, Friedman et al., 1999] in these frameworks soon afterward. In the past few years, these researchers have gathered in the statistical relational learning (SRL) workshops [Getoor and Jensen, 2003, 2000, De Raedt et al., 2005, 2007a]. A detailed overview and introduction to this area is contained in Chapter 8 and two recent volumes in this area include [Getoor and Taskar, 2007, De Raedt et al., 2008]. Statistical relational learning is one of the most exciting and promising areas for relational and logical learning today.

To conclude, the field of logical and relational learning has a long history and is now being studied under different names: inductive logic programming, multi-relational data mining and (statistical) relational learning. The approach taken in this book is to stress the similarities between these trends rather than the differences because the problems studied are essentially the same even though the formalisms employed may be different. The author hopes that this may contribute to a better understanding of this exciting field.

2

An Introduction to Logic

The data mining and machine learning approaches discussed in this book employ relational or logical representations. This chapter introduces relational database representations and their formalization in logic. Logic is employed because it is an expressive, well-understood and elegant formalism for representing knowledge. We focus on definite clause logic, which forms the basis of computational logic and logic programming. The chapter introduces the syntax as well as the model-theoretic and proof-theoretic semantics of definite clause logic. Throughout this book an attempt is made to introduce all the necessary concepts and terminology in an intuitive and yet precise manner, without resorting to unnecessary formal proof or theory. The reader interested in a more detailed theoretical account of clausal logic may want to consult [Lloyd, 1987, Genesereth and Nilsson, 1987, Flach, 1994, Hogger, 1990, Kowalski, 1979], and for a more detailed introduction to the programming language Prolog we refer him to [Flach, 1994, Bratko, 1990, Sterling and Shapiro, 1986].

2.1 A Relational Database Example

This book is concerned with mining relational data. The data to be mined will typically reside in a relational database. An example relational database, written in the logical notation employed throughout this book, is given in the following example.

Example 2.1. The database contains information about publications, authors and citations. The entry $\mathsf{authorOf(lloyd, logic\_for\_learning)} \leftarrow$ for the relation $\mathsf{authorOf/2}$ represents the fact that lloyd is the author of the publication $\mathsf{logic\_for\_learning}$. Similarly, $\mathsf{reference(logic\_for\_learning, foundations\_of\_lp)} \leftarrow$ is a fact stating that $\mathsf{foundations\_of\_lp}$ is in the bibliography of $\mathsf{logic\_for\_learning}$.

```
reference(logic_for_learning, foundations_of_lp) ←
reference(logic_for_learning, learning_logical_definitions_from_relations) ←
reference(logic_for_learning, ai_a_modern_approach) ←
reference(ai_a_modern_approach, foundations_of_lp) ←
reference(ai_a_modern_approach, ilp_theory_and_methods) ←
reference(ilp_theory_and_methods, learning_logical_definitions_from_relations) ←
reference(ilp_theory_and_methods, foundations_of_lp) ←

authorOf(quinlan, learning_logical_definitions_from_relations) ←
authorOf(lloyd, foundations_of_lp) ←
authorOf(lloyd, logic_for_learning) ←
authorOf(russell, ai_a_modern_approach) ←
authorOf(norvig, ai_a_modern_approach) ←
authorOf(muggleton, ilp_theory_and_methods) ←
authorOf(deraedt, ilp_theory_and_methods) ←
```

Even though the database contains only a small extract of a real database, it will be used to introduce some of the key concepts of the knowledge representation formalism employed in this book. The formalism employed is that of clausal logic because it is an expressive representation formalism, it is well-understood and it provides us with the necessary theory and tools for relational learning. At the same time, it is closely related to relational database formalisms. This is illustrated by the bibliographic database just introduced and will become more apparent throughout this section (and Sect. 4.4).

In logic, a relation is called a *predicate* p/n (e.g., authorOf/2) where p denotes the name of the predicate (authorOf) and n the arity (2), which indicates the number of arguments the predicate takes. The arguments of predicates are *terms*. *Constants* are a particular type of term that start with a lowercase character, for instance, quinlan and logic_for_learning. They refer to a particular object in the domain of discourse, such as the book "Logic for Learning" [Lloyd, 2003] or the author "J.R. Quinlan". All entries in Ex. 2.1 are *facts*, which are expressions of the form $p(t_1, \cdots, t_n) \leftarrow$, where p/n is a predicate and the t_i are terms. Facts correspond to tuples in a relational database. They express unconditional truths. For instance, the fact authorOf(lloyd, logic_for_learning) ← expresses that lloyd is the author of the publication logic_for_learning. As in relational databases it is possible to query for information that resides in the database.

Example 2.2. The query ← authorOf(lloyd, logic_for_learning) asks whether lloyd is the author of logic_for_learning. A theorem prover (such as Prolog) would return the answer yes, implying that the fact is true, that is, that it logically follows from the specified database.

The query ← authorOf(lloyd, Pub) asks whether there is a publication authored by lloyd. The term Pub is called a *variable*, and by convention, variables start with an uppercase character. The theorem prover would answer yes and would also return the *substitutions* {Pub/logic_for_learning} and

{Pub/foundations_of_lp}, which state the conditions under which the original query is true. For those readers familiar with SQL, a popular database language, we also specify the corresponding query in SQL[1]:

```
SELECT Pub FROM authorOf WHERE Author = lloyd
```

The query

$$\leftarrow \mathsf{authorOf}(\mathsf{Author}, \mathsf{logic\_for\_learning}), \mathsf{authorOf}(\mathsf{Author}, \mathsf{foundations\_of\_lp})$$

asks whether there is an author who wrote logic_for_learning as well as foundations_of_lp. The single substitution that would be returned for this query is {Author/lloyd}. In SQL, this query is written as:

```
SELECT Author
FROM authorOf t1, authorOf t2
WHERE t1.Author = t2.Author AND t1.Pub = logic_for_learning
        and t2.Pub = foundations_of_lp
```

So, the "," between the two atoms in the query corresponds to *and*, and two occurrences of the same variable must be instantiated to the same constant in the substitution, which corresponds to a "join" in relational databases.

Suppose now that instead of being interested in knowing which publications refer to which other publications, we are interested in knowing the authors who cite one another. They can be listed using the following query:

$$\leftarrow \mathsf{authorOf}(\mathsf{A}, \mathsf{P}), \mathsf{reference}(\mathsf{P}, \mathsf{Q}), \mathsf{authorOf}(\mathsf{B}, \mathsf{Q})$$

Many answers would be generated for this query, including:

{A/lloyd, P/logic_for_learning, Q/foundations_of_lp, B/lloyd}
{A/lloyd, P/logic_for_learning, Q/ai_a_modern_appraoch, B/russell}
...

If the query is posed a number of times, it is useful to define a new predicate cites/2 using the clause:

$$\mathsf{cites}(\mathsf{A}, \mathsf{B}) \leftarrow \mathsf{authorOf}(\mathsf{A}, \mathsf{P}), \mathsf{reference}(\mathsf{P}, \mathsf{Q}), \mathsf{authorOf}(\mathsf{B}, \mathsf{Q})$$

This clause states that A cites B if A is the authorOf P, P references Q, and B is the authorOf Q. This clause corresponds to the definition of a *view* or an *intensional* relation in relational database terminology, which can be implemented by the following statement in SQL.

```
CREATE VIEW cites
AS SELECT a1.Author, a2.Author
FROM authorOf a1, authorOf a2, reference r
WHERE a1.Pub = r.Pub1 AND a2.Pub = r.Pub2
```

[1] The reader unfamiliar with SQL can safely skip the SQL queries.

The effect is that the predicate cites can now be queried as if it contained the following facts:

cites(lloyd, lloyd) ←
cites(lloyd, russell) ←
cites(russell, lloyd) ←
cites(russell, muggleton) ←
cites(russell, deraedt) ←
cites(muggleton, quinlan) ←
cites(muggleton, lloyd) ←
cites(lloyd, quinlan) ←
cites(lloyd, norvig) ←
cites(norvig, lloyd) ←
cites(norvig, muggleton) ←
cites(norvig, deraedt) ←
cites(deraedt, quinlan) ←
cites(deraedt, lloyd) ←

Exercise 2.3. Pose a query to identify authors who cite themselves. Use both the original database (consisting of reference/2 and authorOf/2) and the one where the cites predicate has been defined. Define also a new predicate self_citation/1 that succeeds for those authors who cite themselves.

2.2 The Syntax of Clausal Logic

Whereas the previous section introduced some logical concepts in a rather informal manner, the present section will introduce them in a more formal and precise manner.

A *term* t is a constant, a variable or a compound term $f(t_1, ..., t_n)$ composed of a function symbol f/n (where f is the name of the function and n is its arity) and n terms t_i. Simple terms are, for instance, logic_for_learning (a constant) and Pub (a variable). We will use the convention that constants and function symbols start with a lowercase character and variables start with an uppercase character.

Compound terms were so far not introduced as they do not belong to the relational subset of clausal logic. They are used to represent structured objects. Examples of compound terms include card(j, hearts), where card/2 is a function symbol, and j and hearts are constants. It can be used to denote the Jack of Hearts card. Compound terms can also be nested. For instance, one can denote the father of John using the term fatherOf(john). The grandfather of John would then be denoted as fatherOf(fatherOf(john)), that is, as the father of the father of John. This notation is also used to represent the natural numbers using logic: the number 0 would be represented by 0, the number 1 by succ(0), the successor of 0, the number 2 by succ(succ(0)), etc. Lists are represented using the functor cons/2 (in Prolog the functor is often represented as ·/2). The first argument of cons/2 denotes the first element of the list, the second argument denotes the rest of the list, and the constant nil is used to denote the empty list. For instance, the list [1, 2] is represented as cons(1, cons(2, nil)).[2]

[2] The list functor cons/2 is usually written in Prolog in infix notation. Using this notation cons(A, B) is written as [A | B], and cons(A, cons(B, C)) as [A | [B | C]] or [A, B | C] for short.

An *atom* is a formula of the form $p(t_1, .., t_n)$, where p/n is a predicate symbol and the t_i are terms. For instance, $\mathsf{authorOf(lloyd, logic\_for\_learning)}$, $\mathsf{largerRank(ace, X)}$, $\mathsf{faceCard(j, hearts)}$, and $\mathsf{pair(card(j, hearts), card(j, diamonds))}$ are atoms. Although terms and atoms possess a similar syntax, their meaning is quite different. These differences are clearest when looking at ground terms and atoms: ground terms represent objects in the domain of discourse, whereas ground atoms represent a particular relationship among the objects denoted by the terms appearing in the atom. Therefore, ground atoms possess a truth-value, that is, they are either true or false, whereas ground terms (and objects) do not posses truth-values. Indeed, it does not make sense to talk about the truth-value of the constant green, the persons "Mary Ann' or $\mathsf{fatherOf(john)}$, or the list $\mathsf{con(a, cons(b, nil))}$. However, the atom $\mathsf{parent(fatherOf(john), john)}$ (for the predicate $\mathsf{parent/2}$) would typically be true.

By now, we are able to define clauses, which are the key constructs in clausal logic. Clausal logic is a subset of first-order logic that forms the basis of logic programming and the programming language Prolog. It is frequently employed within artificial intelligence due to its uniform and simple representation, and the existence of efficient inference engines or theorem provers for clausal logic.

Definition 2.4. *A* clause *is an expression of the form* $h_1; \cdots ; h_n \leftarrow b_1, \cdots , b_m$ *where the* h_i *and* b_j *are logical atoms.*

The symbol "," stands for conjunction (and), the symbol ";" for disjunction (or), and "←" for implication (if). Furthermore, all variables are universally quantified (though this is not explicitly written). So, the clause

$$\mathsf{female(X); male(X)} \leftarrow \mathsf{human(X)}$$

specifies that *for all* X, when X is human, X is also male or female. $h_1; \cdots ; h_n$ is referred to as the *conclusion* part of the clause, or the *head* of the clause, $b_1, \cdots , b_m$ as the *condition* part or the *body* of the clause. It will often be convenient to employ the notation $body(c)$ for $\{b_1, \cdots , b_m\}$ and $head(c)$ for $\{h_1, \cdots , h_n\}$, where c is the clause $h_1; \cdots ; h_n \leftarrow b_1, \cdots , b_m$. This set notation can be extended to the overall clause. For instance, the above clause c is represented by the set $\{h_1, \cdots , h_n, \neg b_1, \cdots , \neg b_m\}$ of the literals it contains, where a *literal* is either a logical atom a or a negated atom $\neg a$. The set notation for clauses reflects also that a clause is a disjunction $h_1 \vee \cdots \vee h_n \vee \neg b_1 \vee \cdots \vee \neg b_m$ of literals.

There exist several special types of clauses:

- *facts*, where $n = 1$ and $m = 0$,
- *definite clauses*, where $n = 1$,
- *Horn clauses*, where $n = 1$ or $n = 0$, and
- *denials*, where $n = 0$.

The clauses that are most often employed in logical and relational learning are facts and definite clauses, which we already encountered in the previous section. Denials represent negative information, for instance, $\leftarrow$ human(dracula) specifies that dracula is *not* human, that is, that human(dracula) is false. Denials are used as queries or *goals*. The reasons for this will become clear soon.

Typically, the database consists of a set of clauses. A set of clauses $\{c_1, \cdots, c_n\}$ specifies that all clauses in the set are true, that is, the set denotes a *conjunction* $c_1, \cdots, c_n$ of the clauses c_i. We will sometimes refer to such sets of clauses as (clausal) *theories*, *databases*, or *knowledge bases*.

A *substitution* $\theta = \{V_1/t_1, \cdots, V_n/t_n\}$ is an assignment of terms $t_1, ..., t_n$ to variables $V_1, ..., V_n$, for instance, {A/lloyd, Pub/logic_for_learning}. The *instantiated* formula $F\theta$,where F is a term, atom, or clause and $\theta = \{V_1/t_1, \cdots, V_n/t_n\}$ a substitution, is the formula obtained by simultaneously replacing all variables $V_1, ..., V_n$ in F by the terms $t_1, ..., t_n$. For instance, the formula card(X, Y)θ, where θ = { X/j, Y/diamonds}, denotes card(j, diamonds).

2.3 The Semantics of Clausal Logic — Model Theory

Now that the syntax of clausal logic has been defined, its semantics can be introduced.

The semantics of logic are based on the concept of an *interpretation*. Interpretations are defined using the notion of a *domain*, which contains the set of objects that exist (in the interpretation). Roughly speaking, an *interpretation* of a set of clauses is an assignment that maps

- constants onto objects in the domain,
- function symbols f/n onto n-ary functions defined on the domain; these functions map an n-tuple of objects onto an object in the domain,
- predicates p/n onto n-ary relations defined on the domain; these relations represent a set of n-tuples of objects.

For instance, consider the constant john. It could represent a particular person named John Doe Jr. The function symbol fatherOf/1 could then represent the function that maps every person to his or her father. It could map John Doe Jr. onto John Doe Sr. The predicate parent/2 could then map to the parent relationship. These mappings then define the truth-values of atoms in the interpretation. For instance, the atom parent(fatherOf(john), john) would map to true, because John Doe Sr. (fatherOf(john)) is a parent of John Doe Jr. (john) in our interpretation, assuming that the tuple (John Doe Sr., John Doe Jr.) is in the relation denoted by parent.

Because it is inconvenient and complicated to work with general interpretations, and because when working with *clausal* logic, this is not really needed, we will restrict our attention to a special class of interpretations named after the French logician Jacques Herbrand. A *Herbrand interpretation* uses the

Herbrand domain as its domain. The *Herbrand domain* is defined as the set of all ground terms that can be constructed using the constants and function symbols that occur in the set of clauses considered.

Example 2.5. The Herbrand universe in the bibliographic database of Ex. 2.1 consists of all constants appearing in the facts, i.e.,

$$\{\mathsf{quinlan}, \mathsf{lloyd}, \mathsf{muggleton}, \ldots, \mathsf{logic\_for\_learning}, \ldots, \mathsf{ilp\_theory\_and\_methods}\}.$$

Example 2.6. Now consider the clauses defining the natural numbers:

$$\begin{aligned}&\mathsf{nat(0)} \leftarrow\\ &\mathsf{nat(succ(X))} \leftarrow \mathsf{nat(X)}\end{aligned}$$

The first clause states that $\mathsf{0}$ is a natural number; the second one that $\mathsf{succ(X)}$ is a natural number if X is one. The Herbrand universe in this case is the infinite set $\{\mathsf{0}, \mathsf{succ(0)}, \mathsf{succ(succ(0))}, \ldots\}$.

A Herbrand interpretation now maps each ground term to itself. So each ground term refers to itself; for instance, john now refers to the object john instead of to some person such as John Doe. In a similar manner, predicates are mapped onto relations over the Herbrand universe. The result is that a Herbrand interpretation can be viewed as the set of ground atoms that are true in the interpretation. This is the view that we will be employing throughout this book when talking about interpretations.

Definition 2.7. *A* Herbrand *interpretation of a set of clauses is a set of ground atoms (over the constant, function and predicate symbols occurring in the set of clauses).*

All ground atoms in the interpretation are assumed to be true, and all others are assumed to be false.

Example 2.8. One possible Herbrand interpretation I_1 over the bibliographic database of Ex. 2.1 is:

$$\{\mathsf{authorOf(russell, logic\_for\_learning)}, \mathsf{authorOf(russell, quinlan)}\}$$

The interpretation I_2 consists of all the ground atoms occurring in the bibliographic database.

Example 2.9. Similarly, for the above specified natural numbers, I_3, I_4, and I_5 defined below are Herbrand interpretations.

$$\begin{aligned}I_3 &= \{\}\\ I_4 &= \{\mathsf{nat(0)}, \mathsf{nat(succ(0))}\}\\ I_5 &= \{\mathsf{nat(0)}, \mathsf{nat(succ(0))}, \mathsf{nat(succ(succ(0)))}, \ldots\}.\end{aligned}$$

Some interpretations do not really reflect the properties of the clauses that we have written down. For instance, the first interpretations in both the bibliographic database and the definition of the natural numbers are intuitively impossible given the clauses that we have written down. This motivates the following definition.

Definition 2.10. *A Herbrand interpretation I is a* Herbrand model *for a set of clauses C if and only if for all clauses $h_1; \cdots ; h_n \leftarrow b_1, \cdots , b_m \in C$ and for all ground substitutions θ: $\{b_1\theta, \cdots , b_m\theta\} \subseteq I \rightarrow \{h_1\theta, \cdots , h_n\theta\} \cap I \neq \emptyset$.*

So, a Herbrand interpretation I is a model for a clause c if for all substitutions θ for which $body(c)\theta$ is true in I, $head(c)\theta$ is also true in I. If an interpretation is a model for a clause (or set of clauses), the interpretation *satisfies* the clause (or set of clauses); otherwise, the interpretation *violates* the clause (or set of clauses). Clausal theories that possess a Herbrand model are called *satisifable*; those that do not possess one are called *unsatisfiable.*
When the context is clear, we will talk about interpretations and models rather than Herbrand interpretations and Herbrand models.

Example 2.11. Reconsider our bibliographic database and the interpretation I_1. I_1 is not a model for the database because there exists a clause f

$$\mathsf{authorOf(russell, ai\_a\_modern\_approach)} \leftarrow$$

such that $body(f) = \{\} \subseteq I_1$ but

$$head(f) = \{\mathsf{authorOf(russell, ai\_a\_modern\_approach))}\} \cap I_1 = \emptyset.$$

At the same time, it is easy to see that I_2 is a model for the bibliographic database.

Example 2.12. For the natural numbers, I_3 is not a model, because of the fact $\mathsf{nat(0)} \leftarrow$. Neither is I_4, because of the recursive clause c for which there exists a substitution $\theta =\{\mathsf{X/succ(0)}\}$ for which $body(c)\theta = \{\mathsf{nat(succ(0))}\}\subseteq I_4$ but $head(c)\theta =\{\mathsf{nat(succ(succ(0)))}\}\cap I_4 = \emptyset$. Finally, I_5 is a model for the two clauses defining $\mathsf{nat/1}$.

Model theory is the basis for reasoning about the declarative semantics of logical formulae, which relies on the notion of *logical entailment.* Logical entailment defines when one formula is a consequence of, or follows from, another one.

Definition 2.13. *Let C be a set of clauses and c be a clause. C* logically entails *c, notation $C \models c$, if and only if all models of C are also models of c.*

In this definition, not only the Herbrand interpretations (and models) but *all* interpretations and models are considered. The following theorems, however, allow us to focus our attention on Herbrand interpretations and models to reason about the satisfiability of a particular formula.

Theorem 2.14. *(Proposition 3.30 from [Nienhuys-Cheng and de Wolf, 1997]) Let T be a set of clauses. Then T has a model if and only if T has a Herbrand model.*

To generalize the applicability of this result to entailment between two formulae C and c, the theorem below can be used.

Theorem 2.15. *(This follows from Proposition 3.31 [Nienhuys-Cheng and de Wolf, 1997]) Let C be a set of clauses and c be a clause. Then $C \models c$ if and only if $C \land \neg c$ is unsatisfiable, that is, if $C \land \neg c \models \Box$.*

The empty clause $\leftarrow$, sometimes written as $\Box$, is the clause whose head and body are both empty. This clause is unsatisfiable because its body is always true and its head is never true, regardless of the interpretation.

Example 2.16. Reconsider our bibliographic database B consisting of all facts listed in Ex. 2.1. Then

$$B \models \mathsf{authorOf(lloyd, logic\_for\_learning)} \leftarrow$$

because

$$B \land \neg\ \mathsf{authorOf(lloyd, logic\_for\_learning)}$$

is unsatisfiable as $\mathsf{authorOf(lloyd, logic\_for\_learning)}$ and its negation are contradictory.

Example 2.17. Similarly, let N consist of the two clauses defining $\mathsf{nat/1}$. Then $N \models \mathsf{nat(0)}$ and also $N \models \mathsf{nat(succ(0))}$.

An algorithm to generate a (Herbrand) model of a set of clauses is listed in Algo. 2.1. It assumes that all clauses are *range-restricted*, i.e., that all variables appearing in the head of a clause also appear in its body. Range-restrictedness implies that all facts are ground.

Algorithm 2.1 Generating a model of a set of clauses

```
$M := \emptyset$
while $M$ is not a model of $C$ do
  if there is a denial $\leftarrow b_1, \cdots, b_m$ in $C$ that is violated by $M$ then
    backtrack
  end if
  select $h_1; \cdots, h_n \leftarrow b_1, \cdots, b_m$ from $C$ and $\theta$ (choice point) such that
          $\{b_1\theta, \cdots, b_m\theta\} \subseteq M$, and
          $\{h_1\theta, \cdots, h_n\theta\} \cap M = \emptyset$
  add one of the $h_i\theta$ to $M$ (choice point)
end while
```

The algorithm starts with the empty model and repeatedly expands it by non-deterministically adding facts belonging to the head of a violated clause.

This process continues until M either becomes a model of the theory, or until a denial is violated. When a denial is violated, the model cannot be expanded by adding facts, which explains why the algorithm then backtracks to earlier choice points. Observe that the algorithm can fail when no model exists, and also that it can loop infinitely while attempting to construct an infinite model. There exists also a very elegant and short implementation of this algorithm in Prolog. It is called SATCHMO and was published by Manthey and Bry [1987]; it is described in detail in [Flach, 1994].

Example 2.18. Consider the following clausal theory:

$$\begin{array}{l}
\mathsf{human(X)} \leftarrow \mathsf{male(X)} \\
\mathsf{human(X)} \leftarrow \mathsf{female(X)} \\
\mathsf{female(X); male(X)} \leftarrow \mathsf{human(X)} \\
\mathsf{human(john)} \leftarrow
\end{array}$$

One possible trace of Algo. 2.1 generates the following sequence of interpretations:

$$\begin{array}{l}
\{\} \\
\{\mathsf{human(john)}\} \text{ using the last clause.} \\
\{\mathsf{human(john), female(john)}\} \text{ using the third clause.}
\end{array}$$

Another model for this example is $\{\mathsf{human(john), male(john)}\}$.

Example 2.19. Reconsider the bibliographic database together with the clause defining the predicate cites/2. Algo. 2.1 would generate the interpretation consisting of all ground facts for the predicates reference/2, cites/2 and authorOf/2 that were listed in the earlier examples.

Example 2.20. Reconsider the definition of the natural numbers using the predicate nat/1. For these clauses, the algorithm does not terminate, because it attempts to generate the infinite model I_5 defined above.

Exercise 2.21. Does the following clausal theory have a model? If so, generate one.

$$\begin{array}{l}
\leftarrow \mathsf{student(X), vampire(X)} \\
\leftarrow \mathsf{student(X), professor(X)} \\
\leftarrow \mathsf{female(X), male(X)} \\
\mathsf{being(dracula)} \leftarrow \\
\mathsf{clever(X); student(X)} \leftarrow \mathsf{being(X)} \\
\mathsf{female(X); male(X); vampire(X)} \leftarrow \mathsf{being(X)} \\
\mathsf{student(X); professor(X); vampire(X)} \leftarrow \mathsf{being(X)}
\end{array}$$

Exercise 2.22. Use the theorem prover SATCHMO implemented in Prolog (cf. [Manthey and Bry, 1987] or [Flach, 1994]) to generate more models of the theory in the previous example. (This exercise requires that you have access to a Prolog implementation. Some excellent Prolog implementations, such as SWI-Prolog and YAP-Prolog, are available from the public domain.)

Some theories (sets of clauses) have multiple models. Furthermore, one model can be a subset of another model, which motivates the notion of a *minimal model.* A Herbrand model is *minimal* if and only if none of its subsets is a model.

Example 2.23. Reconsider the theory listed in Ex. 2.18. The two models listed there are minimal. However, the model $\{\mathsf{human(john)}, \mathsf{male(john)}, \mathsf{female(john)}\}$ is not minimal and this interpretation would not be a model if the clause $\leftarrow \mathsf{female(X)}, \mathsf{male(X)}$ belonged to the theory.

When restricting one's attention to definite clauses, which is the subset of clausal logic most often applied in logic programming, the *minimal* model is unique, which explains why it is called the *least Herbrand model.* The least Herbrand model of a set of clauses C will be denoted as $M(C)$. The least Herbrand model is an important concept because it captures the semantics of the set of clauses. It consists of all ground atoms that are logically entailed by the definite clause theory. This can be formally specified as:

Property 2.24. Let C be a set of definite clauses and f be a ground fact. Then $C \models f$ if and only if $f \in M(C)$.

The least Herbrand model also captures the intuitive meaning of the theory. This is best illustrated by the bibliographic database, where the least Herbrand model is that sketched in Ex. 2.19, which indeed contains exactly those facts which would be generated using the SQL statement, and which one would intuitively expect.

When applied to (range-restricted) definite clause theories, Algo. 2.1 will indeed generate the least Herbrand model of the theory. However, in this case one can also use the simplified version sketched in Algo. 2.2.

Algorithm 2.2 Computing the least Herbrand model of a definite clause theory

```
$M_0 := \emptyset$
$M_1 := \{f | f \leftarrow \text{is a fact in } C\}$
$i := 1$
while $M_i \neq M_{i-1}$ do
  $M_{i+1} := \emptyset$
  for all $h \leftarrow b_1, \cdots, b_n \in C$ do
    for all $\theta$ such that $\{b_1\theta, \cdots, b_n\theta\} \subseteq M_i$ do
      add $h\theta$ to $M_{i+1}$
    end for
  end for
  $M_{i+1} := M_i \cup M_{i+1}$
  $i := i + 1$
end while
```

The key difference between this algorithm and the previous one is that it processes all violated clauses in *parallel* to generate the next model. It can be optimized by adding $\{b_1\theta, \cdots, b_n\theta\} \cap (M_i - M_{i-1}) \neq \emptyset$ as an additional constraint in the inner for loop. This way the same conclusions $h\theta$ will not be regenerated in every iteration, which will remove some redundant computations.

Example 2.25. Suppose the definite clause theory is:

$$\begin{array}{l}
\mathsf{ancestor(X,Y)} \leftarrow \mathsf{parent(X,Y)} \\
\mathsf{ancestor(X,Y)} \leftarrow \mathsf{parent(X,Z), ancestor(Z,Y)} \\
\\
\mathsf{parent(rose,luc)} \leftarrow \\
\mathsf{parent(leo,rose)} \leftarrow
\end{array}$$

The algorithm then computes the following sequence of models:

$$\begin{array}{l}
M_0 = \emptyset \\
M_1 = \{\mathsf{parent(rose,luc), parent(leo,rose)}\} \\
M_2 = M_1 \cup \{\mathsf{ancestor(rose,luc), ancestor(leo,rose)}\} \\
M_3 = M_2 \cup \{\mathsf{ancestor(leo,luc)}\} \\
M_4 = M_3
\end{array}$$

Exercise 2.26. Specify the least Herbrand model of the following theory:

$$\begin{array}{l}
\mathsf{plus(0,X,X)} \leftarrow \mathsf{nat(X)} \\
\mathsf{plus(succ(X),Y,succ(Z))} \leftarrow \mathsf{plus(X,Y,Z)} \\
\\
\mathsf{nat(0)} \leftarrow \\
\mathsf{nat(succ(X))} \leftarrow \mathsf{nat(X)}
\end{array}$$

2.4 Inference with Clausal Logic — Proof Theory

Now that the semantics of clausal logic has been defined, we discuss how to perform inference in clausal logic. Deductive inference is concerned with deciding whether one formula F logically entails another one G, that is, deciding whether $F \models G$. When working with clausal logic, the typical inference procedure is based on the deductive inference rule known as resolution. The resolution principle was introduced by Robinson [1965]. It is employed by the large majority of theorem provers for first-order logic. Even though many variants of the resolution principle exist, we will restrict our attention to resolution for Horn clauses and SLD-resolution, because this form of resolution underlies the programming language Prolog on which the vast majority of logical learning approaches is based.

Let us start by introducing resolution for propositional logic. In propositional logic, all predicates have arity 0, which implies that there are no constants, variables or structured terms that need to be taken into account.

Given the clauses

$$l \leftarrow b_1, \cdots, b_n \text{ and } h \leftarrow c_1, \cdots, c_{i-1}, l, c_i, \cdots, c_m$$

the *propositional resolution* operator infers the *resolvent*

$$h \leftarrow c_1, \cdots, c_{i-1}, b_1, \cdots, b_n, c_i, \cdots, c_m \tag{2.1}$$

The rule is sometimes displayed as

$$\frac{l \leftarrow b_1, \cdots, b_n \text{ and } h \leftarrow c_1, \cdots, c_{i-1}, l, c_i, \cdots, c_m}{h \leftarrow c_1, \cdots, c_{i-1}, b_1, \cdots, b_n, c_i, \cdots, c_m} \tag{2.2}$$

which indicates that if the clauses above the line are presented, the ones below the line may be inferred. We sometimes use the notation

$$\begin{gathered} l \leftarrow b_1, \cdots, b_n \text{ and } h \leftarrow c_1, \cdots, c_{i-1}, l, c_i, \cdots, c_m \\ \vdash_{res} \\ h \leftarrow c_1, \cdots, c_{i-1}, b_1, \cdots, b_n, c_i, \cdots, c_m \end{gathered} \tag{2.3}$$

to denote a particular resolution step.

This inference step is graphically illustrated in Fig. 2.1, where the operator takes the typical V form. Notice that this rule also holds when $h = \{\}$, that is, when working with a denial instead of a definite clause.

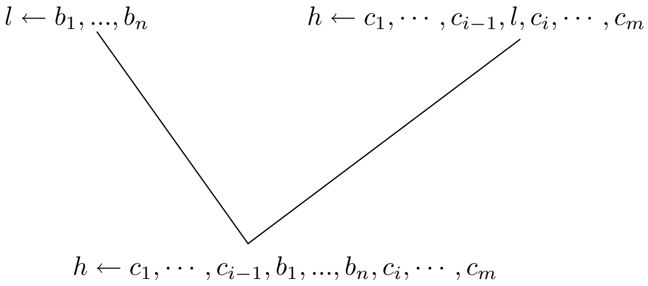

$l \leftarrow b_1, ..., b_n$
$h \leftarrow c_1, \cdots, c_{i-1}, l, c_i, \cdots, c_m$
$h \leftarrow c_1, \cdots, c_{i-1}, b_1, ..., b_n, c_i, \cdots, c_m$

Fig. 2.1. The propositional resolution operator

Example 2.27.

$$\begin{gathered} \mathsf{mammal} \leftarrow \mathsf{rabbit} \\ \mathsf{animal} \leftarrow \mathsf{mammal} \\ \\ \vdash_{res} \\ \\ \mathsf{animal} \leftarrow \mathsf{rabbit} \end{gathered}$$

The resolution operator is *sound*, which means that whenever $c_1 \wedge c_2 \vdash_{res} c$ holds, $c_1 \wedge c_2 \models c$ holds as well.

When more than one resolution step is performed, one talks about *resolution derivations* or *proofs*. A resolution proof of a clause c from a theory $T = \{c_1, ..., c_n\}$, notation $T \vdash c$, is a sequence of resolution steps $c_{1,1} \wedge c_{1,2} \vdash_{res} c_1, \cdots, c_{k,1} \wedge c_{k,2} \vdash_{res} c_k$ where $c_k = c$ and $c_{i,j} \in T \cup \{c_1, \cdots, c_{i-1}\}$. Resolution proofs can be graphically represented as trees; see Fig. 2.2 for an example.

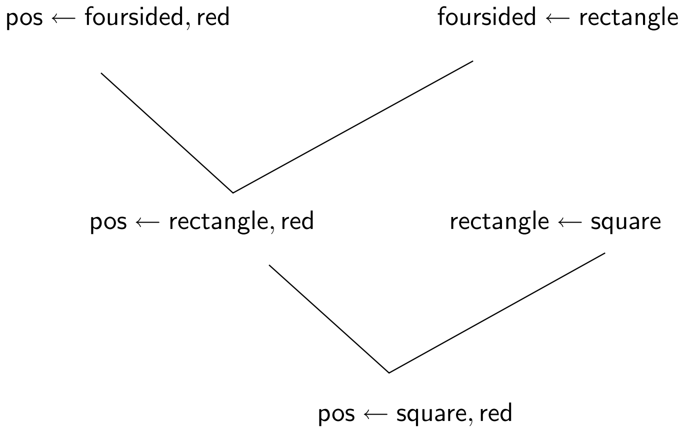


Fig. 2.2. A resolution derivation

Example 2.28. A resolution proof of the clause $\mathsf{pos} \leftarrow \mathsf{square}, \mathsf{red}$ (red squares are positive) is shown in Fig. 2.2. It assumes the following theory T is given:

$$\begin{array}{l} \mathsf{foursided} \leftarrow \mathsf{rectangle} \\ \mathsf{rectangle} \leftarrow \mathsf{square} \\ \mathsf{pos} \leftarrow \mathsf{foursided}, \mathsf{red} \end{array}$$

A popular technique in mathematics when proving theorems is to assume that a theorem is false and to prove that this leads to a contradiction. This technique also applies to logical inference and theorem proving, where it is known as proving by *refutation*. The idea of refutation was already formulated in Theorem 2.15, where it was stated that $C \models c$ if and only if $C \wedge \neg c \models \square$. A proof by refutation is now a resolution derivation that ends in the empty clause $\square$ or $\leftarrow$, which is unsatisfiable.

Resolution is not a *complete* operator for definite clause logic (even in the propositional case). Completeness would require that whenever $C \models c$ there also exists a resolution derivation of c from C.

Example 2.29. Indeed,

$$\mathsf{mammal} \leftarrow \mathsf{rabbit} \models \mathsf{mammal} \leftarrow \mathsf{rabbit}, \mathsf{brown}$$

but it is impossible to derive the second clause by resolution from the first one.

Fortunately, resolution is *refutation complete*, as indicated in the following property.

Property 2.30. (Refutation completeness; cf. Theorem 5.18 of [Nienhuys-Cheng and de Wolf, 1997]) Let C be a set of clauses. Then C is unsatisfiable, that is, $C \models \Box$, if and only if there exists a resolution derivation of the empty clause $\Box$ starting from C.

Due to the soundness and refutation completeness of resolution (for propositional Horn clauses) we now have an effective procedure for deciding logical entailment. Deciding whether a set of Horn clauses logically entails a Horn clause, that is whether $C \models h \leftarrow b_1, \cdots, b_n$, can be realized as follows:

- negate the clause $h \leftarrow b_1, \cdots, b_n$, which yields the clauses $T = \{ \leftarrow h, b_1 \leftarrow, \cdots, b_n \leftarrow \}$
- try to derive the empty clause $\Box$ from $C \wedge T$, that is, decide whether $C \wedge T \vdash \Box$.

Example 2.31. Reconsider the clauses in Ex. 2.28. We now prove by refutation that $T \models \mathsf{pos} \leftarrow \mathsf{square}, \mathsf{red}$. To this end, we first negate the clause, which yields the clauses:

$$\mathsf{square} \leftarrow \text{ and } \mathsf{red} \leftarrow \text{ and } \leftarrow \mathsf{pos}$$

Figure 2.3 shows how to derive $\Box$ from these clauses and T.

So far, we have introduced resolution for propositional logic, but resolution becomes more interesting when clauses contain terms. The key difference is that *unification* must be used in this case. Unification is needed to make a literal in one clause match a literal in the other clause. For instance, when trying to resolve

$$\mathsf{father(X,Y)} \leftarrow \mathsf{parent(X,Y)}, \mathsf{male(X)}$$

with

$$\leftarrow \mathsf{father(luc, maarten)}$$

it is necessary to unify the literals $\mathsf{father(X,Y)}$ and $\mathsf{father(luc, maarten)}$ using the substitution $\{\mathsf{X/luc}, \mathsf{Y/maarten}\}$ to yield the clause

$$\leftarrow \mathsf{parent(luc, maarten)}, \mathsf{male(luc)}.$$

Unification was already implicitly used in Algos. 2.1 and 2.2. Formally, a *unifier* of two expressions f_1 and f_2 (terms or atoms) is a substitution such that $f_1\theta = f_2\theta$.

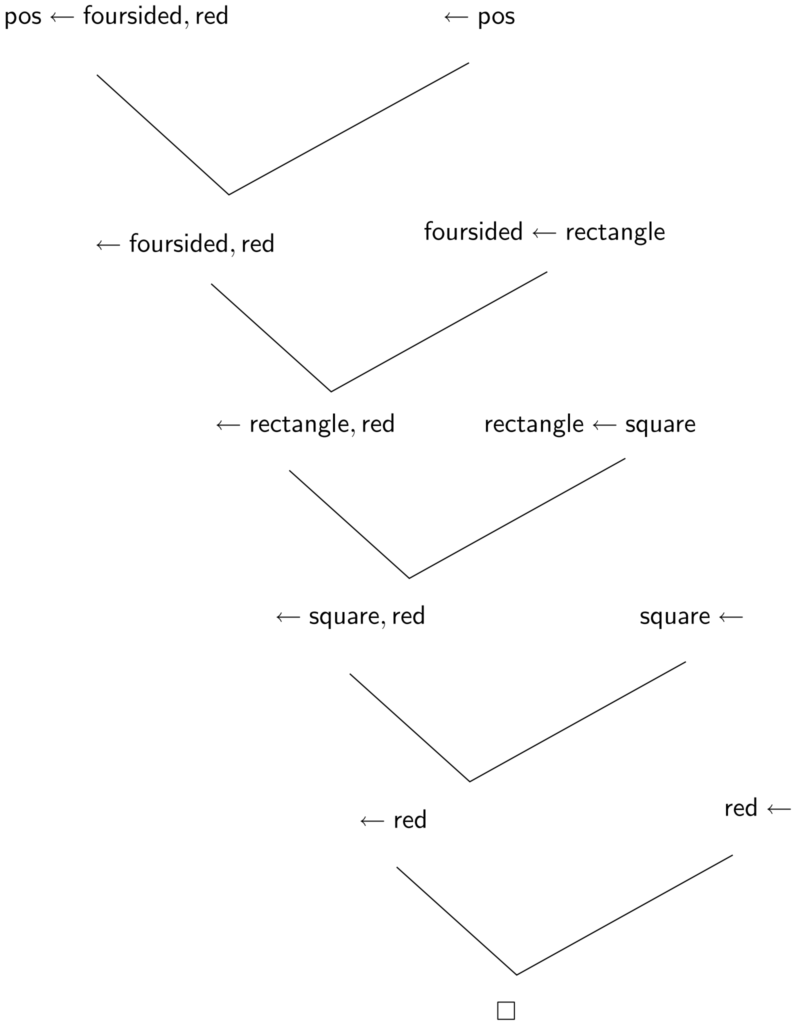


Fig. 2.3. A proof by refutation

Example 2.32. For instance, to unify $\mathsf{father(luc, X)}$ and $\mathsf{father(Y, soetkin)}$ one can use the substitution $\{\mathsf{Y/luc, X/soetkin}\}$, and to unify the atoms $\mathsf{plus(succ(succ(0)), succ(X), succ(Y))}$ and $\mathsf{plus(A, B, succ(succ(C)))}$ one can use

$$\begin{aligned}\theta_1 &= \{\mathsf{A/succ(succ(0)), B/succ(X), Y/succ(C)}\}, \text{ or}\\ \theta_2 &= \{\mathsf{A/succ(succ(0)), B/succ(0), X/0, Y/succ(C)}\}.\end{aligned}$$

As another example consider the atoms $\mathsf{plus(0, X, X)}$ and $\mathsf{plus(Y, succ(Y), Z)}$. One possible unifier is $\{\mathsf{Y/0, X/succ(0), Z/succ(0)}\}$, which illustrates how bindings over one occurrence of a variable are propagated to other occurrences.

Notice also that unifiers do not always exist; for instance, the atoms $\mathsf{father(john, frank)}$ and $\mathsf{father(X, paul)}$ do not unify. In addition, as Ex. 2.32 illustrates, unifiers are not necessarily unique. To guarantee the refutation completeness of resolution when working with definite clauses in first-order logic it is necessary to work with a special type of unifier, which one calls *most general unifiers.*

Definition 2.33. *The most general unifier θ of two formulae F_1 and F_2 is a unifier such that for every other unifier σ of F_1 and F_2, there exists a non-trivial substitution ρ such that $\sigma = \theta\rho$. A trivial substitution is one that maps each variable onto itself.*

The substitution $\theta\rho$ denotes the substitution that first applies θ and then ρ. The resulting set of equalities can be obtained by first applying ρ to the right-hand side terms in θ and then adding the other equalities in ρ (for variables not yet occurring in θ) to θ.

Example 2.34. According to this definition, the substitution θ_2 for the plus atoms in Ex. 2.32 is not a most general unifier as $f_i\theta_2$ can be written as $(f_i\theta_1)\rho$ where $\rho = \{\mathsf{X/0}\}$. Similarly, $\sigma =\{\mathsf{Z/f(h(W)), X/h(W), Y/g(a)}\}$ is not a most general unifier of $\mathsf{p(f(f(X)), g(Y))}$ and $\mathsf{p(f(Z), g(g(a))}$ as the $F_i\sigma$ can be written as $(F_i\theta)\rho$ where $\theta = \{\mathsf{Z/f(X), Y/g(a)}\}$ and $\rho = \{\mathsf{X/h(W)}\}$. Observe that $\sigma = \theta\rho$.

Even though the reader by now has probably already obtained an intuitive understanding of how to compute the most general unifier, we sketch a unification algorithm in Algo. 2.3.

Algorithm 2.3 Computing the $mgu(t_1, t_2)$ of two terms t_1 and t_2; after Lloyd [1987]

```
$\theta := \emptyset$
while $t_1\theta \neq t_2\theta$ do
  find the disagreement set $D$ of $t_1\theta$ and $t_2\theta$
  if there exist a variable $v$ and term $t$ in $D$ and $v$ does not occur in $t$ then
    $\theta := \theta\{v = t\}$
  else
    output $t_1$ and $t_2$ are not unifiable
  end if
end while
```

The *disagreement set* of two terms t_1 and t_2 is found by locating the leftmost positions in the terms at which t_1 and t_2 are different and extracting from each position the corresponding sub-term. Unification and disagreement sets are illustrated in the following example.

Example 2.35. Reconsider computing the *mgu* of the atoms $\mathsf{plus}(0,X,X)$ and $\mathsf{plus}(Y,\mathsf{succ}(Y),Z)$ using Algo. 2.3. The disagreement sets and substitutions evolve as follows through the different iterations of the algorithm:

1. $\theta = \emptyset$ and $\mathsf{plus}(0,X,X), \mathsf{plus}(Y,\mathsf{succ}(Y),Z)$. Hence, $D = \{Y, 0\}$
2. $\theta = \{Y/0\}$ and $\mathsf{plus}(0,X,X), \mathsf{plus}(0,\mathsf{succ}(0),Z)$. Hence, $D = \{X, \mathsf{succ}(0)\}$
3. $\theta = \{Y/0, X/\mathsf{succ}(0)\}$ and $\mathsf{plus}(0,\mathsf{succ}(0),\mathsf{succ}(0)), \mathsf{plus}(0,\mathsf{succ}(0),Z)$. Hence, $D = \{\mathsf{succ}(0), Z\}$
4. $\theta = \{Y/0, X/\mathsf{succ}(0), \mathsf{succ}(0)/Z\}$ and $t_1\theta = t_2\theta = \mathsf{plus}(0,\mathsf{succ}(0),\mathsf{succ}(0))$ implying that θ is the *mgu*.

Now we can define the general resolution rule. Given the clauses $l \leftarrow b_1, \cdots, b_n$ and $h \leftarrow c_1, \cdots, c_{i-1}, l', c_{i+1}, \cdots, c_m$ the *resolution* operator infers the *resolvent* $h\theta \leftarrow b_1\theta, \cdots, b_n\theta, c_1\theta, \cdots, c_m\theta$, where $\theta = mgu(l, l')$. In formal form, this yields:

$$\frac{l \leftarrow b_1, \cdots, b_n \text{ and } h \leftarrow c_1, \cdots, c_{i-1}, l', c_{i+1}, \cdots, c_m \text{ and } \theta = mgu(l, l')}{h\theta \leftarrow b_1\theta, \cdots, b_n\theta, c_1\theta, \cdots, c_m\theta} \tag{2.4}$$

Again, this is written as

$$\begin{array}{c} l \leftarrow b_1, \cdots, b_n \text{ and } h \leftarrow c_1, \cdots, c_{i-1}, l', c_{i+1}, \cdots, c_m \\ \vdash_{res} \\ h\theta \leftarrow b_1\theta, \cdots, b_n\theta, c_1\theta, \cdots, c_m\theta \end{array} \tag{2.5}$$

Example 2.36.

$$\begin{array}{l} \mathsf{cites}(X,Y) \leftarrow \mathsf{authorOf}(X,A), \mathsf{authorOf}(Y,B), \mathsf{reference}(A,B) \\ \mathsf{authorOf}(\mathsf{lloyd}, \mathsf{logic\_for\_learning}) \leftarrow \end{array}$$

$$\vdash_{res}$$

$$\mathsf{cites}(\mathsf{lloyd},Y) \leftarrow \mathsf{authorOf}(Y,B), \mathsf{reference}(\mathsf{logic\_for\_learning},B)$$

The properties and definitions of resolution proofs, and trees, refutation proofs, soundness and refutation completeness for propositional logic carry over to the first-order case. Only one point changes: the method to prove $C \models h \leftarrow b_1, \cdots, b_n$ by refutation. More specifically, the step where the clause is negated needs to take into account the quantifiers and the variables occurring in the clause. As the negation of a universally quantified formula is an existentially quantified negated formula, the refutation proof uses $\leftarrow h\theta$, $b_1\theta \leftarrow$, ... , $b_n\theta \leftarrow$, where θ is a so-called *skolem substitution.* Skolem substitutions replace all variables in $h \leftarrow b_1, ..., b_n$ by distinct constants not appearing anywhere else in the theory or clause.

Example 2.37. To prove by refutation that

$$\mathsf{flies(X)} \leftarrow \mathsf{bird(X)} \models \mathsf{flies(Y)} \leftarrow \mathsf{bird(Y), normal(Y)},$$

we need to negate the clause. The negated clause is represented by the following clauses:

$$\mathsf{bird(sk)} \leftarrow \text{ and } \mathsf{normal(sk)} \leftarrow \text{ and } \leftarrow \mathsf{flies(sk)}$$

It is left as an exercise for the reader to show that there indeed exists a resolution derivation of the empty clause $\square$ starting from the resulting theory.

From a computational perspective, it is important to know also that in propositional logic, logical entailment is *decidable*, whereas in first-order logic it is only *semi-decidable*. So, there exist algorithms for propositional logic that will correctly answer $C \models c$ in finite time. *Semi-decidability*, however, means that there only exist algorithms that will terminate when $C \models c$ but may not terminate when $C \not\models c$. Notice also that when working with relational definite clause logic — that is, definite clause logic with only constant symbols and no function symbols — queries can again be decided in finite time. One way to answer queries for relational definite clause logic in finite time is to first compute the least Herbrand model of the database and then to answer the queries against the least Herbrand model using the SLD-resolution procedure listed below. Other, more effective ways for realizing this also exist, but are beyond the scope of this chapter.

2.5 Prolog and SLD-resolution

First-order logic forms the basis of the programming language Prolog. It is employed by many logical and relational learning approaches, which motivates the present section, in which Prolog's query-answering procedure is investigated. All the logical concepts underlying *pure* Prolog have already been introduced in the previous section. A pure Prolog program is a set of definite clauses and its semantics is given by the least Herbrand model. Several examples of such programs were already given, in particular in Sect. 2.1, where relational databases were represented by a set of definite clauses, and where queries (or goals) in the form of *denials* were employed. Recall that a *denial* is a Horn clause of the form $\leftarrow q_1, \cdots, q_n$. An example query is, for instance, $\leftarrow \mathsf{reference(X, foundations\_of\_lp), authorOf(russell, X)}$, which asks whether there exists an X that references $\mathsf{foundations\_of\_lp}$ and that is written by $\mathsf{russell}$. If one makes the quantifier explicitly, one obtains $\exists \mathsf{X} : \mathsf{reference(X, foundations\_of\_lp)} \wedge \mathsf{authorOf(russell, X)}$. To answer such queries, Prolog employs again the idea of refutation. So, it would negate the expression

$$\exists \mathsf{X} : \mathsf{reference(X, foundations\_of\_lp)} \wedge \mathsf{authorOf(russell, X)}$$

yielding

$$\forall \mathsf{X} : \neg(\mathsf{reference}(\mathsf{X}, \mathsf{foundations\_of\_lp}) \wedge \mathsf{authorOf}(\mathsf{russell}, \mathsf{X}))$$

which is written in clausal logic as the denial

$$\leftarrow \mathsf{reference}(\mathsf{X}, \mathsf{foundations\_of\_lp}), \mathsf{authorOf}(\mathsf{russell}, \mathsf{X}).$$

Prolog will then attempt to derive the empty clause $\Box$ from such denials and return the substitutions θ computed in these derivations as answers.

There are in general many possible derivations and so, a systematic way is needed to explore these in order to obtain all possible answers to a query. Prolog employs a particular form of resolution, called *SLD-resolution*, to realize this. The *SLD-resolution operator* is the special case of the resolution operator $\vdash_{res}$ introduced earlier in which the literal selected is always the leftmost one in the query. Formally, this yields:

$$\frac{l \leftarrow b_1, \cdots, b_n \text{ and } \leftarrow l', c_1, \cdots, c_m \text{ and } \theta = mgu(l, l')}{\leftarrow b_1\theta, \cdots, b_n\theta, c_1\theta, \cdots, c_m\theta} \tag{2.6}$$

An *SLD-derivation* of a query q_n from a set of definite clauses T and a query q_0, notation $T \vdash_{SLD} q$, is a sequence of resolution steps $c_0 \wedge q_0 \vdash_{res} q_1, c_1 \wedge q_1 \vdash_{res} q_2, \cdots, c_{n-1} \wedge q_{n-1} \vdash_{res} q_n$ where $c_i \in T$ for all i. Note also that each time a clause of T is selected to resolve with, its variables are renamed in such a way that the renamed variables have not been used before.

SLD-derivations from a particular theory T can be graphically displayed as a sequence of queries $q_0, \cdots, q_n$ if one assumes that all the clauses c used in the derivation belong to T.

Example 2.38. Reconsider the theory defining the ancestor/2 predicate in Ex. 2.25. The refutation proof and the SLD-derivation for the query or goal $\leftarrow$ ancestor(leo, luc) are shown in Figs. 2.4 and 2.5, respectively.

For a particular query q and theory T, there are often several possible SLD-derivations. Some of these will be successful, that is, end in $\Box$; others will fail and result in dead ends from which no further SLD-resolution steps are possible w.r.t. the theory T. The *set* of all possible SLD-derivations for a particular query and theory is summarized in the so-called *SLD-tree*; an example for the anc/2 predicate defined earlier is shown in Fig. 2.6.

The possible resolvents of a node in the SLD-tree are ordered (from left to right) according to the order in which the clauses are written in the theory (from top to bottom). When trying to answer a query, Prolog will traverse the SLD-tree *depth-first* and *left-to-right*. This is actually an incomplete search strategy because Prolog could get stuck in an infinite branch as illustrated in Fig. 2.7 for the nat/1 predicate. A breadth-first strategy for exploring the SLD-tree yields a complete search strategy, but it is not used by Prolog because the memory requirements imposed by breadth-first search are computationally too expensive.

Exercise 2.39. Can you find a query for the anc/2 program for which the SLD-tree is infinite?

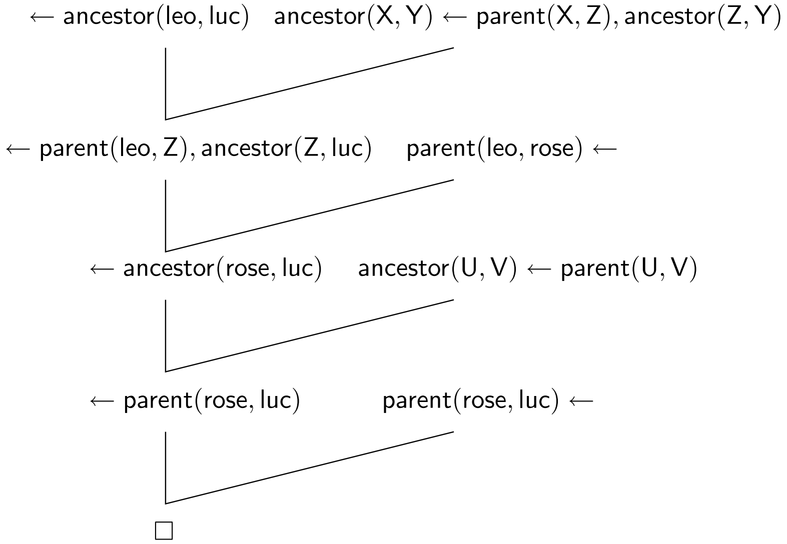

← ancestor(leo, luc)
ancestor(X, Y) ← parent(X, Z), ancestor(Z, Y)
← parent(leo, Z), ancestor(Z, luc)
parent(leo, rose) ←
← ancestor(rose, luc)
ancestor(U, V) ← parent(U, V)
← parent(rose, luc)
parent(rose, luc) ←
□

Fig. 2.4. A proof by refutation

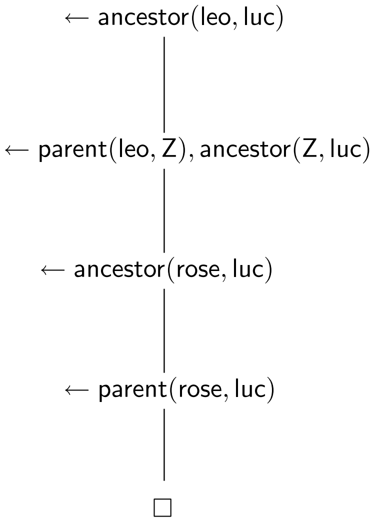


Fig. 2.5. An SLD-refutation and derivation.

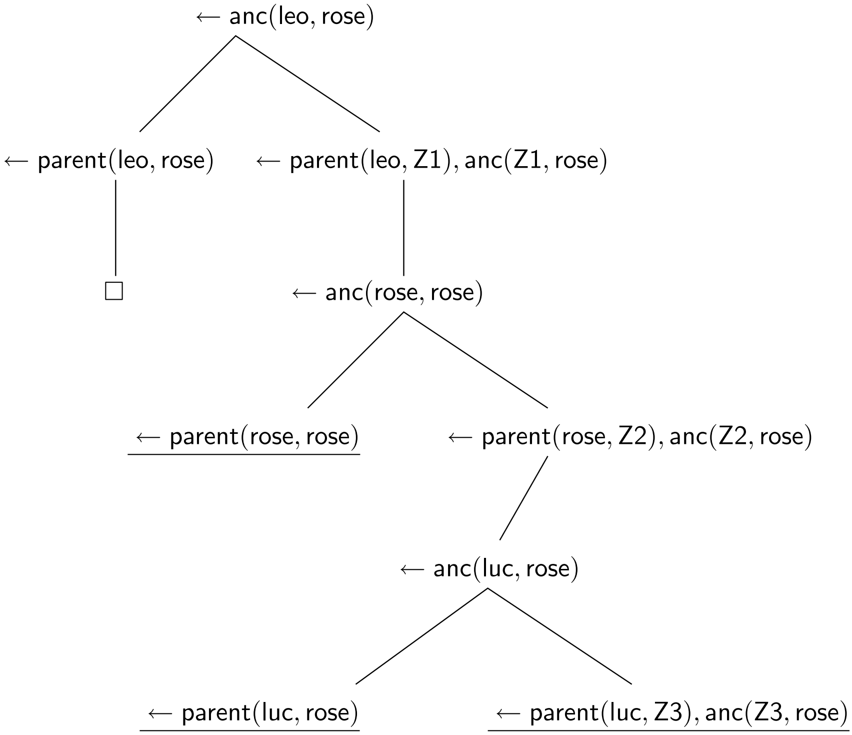


Fig. 2.6. The SLD-tree for ← anc(leo, rose)

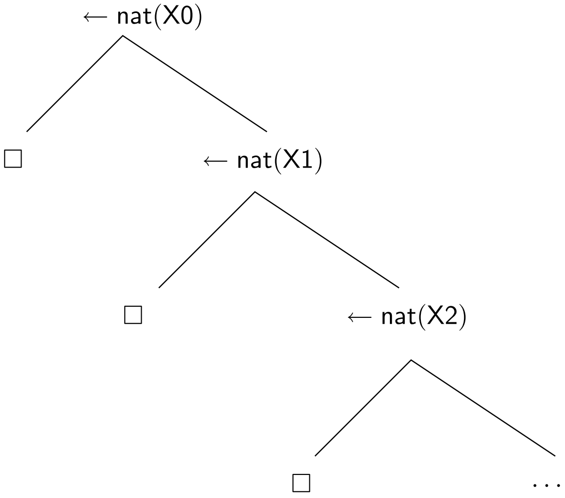


Fig. 2.7. Part of the infinite SLD-tree for ← nat(X0)

2.6 Historical and Bibliographic Remarks

This chapter has briefly reviewed the essentials of computational logic and logic programming needed throughout the rest of this book. The central ideas underlying computational logic and logic programming were introduced by Robinson [1965] and Kowalski [1979]. They formed the basis of the programming language Prolog, which was first implemented by the team of Alain Colmerauer around 1972. These seminal works inspired many further developments in computational logic, and today computational logic is a mature subfield of computer science. Technical contributions to computational logic are contained in the former *Journal of Logic Programming* and the more recent *Theory and Practice of Logic Programming* journal. There exist also several good textbooks on various aspects of logic programming, which the reader may want to consult. A gentle and brief introduction to logic programming and Prolog is given by Flach [1994]. The first three chapters are an especially useful addition to the present chapter. Three outstanding books, which are more oriented towards Prolog programming, are [Bratko, 1990, Sterling and Shapiro, 1986, O'Keefe, 1990]. The role of logic for artificial intelligence is discussed in [Russell and Norvig, 2004, Genesereth and Nilsson, 1987]. The formal foundations of logic programming and inductive logic programming are presented in [Lloyd, 1987, Nienhuys-Cheng and de Wolf, 1997]. Logic has also been used as a representation for databases; cf. [Gallaire et al., 1984, Bancilhon and Ramakrishnan, 1986].

3

An Introduction to Learning and Search

In this chapter, we introduce machine learning and data mining problems, and argue that they can be viewed as search problems. Within this view, the goal is to find those hypotheses in the search space that satisfy a given quality criterion or minimize a loss function. Several quality criteria and loss functions, such as consistency (as in concept learning) and frequency (in association rule mining) are presented, and we investigate desirable properties of these criteria, such as monotonicity and anti-monotonicity. These properties are defined w.r.t. the is more general than *relation and allow one to prune the search for solutions. We also outline several algorithms that exploit these properties.*

3.1 Representing Hypotheses and Instances

In Chapter 1, we presented several showcase applications of logical and relational learning. We also used these cases to introduce the tasks addressed by machine learning and data mining in an informal though general way. Recall that data mining was viewed as the task of finding all patterns expressible within a language of hypotheses satisfying a particular quality criterion. On the other hand, machine learning was viewed as the problem of finding that function within a language of hypotheses that minimizes a loss function. Within this view, machine learning becomes the problem of *function approximation*. Inspecting these views reveals that they are fairly close to one another, and that there are many common issues when looking at symbolic machine learning and data mining.

One of these issues is concerned with knowledge representation. How should patterns, functions, hypotheses and data be represented? It will be useful to distinguish different representation languages for data (instances or examples) and hypotheses (functions, concepts or patterns). Therefore, we assume there is

- a *language of examples* $\mathcal{L}_e$, whose elements are descriptions of instances, observations or data, and

- a *language of hypotheses* $\mathcal{L}_h$, whose elements describe hypotheses (functions or patterns) about the instances, observations or data.

In many situations, it is helpful to employ background knowledge in the mining and learning process. However, for ease of exposition, we postpone the discussion of background knowledge to Section 4.9.

The goal of data mining and machine learning is then to discover hypotheses that provide information about the instances. This implies that the relationship between the language of examples $\mathcal{L}_e$ and of hypotheses $\mathcal{L}_h$ must be known. This relationship can be modeled elegantly by viewing hypotheses $h \in \mathcal{L}_h$ as functions $h : \mathcal{L}_e \rightarrow \mathcal{Y}$ to some domain $\mathcal{Y}$. The learning task is then to approximate an unknown target function f well. This view is illustrated in Fig. 3.1. Different domains are natural for different learning and mining tasks. For instance, in *regression*, the task is to learn a function from $\mathcal{L}_e$ to $\mathcal{Y} = \mathbb{R}$, that is, to learn a real-valued function. As an illustration, consider that we want to learn to assign (real-valued) activities to a set of molecules. On the other hand, when learning definitions of concepts or mining for local patterns, $\mathcal{Y} = \{0, 1\}$ or, equivalently, $\mathcal{Y} = \{true, false\}$. In concept learning, the task could be to learn a description that matches all and only the active molecules. The resulting description is then the concept description.

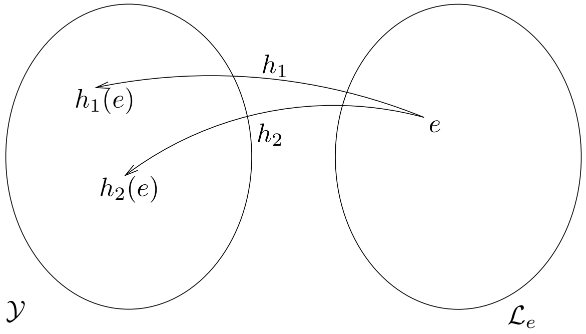


Fig. 3.1. Hypotheses viewed as functions

When the domain of the hypotheses is binary, that is, when $\mathcal{Y} = \{0, 1\}$, it is useful to distinguish the instances that are *covered* by a hypothesis, that is, mapped to 1, from those that are not. This motivates the following definition:

Definition 3.1. *The covers relation* $\mathbf{c}$ *is a relation over* $\mathcal{L}_h \times \mathcal{L}_e$*, and* $\mathbf{c}(h, e) =$ *true if and only if* $h(e) = 1$*.*

Thus the covers relation corresponds to a kind of matching relation. We will sometimes write $\mathbf{c}(h)$ to denote the set of examples in $\mathcal{L}_e$ covered by the hypothesis $h \in \mathcal{L}_h$. Furthermore, the set of examples from $D \subseteq \mathcal{L}_h$ covered by a hypothesis h will sometimes be denoted as $\mathbf{c}(h, D)$. So, $\mathbf{c}(h) = co(h, \mathcal{L}_e)$. This relation is graphically illustrated in Figure 3.2.

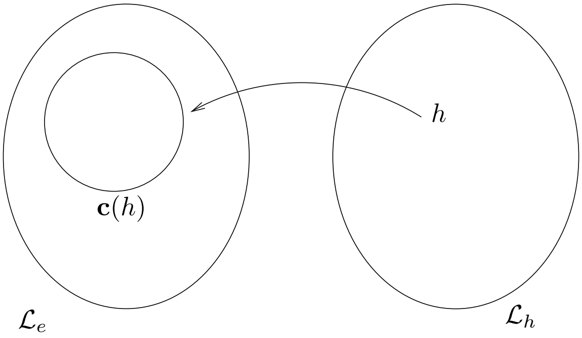


Fig. 3.2. The covers relation

Different notions of coverage as well as choices for $\mathcal{L}_e$ and $\mathcal{L}_h$ can be made. For logical and relational learning, this will be extensively discussed in the next chapter. For the present chapter, however, we will focus on using simple boolean or item-set representations that are so popular in machine learning and data mining. Because these representations are so simple they are ideal for introducing machine learning and data mining problems and algorithms.

3.2 Boolean Data

Due to their simplicity, boolean representations are quite popular within computational learning theory and data mining, where they are better known under the name *item-sets*. In boolean learning, an example is an interpretation over propositional predicates. Recall that this is an assignment of the truth-values $\{true, false\}$ to a set of propositional variables. In the terminology of boolean logic, Herbrand interpretations are often called *variable assignments*.

One of the most popular data mining tasks involving boolean data is that of basket analysis.

Example 3.2. In basket analysis, the aim is to analyze the purchases of clients in, for instance, a supermarket. There is one propositional variable for each of the products available in the supermarket. Assume we have the following set of products $\mathcal{I} = \{\mathsf{sausage}, \mathsf{beer}, \mathsf{wine}, \mathsf{mustard}\}$.

Consider then that the client buys $\mathsf{sausage}, \mathsf{beer}$ and $\mathsf{mustard}$. This corresponds to the interpretation or item-set $\{\mathsf{sausage}, \mathsf{beer}, \mathsf{mustard}\}$. In this case, the language of examples is

$$\mathcal{L}_e = \{I | I \subseteq \{\mathsf{sausage}, \mathsf{beer}, \mathsf{mustard}, \mathsf{wine}\}\}$$

For boolean data, various types of hypotheses languages have been employed. Perhaps, the most popular one is that of conjunctive expressions of the form $p_1 \wedge \ldots \wedge p_n$ where the p_i are propositional atoms. In the data mining literature, these expressions are also called *item-sets* and usually represented as $\{p_1, \cdots, p_n\}$; in the literature on computational learning theory [Kearns and Vazirani, 1994] they are known as *monomials.* So, in this case: $\mathcal{L}_h = \mathcal{L}_e$, which is sometimes called the *single-representation trick.* Using clausal logic, item-sets can be represented by the set of facts $\{p_1 \leftarrow, \ldots, p_n \leftarrow\}$, though this notation is less convenient because it is too lengthy. It will be convenient to use the notation $\mathcal{L}_\mathcal{I}$ to denote all item-sets or conjunctive expressions over $\mathcal{I}$, the set of all items. More formally,

$$\mathcal{L}_\mathcal{I} = \{I | I \subseteq \mathcal{I}\} \tag{3.1}$$

Continuing the basket analysis example above, the hypothesis that someone buys mustard and beer could be represented using mustard ← and beer ←, or more compactly as $\{\mathsf{mustard}, \mathsf{beer}\}$. It is easily verified that this hypothesis covers the example $\{\mathsf{sausage}, \mathsf{beer}, \mathsf{mustard}\}$. The clause $\mathsf{mustard} \leftarrow \mathsf{sausage}, \mathsf{beer}$ describes an association rule, that is, a particular kind of pattern. It states than if a client buys beer and sausage she also buys mustard. When the coverage relation is chosen to coincide with the notion of satisfiability, the example is covered by the clause.

When using purely logical descriptions, the function represented by a hypothesis is typically boolean. However, for the domain of item-sets it is also possible to specify real-valued functions. Consider, for instance, the function

$$h(e) = \mathsf{sausage} + 2 \times \mathsf{beer} + 4 \times \mathsf{wine} + \mathsf{mustard}$$

that computes the price of the basket e.

3.3 Machine Learning

The fundamental problem studied in machine learning is that of function approximation. In this setting, it is assumed that there is an unknown target function $f : \mathcal{L}_e \rightarrow \mathcal{Y}$, which maps instances in $\mathcal{L}_e$ to values in $\mathcal{Y}$. In addition, a set of examples E of the input-output behavior of f is given. The task is then to find a hypothesis $h \in \mathcal{L}_h$ that approximates f well as measured by a so-called *loss* function.

Given

- a language of examples $\mathcal{L}_e$;
- a language of hypotheses $\mathcal{L}_h$;
- an unknown target function $f : \mathcal{L}_e \rightarrow \mathcal{Y}$;
- a set of examples $E = \{(e_1, f(e_1)), \cdots, (e_n, f(e_n))\}$ where each $e_i \in \mathcal{L}_e$;

- a loss function $loss(h, E)$ that measures the quality of hypotheses $h \in \mathcal{L}_h$ w.r.t. the data E;

Find the hypothesis $h \in \mathcal{L}_h$ that minimizes the loss function, that is, for which

$$h = \arg\min loss(h, E) \tag{3.2}$$

As already indicated, various machine learning tasks can be obtained by varying $\mathcal{Y}$. In the simplest case of *binary classification* or *concept learning*, $\mathcal{Y} = \{1, 0\}$, and the task is to learn how to discriminate positive from negative examples. When working with item-sets, this could be baskets that are profitable or not. A natural loss function for this task minimizes the *empirical risk*:

$$loss_{er}(E, h) = \frac{1}{|E|} \sum_i |f(e_i) - h(e_i)| \tag{3.3}$$

So, minimizing the empirical risk corresponds to minimizing the number of errors made on the training data E. Note, however, that minimizing the empirical risk does not guarantee that the hypothesis will also have a high accuracy on unseen data. This view on classification can easily be generalized to take into account more than two classes.

A *regression* setting is obtained by choosing $\mathcal{Y} = \mathbb{R}$. The task is then to learn to predict real values for the examples. As an example of such a function, consider learning a function that predicts the profit the shop makes on a basket. The most popular loss function for regression minimizes the sum of the squared errors, the so-called *least mean squares* loss function:

$$loss_{lms}(E, h) = \sum_i (f(e_i) - h(e_i))^2 \tag{3.4}$$

Finally, in a probabilistic setting, the function to be approximated can be replaced by a probability distribution or density. A popular criterion in this case is to maximize the (log) likelihood of the data; cf. Chapter 8.

This view of machine learning as function approximation will be useful especially in later chapters, such as Chapter 8 on probabilistic logic learning and Chapter 9 on distance and kernel-based learning.

3.4 Data Mining

The purpose of most common data mining tasks is to find hypotheses (expressible within $\mathcal{L}_h$) that satisfy a given quality criterion $\mathcal{Q}$. The quality criterion $\mathcal{Q}$ is then typically expressed in terms of the coverage relation $\mathbf{c}$ and the data set D. This can be formalized in the following definition:

Given

- a language of examples $\mathcal{L}_e$;
- a language of hypotheses (or patterns) $\mathcal{L}_h$;
- a data set $D \subseteq \mathcal{L}_e$; and
- a quality criterion $\mathcal{Q}(h, D)$ that specifies whether the hypothesis $h \in \mathcal{L}_h$ is acceptable w.r.t. the data set D;

Find the set of hypotheses

$$Th(\mathcal{Q}, D, \mathcal{L}_h) = \{h \in \mathcal{L}_h \mid \mathcal{Q}(h, D) \textit{ is true}\} \tag{3.5}$$

When the context is clear, we will often abbreviate $Th(\mathcal{Q}, D, \mathcal{L}_h)$ as Th. This definition has various special cases and variants. First, the data mining task can be to find *all* elements, k elements or just one element that satisfies the quality criterion $\mathcal{Q}$. Second, a large variety of different quality criteria are in use. These can be distinguished on the basis of their *global*, *local* or *heuristic* nature. *Local* quality criteria are predicates whose truth-value is a function of the hypothesis h, the covers relation **c** and the data set D only. On the other hand, a *global* quality criterion is not only a function of the hypothesis h, the covers relation **c** and the data set D, but also of the other hypotheses in $\mathcal{L}_h$.

One function that is commonly used in data mining is that of frequency. The frequency $freq(h, D)$ of a hypothesis h w.r.t. a data set D is the cardinality of the set **c**(h,D):

$$freq(h, D) = \mid \mathbf{c}(h, D) \mid \tag{3.6}$$

In this definition, the *absolute* frequency is expressed in absolute terms, that is, the frequency is a natural number. Sometimes, frequency is also expressed *relatively* to the size of the data set D. Thus the *relative* frequency is

$$rfreq(h, D) = \frac{freq(h, D)}{\mid D \mid} \tag{3.7}$$

An example of a local quality criterion is now a minimum frequency constraint. Such a constraint states that the frequency of a hypothesis h on the data set D should exceed a threshold, that is, $\mathcal{Q}(h, D)$ is of the form $freq(h, D) > x$ where x is a natural number or $rfreq(h, D) > y$ where y is a real number between 0 and 1. These criteria are local because one can verify whether they hold by accessing the hypothesis h and D only. There is no need to know the frequency of the other hypotheses in $\mathcal{L}_h$.

An example of a global quality criterion is to require that the accuracy of a hypothesis h w.r.t. a set of positive P and negative example N is maximal. The accuracy $acc(h, P, N)$ is then defined as

$$acc(h, P, N) = \frac{freq(h, P)}{freq(h, P) + freq(h, N)}. \tag{3.8}$$

The maximal accuracy constraint now states

$$\mathcal{Q}(h, P, N) = \big(h = \arg\max_{h \in \mathcal{L}_h} acc(h, P, N)\big) \tag{3.9}$$

This constraint closely corresponds to minimizing the empirical loss in a function approximation setting.

Because the machine learning and data mining views are quite close to one another, at least when working with symbolic representations, we shall in the present chapter largely employ the data mining perspective. When shifting our attention to take into account more numerical issues, in Chapter 8 on probabilistic logic learning and Chapter 9 on distance and kernel-based learning, the machine learning perspective will be more natural. The reader must keep in mind though, that in most cases the same principles apply and are, to some extent, a matter of background or perspective.

3.5 A Generate-and-Test Algorithm

Depending on the type and nature of the quality criterion considered, different algorithms can be employed to compute $Th(\mathcal{Q}, D, \mathcal{L}_h)$. For a given quality criterion and hypotheses space, one can view mining or learning as a search process. By exploiting this view, a (trivial) algorithm based on the well-known generate-and-test technique in artificial intelligence can be derived. This so-called *enumeration algorithm* is shown in Algo. 3.1.

Algorithm 3.1 The enumeration algorithm

```
for all $h \in \mathcal{L}_h$ do
  if $\mathcal{Q}(h, D) = true$ then
    output $h$
  end if
end for
```

Although the algorithm is naive, it has some interesting properties: whenever a solution exists, the enumeration algorithm will find it. The algorithm can only be applied if the hypotheses language $\mathcal{L}_h$ is *enumerable*, which means that it must be possible to generate all its elements. As the algorithm searches the whole space, it is inefficient. This is a well-known property of generate-and-test approaches. Therefore, it is advantageous to structure the search space in machine learning, which will allow for its pruning. Before discussing how the search space can be structured, let us illustrate the enumeration algorithm. This illustration, as well as most other illustrations and examples in this chapter, employs the representations of boolean logic.

Example 3.3. Reconsider the problem of basket analysis sketched in Ex. 3.2. In basket analysis, there is a set of propositional variables (usually called items)

$\mathcal{I} = \{\mathsf{s} = \mathsf{sausage}, \mathsf{m} = \mathsf{mustard}, \mathsf{b} = \mathsf{beer}, \mathsf{c} = \mathsf{cheese}\}$. Furthermore, every example is an interpretation (or item-set) and the hypotheses are, as argued in Ex. 3.2, members of $\mathcal{L}_{\mathcal{I}}$. Consider also the data set

$$D = \{\{\mathsf{s}, \mathsf{m}, \mathsf{b}, \mathsf{c}\}, \{\mathsf{s}, \mathsf{m}, \mathsf{b}\}, \{\mathsf{s}, \mathsf{m}, \mathsf{c}\}, \{\mathsf{s}, \mathsf{m}\}\}$$

and the quality criterion $\mathcal{Q}(h, D) = (freq(h, D) \geqslant 3)$. One way of enumerating all item-sets in $\mathcal{L}_{\mathcal{I}}$ for our example is given in Fig. 3.3. Furthermore, the item-sets satisfying the constraint are underlined.

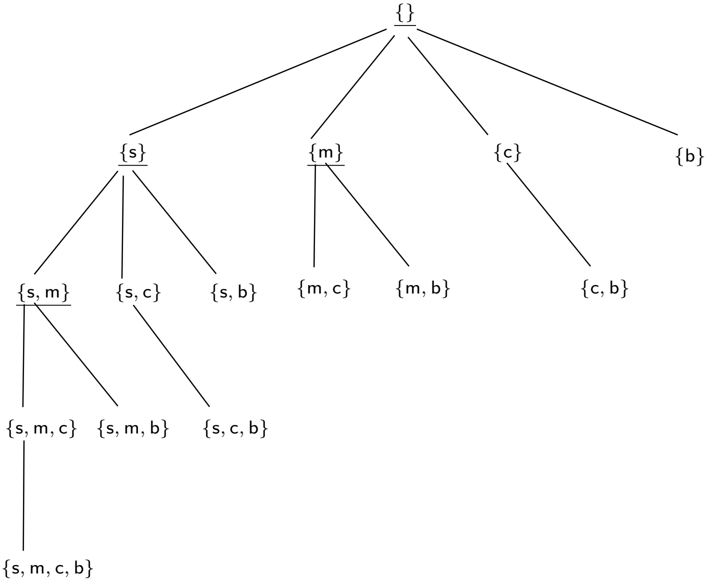


Fig. 3.3. Enumerating and testing monomials or item-sets

3.6 Structuring the Search Space

One natural way to structure the search space is to employ the *generality* relation.

Definition 3.4. *Let $h_1, h_2 \in \mathcal{L}_h$. Hypothesis h_1* is more general than *hypothesis h_2, notation $h_1 \preceq h_2$, if and only if all examples covered by h_2 are also covered by h_1, that is, $\mathbf{c}(h_2) \subseteq \mathbf{c}(h_1)$.*

We also say that h_2 is a specialization of h_1, h_1 is a generalization of h_1 or h_2 is more general than h_1.[1] This notion is illustrated in Fig. 3.4. Furthermore,

[1] It would be more precise to state that h_2 is at least as general than h_1. Nevertheless, we shall use the standard terminology, and say that h_1 is more general than h_2.

when $h_1 \preceq h_2$ but h_1 covers examples not covered by h_2 we say that h_1 is a proper generalization of h_2, and we write $h_1 \prec h_2$.

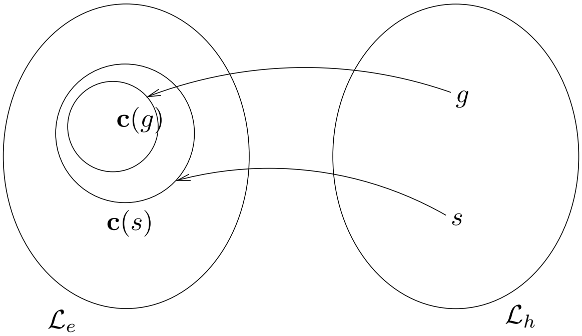


Fig. 3.4. Hypothesis g is more general than hypothesis s

Notice that the generality relation is transitive and reflexive. Hence, it is a *quasi-order*. Unfortunately, it is not always anti-symmetric since there may exist several hypotheses that cover exactly the same set of examples. Such hypotheses are called *syntactic variants*. Syntactic variants are undesirable because they introduce redundancies in the search space. In theory, one can obtain a *partial order* by introducing equivalence classes and working with a canonical form as a representative of the equivalence class. In practice, this is not always easy, as will be explained in the next chapter.

Example 3.5. Consider the task of basket analysis used in Ex. 3.3. The conjunction $\mathsf{sausage} \wedge \mathsf{beer}$ is more general than $\mathsf{sausage} \wedge \mathsf{beer} \wedge \mathsf{cheese}$, or when using set notation $\{\mathsf{sausage}, \mathsf{beer}\}$ is more general than $\{\mathsf{sausage}, \mathsf{beer}, \mathsf{cheese}\}$ because the former is a subset of the latter.

Furthermore, if we would possess background knowledge in the form of a taxonomy stating, for instance, that $\mathsf{alcohol} \leftarrow \mathsf{beer}$; $\mathsf{food} \leftarrow \mathsf{cheese}$; $\mathsf{food} \leftarrow \mathsf{sausage}$; and $\mathsf{food} \leftarrow \mathsf{mustard}$, then the conjunction $\mathsf{food} \wedge \mathsf{beer}$ together with the background theory would be more general than $\mathsf{sausage} \wedge \mathsf{beer}$. Using the taxonomy, specific baskets such as $\{\mathsf{sausage}, \mathsf{beer}\}$ can be completed under the background theory, by computing the least Herbrand model of the item-set and the background theory, yielding in our example $\{\mathsf{sausage}, \mathsf{beer}, \mathsf{food}, \mathsf{alcohol}\}$. This is the learning from interpretations setting, that we shall discuss extensively in the next chapter. Whenever an example contains $\mathsf{sausage}$ in this setting the resulting completed example will contain food as well.

Continuing the illustration, if we assume that the examples only contain the items from $\mathcal{I}$ and are then completed using the clauses listed above, then the conjunctions $\mathsf{alcohol} \wedge \mathsf{cheese}$ and $\mathsf{beer} \wedge \mathsf{cheese}$ are syntactic variants, because there is only one type of item belonging to the category $\mathsf{alcohol}$.

When the language $\mathcal{L}_h$ does not possess syntactic variants, which will be assumed throughout the rest of this chapter, the generality relation imposes a *partial order* on the search space and can be graphically depicted using a so called Hasse diagram. This is illustrated in Fig. 3.5.

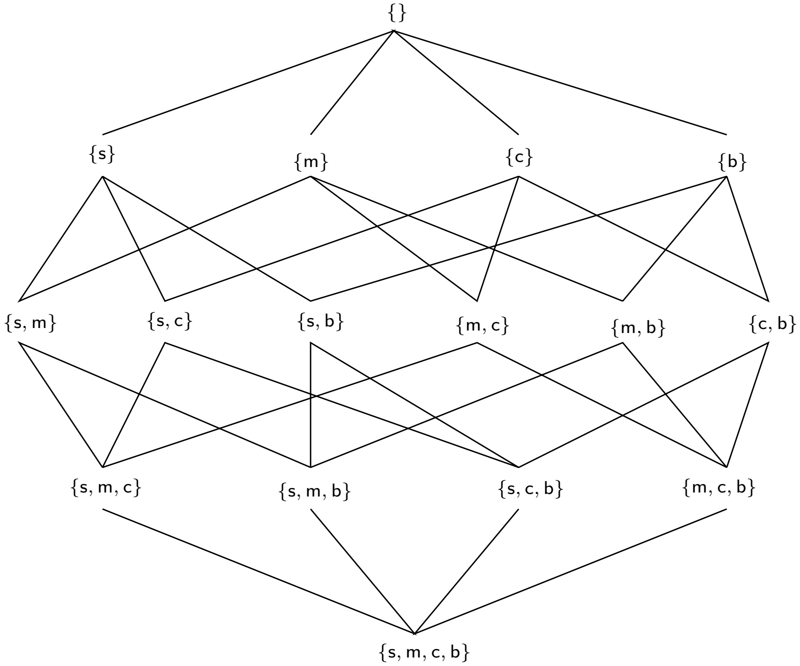


Fig. 3.5. The partial order over the item-sets

It is often convenient to work with a special notation for the maximally general top element $\top$ and the maximally specific bottom element $\bot$ such that $\mathbf{c}(\top) = \mathcal{L}_e$ and $\mathbf{c}(\bot) = \emptyset$. Furthermore, when the elements $\top$ and $\bot$ do not exist in $\mathcal{L}_h$ they are often added to the language. For item-sets, $\top = \emptyset$ and $\bot = \mathcal{I}$.

3.7 Monotonicity

The *generality* relation imposes a useful structure on the search space provided that the quality criterion involves monotonicity or anti-monotonicity.

A quality criterion $\mathcal{Q}$ is *monotonic* if and only if

$$\forall s, g \in \mathcal{L}_h, \forall\ D \subseteq \mathcal{L}_e : (g \preceq s) \wedge \mathcal{Q}(g, D) \rightarrow \mathcal{Q}(s, D) \tag{3.10}$$

It is *anti-monotonic*[2] if and only if

$$\forall s, g \in \mathcal{L}_h, \forall\ D \subseteq \mathcal{L}_e : (g \preceq s) \land \mathcal{Q}(s, D) \rightarrow \mathcal{Q}(g, D) \tag{3.11}$$

To illustrate this definition, observe that a minimum frequency constraint $freq(h, D) \geqslant x$ is anti-monotonic and a maximum frequency constraint $freq(h, D) \leqslant x$ is monotonic. Similarly, the criterion that requires that a given example be covered (that is, $e \in \mathbf{c}(h)$) is anti-monotonic and the one that requires that a given example is not covered (that is, $e \notin \mathbf{c}(h)$) is monotonic. On the other hand, the criterion $acc(h, P, N) \geqslant x$ is neither monotonic nor anti-monotonic.

Exercise 3.6. Let $A_1(h, D)$ and $A_2(h, D)$ be two anti-monotonic criteria, and $M_1(h, D)$ and $M_2(h, D)$ be two monotonic ones. Are the criteria $\neg A_1(h, D)$; $A_1(h, D) \lor A_2(h, D)$; $A_1(h, D) \land A_2(h, D)$; their duals $\neg M_1(h, D)$; $M_1(h, D) \lor M_2(h, D)$; $M_1(h, D) \land M_2(h, D)$; and the combinations $A_1(h, D) \land M_1(h, D)$ and $A_1(h, D) \lor M_1(h, D)$ monotonic and/or anti-monotonic? Argue why.

Exercise 3.7. Show that the criterion $acc(h, P, N) \geqslant x$ is neither monotonic nor anti-monotonic.

Exercise 3.8. * Consider the primitives $free(m)$ for item-sets, which is true if and only if none of the subsets of m have the same frequency as m, and $closed(m)$, which is true if and only if none of the super-sets of m have the same frequency as m. Do freeness and closedness satisfy the anti-monotonicity or monotonicity property? Argue why.

When the quality criterion is monotonic or anti-monotonic it is a good idea to employ the generality relation on the search space and to use specialization or generalization as the basic operations to move through the search space. The reason for this is given by the following two properties, which allow us to prune the search.

Property 3.9. (Prune generalizations) If a hypothesis h does not satisfy a monotonic quality criterion then none of its generalizations will.

Property 3.10. (Prune specializations) If a hypothesis h does not satisfy an anti-monotonic quality criterion then none of its specializations will.

These properties directly follow from the definitions of monotonicity and anti-monotonicity in Eqs. 3.10 and 3.11.

Example 3.11. Reconsider Ex. 3.3 and the anti-monotonic minimum frequency criterion. Because $\mathsf{sausage} \land \mathsf{beer}$ does not satisfy the minimum frequency constraint, none of its specializations do. They can therefore be pruned away, as illustrated in Fig. 3.7.

[2] In the literature, the definitions of the concepts of monotonicity and anti-monotonicity are sometimes reversed.

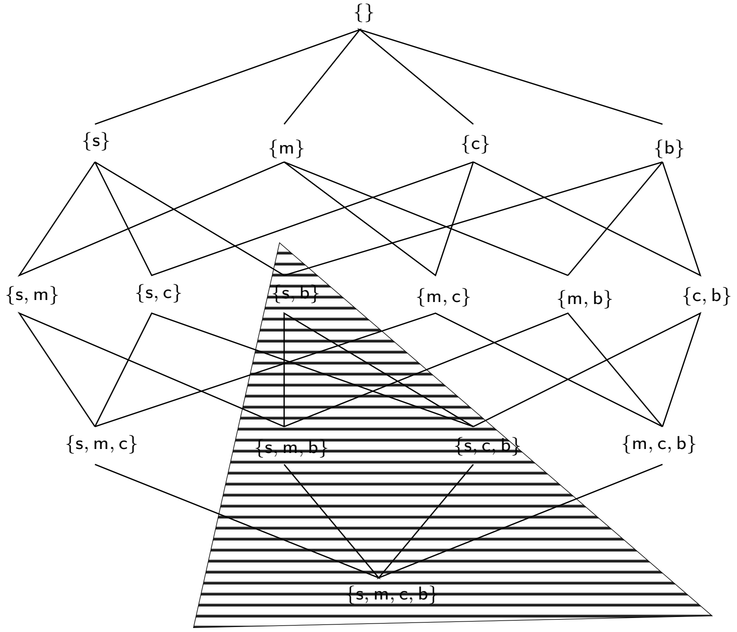


Fig. 3.6. Pruning specializations

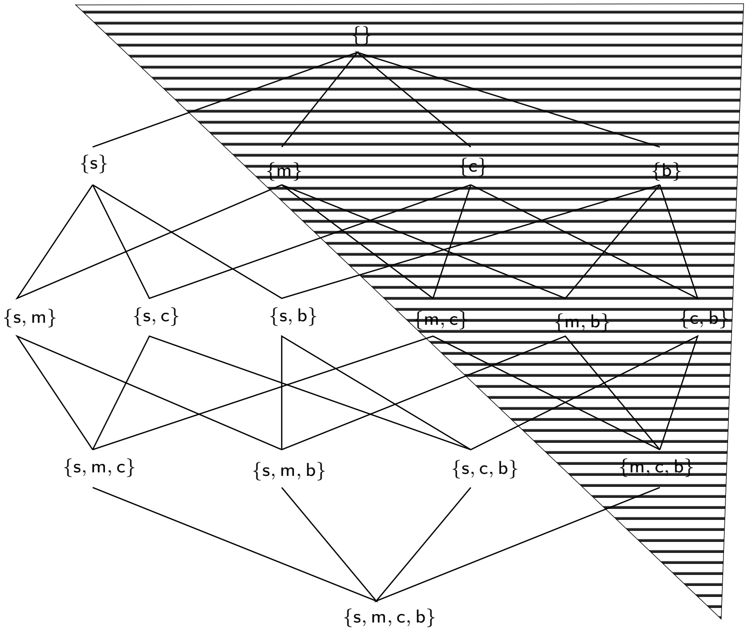


Fig. 3.7. Pruning generalizations

Example 3.12. Reconsider Ex. 3.3 and the monotonic constraint that requires that the example $\{\mathsf{m}, \mathsf{b}, \mathsf{c}\}$ not be covered. Because $\mathsf{mustard} \wedge \mathsf{beer} \wedge \mathsf{cheese}$ covers this example, all its generalizations can be pruned away as illustrated in Fig. 3.7.

3.8 Borders

When monotonic and/or anti-monotonic criteria are used, the solution space has so-called borders. Before introducing borders, let us introduce the $max(T)$ and $min(T)$ primitives:

$$max(T) = \{h \in T \mid \neg\exists t \in T : h \prec t\} \tag{3.12}$$

$$min(T) = \{h \in T \mid \neg\exists t \in T : t \prec h\} \tag{3.13}$$

Intuitively, the maximal elements are the most specific ones. These are also the largest ones when interpreting the symbol $\prec$ as smaller than or equal to. Furthermore, more specific hypotheses are typically also longer.

Example 3.13. Let $T = \{\mathsf{true}, \mathsf{s}, \mathsf{m}, \mathsf{s} \wedge \mathsf{m}\}$. Then $max(T) = \{\mathsf{s} \wedge \mathsf{m}\}$ and $min(T) = \{\mathsf{true}\}$.

Observe that when the hypothesis space $\mathcal{L}_h$ is finite, $max(T)$ and $min(T)$ always exist. When $\mathcal{L}_h$ is infinite, this need not be the case. We illustrate this using string patterns.

Example 3.14. * Many data sets can be conveniently represented using strings over some alphabet Σ; cf. also Chapter 4. An *alphabet* Σ is a finite set of symbols. A *string* $s_1s_2...s_n$ is then a sequence of symbols $s_i \in \Sigma$. For instance, the string over the alphabet $\Sigma = \{\mathsf{a}, \mathsf{c}, \mathsf{g}, \mathsf{t}\}$

$$\mathsf{atgcccaagctgaatagcgtagaggggttttcatcatttgaggacgatgtataa}$$

might represent a sequence of DNA. When working with strings to represent patterns and examples, a natural coverage relation is provided by the notion of substring. A string $S = s_1s_2...s_n$ is a *substring* of a string $T = t_1t_2...t_k$, if and only if $s_1...s_n$ occur at consecutive positions in $t_1...t_k$, that is, there exists a j for which $s_1 = t_j$, $s_2 = t_{j+1}$, ..., and $s_n = t_{j+n}$. For instance, the string *atgc* is a substring of *aatgccccc* with $j = 2$. For the language of all strings over the alphabet Σ, that is, Σ^*, $max(\Sigma^*)$ does not exist. To avoid such complications in this chapter, we assume that $\mathcal{L}_h$ is finite.

For finite languages and monotonic and anti-monotonic quality criteria $\mathcal{Q}$, the solution space $Th(\mathcal{Q}, D, \mathcal{L}_h)$ has boundary sets that are sometimes called borders. More formally, the S-border of maximally specific solutions w.r.t. a constraint $\mathcal{Q}$ is

$$S\big(Th(\mathcal{Q}, D, \mathcal{L}_h)\big) = max\big(Th(\mathcal{Q}, D, \mathcal{L}_h)\big) \tag{3.14}$$

Dually, the G-border of maximally general solutions is

$$G\big(Th(\mathcal{Q}, D, \mathcal{L}_h)\big) = min\big(Th(\mathcal{Q}, D, \mathcal{L}_h)\big) \tag{3.15}$$

Example 3.15. Reconsider Ex. 3.13. The set T is the set of solutions to the mining problem of Ex. 3.3, that is, $T = Th((freq(h, D) \geqslant 3), D, \mathcal{L}_m)$; $S(T) = max(T) = \{\mathsf{s} \wedge \mathsf{m}\}$ and $G(T) = min(T) = \{\mathsf{true}\}$.

The S and G sets are called borders because of the following properties.

Property 3.16. If $\mathcal{Q}$ is an anti-monotonic predicate, then

$$Th(\mathcal{Q}, D, \mathcal{L}_h) = \{h \in \mathcal{L}_h \mid \exists s \in S\big(Th(\mathcal{Q}, D, \mathcal{L}_h)\big) : h \preceq s\}$$

Property 3.17. If $\mathcal{Q}$ is a monotonic predicate, then

$$Th(\mathcal{Q}, D, \mathcal{L}_h) = \{h \in \mathcal{L}_h \mid \exists g \in G\big(Th(\mathcal{Q}, D, \mathcal{L}_h)\big) : g \preceq h\}$$

Thus the borders of a monotonic or an anti-monotonic predicate completely characterize the set of all solutions. At this point the reader may want to verify that the S set in the previous example fully characterizes the set T of solutions to an anti-monotonic query as it contains all monomials more general than the element $\mathsf{s} \wedge \mathsf{m}$ of $S(T)$.

Furthermore, when $\mathcal{Q}$ is the conjunction of a monotonic and an anti-monotonic predicate $\mathcal{M} \wedge \mathcal{A}$ (and the language $\mathcal{L}_h$ is finite), then the resulting solution set is a *version space*. A set T is a *version space* if and only if

$$T = \{h \in \mathcal{L}_h \mid \exists s \in S(T), g \in G(T) : g \preceq h \preceq s\} \tag{3.16}$$

For version spaces, the S and G set together form a *condensed* representation for the version space. Indeed, in many (but not all) cases the border sets will be smaller than the original solution set, while characterizing the same information (as it is possible to recompute the solution set from the border sets).

Example 3.18. Consider the constraint $\mathcal{Q} = (freq(h, D) \geqslant 2) \wedge (freq(h, D) \leqslant 3)$ with D defined as in Ex. 3.3:

$$D = \{\{\mathsf{s}, \mathsf{m}, \mathsf{b}, \mathsf{c}\}, \{\mathsf{s}, \mathsf{m}, \mathsf{b}\}, \{\mathsf{s}, \mathsf{m}, \mathsf{c}\}, \{\mathsf{s}, \mathsf{m}\}\}$$

Then $S(Th) =$$\{\mathsf{s} \wedge \mathsf{m} \wedge \mathsf{c}, \mathsf{s} \wedge \mathsf{m} \wedge \mathsf{b}\}$, and $G(Th) = \{\mathsf{b}, \mathsf{c}\}$ as shown in Fig. 3.8.

Exercise 3.19. Give an example of a quality criterion $\mathcal{Q}$ of the form $\mathcal{M} \vee \mathcal{A}$, with $\mathcal{M}$ a monotonic predicate and $\mathcal{A}$ an anti-monotonic one, a data set D and a language $\mathcal{L}_h$, such that $Th(\mathcal{Q}, D, \mathcal{L}_h)$ is not a version space.

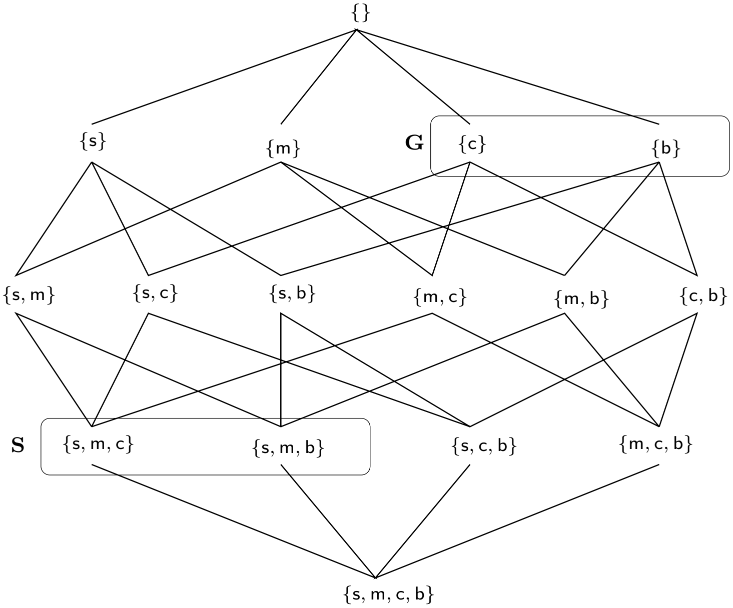


Fig. 3.8. A version space

In concept learning, one is given sets of positive and negative examples P and N. The goal of learning (in an idealized situation where no noise arises), is then to find those hypotheses h that cover all positive and none of the negative examples. Thus concept learning tasks employ the constraint

$$(rfreq(h, P) \geqslant 100\%) \wedge (rfreq(h, N) \leqslant 0\%) \tag{3.17}$$

This is the conjunction of an anti-monotonic and a monotonic predicate. Thus, the solution set to an idealized concept learning task is a version space (according to Eq. 3.16).

Exercise 3.20. Find the S and G sets corresponding to the criterion of Eq. 3.17 when $P = \{\{\mathsf{s}, \mathsf{m}, \mathsf{b}\}, \{\mathsf{s}, \mathsf{c}, \mathsf{b}\}\}$ and $N = \{\{\mathsf{b}\}, \{\mathsf{b}, \mathsf{c}\}\}$.

The S and G sets w.r.t. a version space Th are the so-called *positive* borders because they contain elements that belong to Th. In data mining, one sometimes also works with the negative borders. The negative borders contain the elements that lie just outside Th. The negative borders S^- and G^- can be defined as follows:

$$S^-(Th) = min\big(\mathcal{L}_h - \{h \in \mathcal{L}_h \mid \exists s \in S(Th) : h \preceq s\}\big) \tag{3.18}$$

$$G^-(Th) = max(\mathcal{L}_h - \{h \in \mathcal{L}_h \mid \exists g \in G(Th) : g \preceq h\}) \tag{3.19}$$

Example 3.21. Reconsider the version space Th of Ex. 3.18. For this version space, $S^-(Th) = \{\mathsf{c} \wedge \mathsf{b}\}$ and $G^-(Th) = \{\mathsf{s} \wedge \mathsf{m}\}$.

Finally, note that the size of the border sets can grow very large. Indeed, for certain hypothesis languages (such as item-sets), the size of the G set can grow exponentially large in the number of negative examples for a concept-learning task. Nevertheless, it should be clear that the size of any positive border set can never be larger than that of the overall solution set.

3.9 Refinement Operators

In the previous two sections, it was argued that the generality relation is useful when working with monotonic and/or anti-monotonic quality criteria. The present section introduces *refinement operators* for traversing the search space $\mathcal{L}_h$. The large majority of operators employed in data mining or machine learning algorithms are generalization or specialization operators. They generate a set of specializations (or generalizations) of a given hypothesis. More formally,

A *generalization operator* $\rho_g : \mathcal{L}_h \rightarrow 2^{\mathcal{L}_h}$ is a function such that

$$\forall h \in \mathcal{L}_h : \rho_g(h) \subseteq \{c \in \mathcal{L}_h \mid c \preceq h\} \tag{3.20}$$

Dually, a *specialization* operator $\rho_s : \mathcal{L}_h \rightarrow 2^{\mathcal{L}_h}$ is a function such that

$$\forall h \in \mathcal{L}_h : \rho_s(h) \subseteq \{c \in \mathcal{L}_h \mid h \preceq c\} \tag{3.21}$$

Sometimes, the operators will be applied repeatedly. This motivates the introduction of the level n refinement operator ρ^n:

$$\rho^n(h) = \begin{cases} \rho(h) & \text{if } n = 1, \\ \displaystyle\bigcup_{h' \in \rho^{n-1}(h)} \rho(h') & \text{if } n > 1 \end{cases} \tag{3.22}$$

Furthermore, $\rho^*(h)$ denotes $\rho^\infty(h)$.

Many different types of generalization and specialization operators exist and they are useful in different types of algorithms. Two classes of operators are especially important. They are the so-called *ideal* and *optimal* operators, which are defined below for specialization operators (the corresponding definitions can easily be obtained for generalization operators). ρ is an *ideal* operator for $\mathcal{L}_h$ if and only if

$$\forall h \in \mathcal{L}_h : \rho(h) = min(\{h' \in \mathcal{L}_h \mid h \prec h'\}) \tag{3.23}$$

So, an ideal specialization operator returns all children for a node in the Hasse diagram. Furthermore, these children are *proper* refinements, that is, they are not a syntactic variant of the original hypothesis. Ideal operators are used in heuristic search algorithms.

ρ is an *optimal* operator for $\mathcal{L}_h$ if and only if for all $h \in \mathcal{L}_h$ there exists exactly one sequence of hypotheses $\top = h_0, h_1, ..., h_n = h \in \mathcal{L}_h$ such that

$h_i \in \rho(h_{i-1})$ for all i. Optimal refinement operators are used in complete search algorithms. They have the property that, when starting from $\top$, no hypothesis will be generated more than once.

An operator for which there exists *at least one* sequence from $\top$ to any $h \in \mathcal{L}_h$ is called *complete*, and one for which there exists *at most one* such sequence is *nonredundant*.

Example 3.22. We define two specialization operators ρ_o and ρ_i for the item-sets over $\mathcal{I} = \{\mathsf{s}, \mathsf{m}, \mathsf{b}, \mathsf{c}\}$ using the lexicographic $\ll$ order over $\mathcal{I}$ $\mathsf{s} \ll \mathsf{m} \ll \mathsf{c} \ll \mathsf{b}$:

$$\rho_i(M) = M \cup \{j\} \qquad \text{with } j \in (\mathcal{I} - M) \tag{3.24}$$

$$\rho_o(M) = M \cup \{j\} \qquad \text{with } \forall l \in M : l \ll j \tag{3.25}$$

By repeatedly applying ρ_i to $\top$, one obtains the Hasse diagram in Fig. 3.5, where an edge between two nodes means that the child node is one of the refinements according to ρ_i of the parent node. On the other hand, when applying ρ_o to $\top$, one obtains the tree structure depicted in Fig. 3.3. Both ρ_o^* and ρ_i^* generate all hypotheses in $\mathcal{L}_h$ from $\top$, but there is only a single path from $\top$ to any particular hypothesis using ρ_o. Remark also that using ρ_o amounts to working with a *canonical* form, where $m_1 \wedge ... \wedge m_k$ is in canonical form if and only if $m_1 << ... << m_k$. Repeatedly applying ρ_o on $\top$ only yields hypotheses in canonical form.

Two other operations that are useful in learning and mining algorithms are the *minimally general generalization mgg* and the *maximally general specialization mgs*:

$$mgg(h_1, h_2) = min\{h \in \mathcal{L}_h \mid h \preceq h_1 \wedge h \preceq h_2\} \tag{3.26}$$

$$mgs(h_1, h_2) = max\{h \in \mathcal{L}_h \mid h_1 \preceq h \wedge h_2 \preceq h\} \tag{3.27}$$

If the *mgg* (or *mgs*) operator always returns a unique generalization (or specialization), the operator is called the *least general generalization lgg* or *least upper bound lub* (or the *greatest lower bound glb*). If the *lgg* and *glb* exist for any two hypotheses $h_1, h_2 \in \mathcal{L}_h$, the partially ordered set $(\mathcal{L}_h, \preceq)$ is called a *lattice*. For instance, the language of item-sets $\mathcal{L}_\mathcal{I}$ is a lattice, as the following example shows.

Example 3.23. Continuing the previous example, the operators compute

$$mgg(M_1, M_2) = M_1 \cap M_2 \tag{3.28}$$

$$mgs(M_1, M_2) = M_1 \cup M_2 \tag{3.29}$$

For instance, $mgg(\mathsf{s} \wedge \mathsf{m} \wedge \mathsf{b}, \mathsf{s} \wedge \mathsf{b} \wedge \mathsf{c}) = lgg(\mathsf{s} \wedge \mathsf{m} \wedge \mathsf{b}, \mathsf{s} \wedge \mathsf{b} \wedge \mathsf{c}) = \{\mathsf{s} \wedge \mathsf{b}\}$.

The *mgg* and *lgg* operations are used by specific-to-general algorithms. They repeatedly generalize the current hypothesis with examples. The dual operations, the *mgs* and the *glb*, are sometimes used in algorithms that work from general-to-specific.

Exercise 3.24. Define an ideal and an optimal generalization operator for item-sets.

Exercise 3.25. * Define an ideal and an optimal specialization operator for the hypothesis language of strings Σ^*, where $g \preceq s$ if and only if g is a substring of s; cf. Ex. 3.14. Discuss also the operations *mgg* and *mgs*.

3.10 A Generic Algorithm for Mining and Learning

Now everything is in place to adapt the enumeration algorithm of Algo. 3.1 to employ the refinement operators just introduced. The resulting algorithm is shown in Algo. 3.10. It is a straightforward application of general search principles using the notions of generality.

Algorithm 3.2 A generic algorithm

```
Queue := *Init*;
Th := $\emptyset$;
while not *Stop* do
   *Delete* $h$ from Queue
   if $\mathcal{Q}(h,D)$ = true then
      add $h$ to $Th$
      Queue := Queue $\cup \rho(h)$
   end if
   Queue := *Prune*(Queue)
end while
return Th
```

The algorithm employs a Queue of candidate hypotheses and a set Th of solutions. It proceeds by repeatedly deleting a hypothesis h from Queue and verifying whether it satisfies the quality criterion $\mathcal{Q}$. If it does, h is added to Th; otherwise, all refinements $\rho(h)$ of h are added to the Queue. This process continues until the *Stop* criterion is satisfied. Observe that there are many *generic* parameters (shown in *italics*) in this algorithm. Depending on the particular choice of parameter, the algorithm behaves differently. The *Init* function determines the starting point of the search algorithm. The initialization may yield one or more initial hypotheses. Most algorithms start either at $\top$ and only specialize (the so-called *general-to-specific* systems), or at $\bot$ and only generalize (the *specific-to-general systems*). The function *Delete* determines the actual search strategy: when *Delete* is first-in-first-out, one obtains a breadth-first algorithm, when it is last-in-first-out, one obtains a depth-first algorithm, and when it deletes the best hypothesis (according to some criterion or heuristic), one obtains a best-first algorithm. The operator ρ determines the size and nature of the refinement steps taken through the search space. The function *Stop* determines when the algorithm halts. As argued at the start of this chapter, some algorithms compute *all* elements, k elements or an approximation of an element satisfying $\mathcal{Q}$. If all elements are desired, *Stop* equals Queue=$\emptyset$; when k elements are sought, it is $\mid Th \mid= k$. Finally, some algorithms *Prune* candidate hypotheses from the Queue. Two basic types of pruning exist: *heuristic* pruning, which prunes away those parts of the search space that appear to be uninteresting, and *sound* pruning, which prunes away those parts of the search space that cannot contain solutions.

As with other search algorithms in artificial intelligence, one can distinguish *complete* algorithms from *heuristic* ones. *Complete* algorithms compute all elements of $Th(\mathcal{Q}, D, \mathcal{L}_h)$ in a systematic manner. On the other hand, *heuristic* algorithms aim at computing one or a few hypotheses that score best w.r.t. a given heuristic function. This type of algorithm does not guarantee that the best hypotheses are found.

In the next few subsections, we present a number of instantiations of our generic algorithm. This includes: a complete general-to-specific algorithm in Sect. 3.11, a heuristic general-to-specific algorithm in Sect. 3.12, a branch-and-bound algorithm for finding the top k hypotheses in Sect. 3.13, and a specific-to-general algorithm in Sect. 3.14. Afterward, a further (advanced) section on working with borders is included, before concluding this chapter.

3.11 A Complete General-to-Specific Algorithm

We now outline a basic one-directional complete algorithm that proceeds from general to specific. It discovers all hypotheses that satisfy an anti-monotonic quality criterion $\mathcal{Q}$. It can be considered an instantiation of the generic algorithm, where

- *Init*= $\{\top\}$;
- *Prune*(Queue)= $\{h \in \text{Queue} \mid \mathcal{Q}(h, D) = false\}$,
- *Stop*=(Queue= $\emptyset$), and
- ρ is an optimal refinement operator.

Furthermore, various instantiations of *Delete* are possible.

Example 3.26. Reconsider Ex. 3.3 and assume Algo. 3.3 employs a breadth-first search strategy (obtained by setting *Delete* to first-in-first-out). Then the algorithm traverses the search tree shown in Fig. 3.9 in the order indicated.

Observe that for anti-monotonic quality criteria, Algo. 3.3 only prunes hypotheses that cannot be a solution, and whose children cannot be a solution either. Observe also that the use of an optimal refinement operator is, strictly speaking, not necessary for the correctness of the algorithm but is essential for its efficiency. Indeed, if an ideal operator would be employed instead, the same hypotheses would be generated over and over again.

There exists also a dual algorithm, that searches from specific to general and applies generalization rather than specialization.

Algorithm 3.3 A complete general-to-specific algorithm

```
Queue := {$\top$};
Th := $\emptyset$;
while not $Queue = \emptyset$ do
  $Delete\ h$ from Queue
  if $\mathcal{Q}(h,D)$ = true then
    add $h$ to $Th$
    Queue := Queue $\cup \rho_o(h)$
  end if
end while
return Th
```

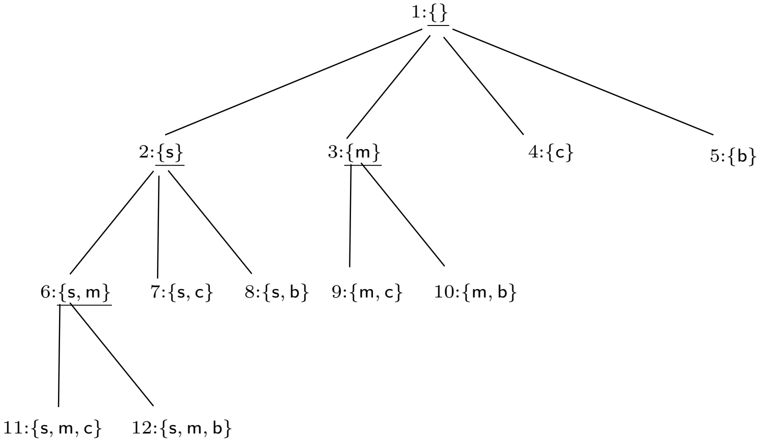


Fig. 3.9. Illustrating complete search w.r.t. an anti-monotonic predicate

Exercise 3.27. Describe the dual algorithm and illustrate it at work on the same data set and hypothesis language, but now use the constraint $(freq(h, D) \leqslant 2)$.

3.12 A Heuristic General-to-Specific Algorithm

The complete algorithm works well provided that the quality criterion is anti-monotonic or monotonic. However, there exist many interesting mining and learning tasks for which the quality criterion is neither monotonic nor anti-monotonic. Furthermore, one might not be interested in all solutions but perhaps in a single best solution or an approximation thereof. In such cases, it is too inefficient to perform a complete search because the pruning properties no longer hold. Therefore, the only resort is to employ a heuristic function f in a greedy algorithm. Such an algorithm is shown in Algo. 3.4.

The algorithm again works from general to specific and keeps track of a Queue of candidate solutions. It repeatedly selects the best hypothesis h from

Queue (according to its heuristic) and tests whether it satisfies $\mathcal{Q}$. If it does, the algorithm terminates and outputs its solution; otherwise, it continues by adding all refinements (using an ideal operator) to the Queue. The Queue is typically also *Pruned*.

Again, the algorithm can be viewed as an instantiation of Algo. 3.10. The following choices have been made:

- *Delete* selects the best hypothesis,
- *Stop*=$\mid Th \mid= 1$,
- an ideal refinement operator is employed,
- *Prune* depends on the particular instantiation. Very often *Prune* retains only the best k hypotheses, realizing a beam search.

Note that – because of the use of a heuristic and a greedy search strategy – it is essential that an ideal operator is being used. Greedy algorithms focus on the currently most interesting nodes and prune away the others. Should an optimal refinement operator be used instead of an ideal one, the direct neighborhoods of the nodes of interest would not be fully explored.

Algorithm 3.4 A heuristic general-to-specific algorithm

```
Queue := {⊤};
Th := ∅;
while Th = ∅ do
  Delete the best h from Queue
  if $\mathcal{Q}(h,D)$ = true then
    add h to Th
  else
    Queue := Queue $\cup \rho_i(h)$
  end if
  Queue := Prune(Queue)
end while
return Th
```

Example 3.28. Let $P = \{\{\mathsf{s},\ \mathsf{m},\ \mathsf{b}\}, \{\mathsf{s},\ \mathsf{b},\ \mathsf{c}\}\}$ and $N = \{\{\mathsf{s},\ \mathsf{m}\}, \{\mathsf{b},\ \mathsf{c}\}\}$ be the data sets; assume that the heuristic function used is $m(h, P, N)$ and that the quality criterion is true if $m(h, P, N) > m(h', P, N)$ for all $h' \in \rho_i(h)$. The $m(h, P, N)$ function is a variant of the accuracy defined in Eq. 3.8:

$$m(h, P, N) = \frac{freq(h, P) + 0.5}{freq(h, P) + freq(h, N) + 1} \tag{3.30}$$

The m function is used instead of *acc* to ensure that when two patterns have equal accuracy, the one with the higher coverage is preferred. Assume also that a beam search with $k = 1$, that is, hill climbing, is used. This results in the search tree illustrated in Fig. 3.12. The nodes are expanded in the order indicated.

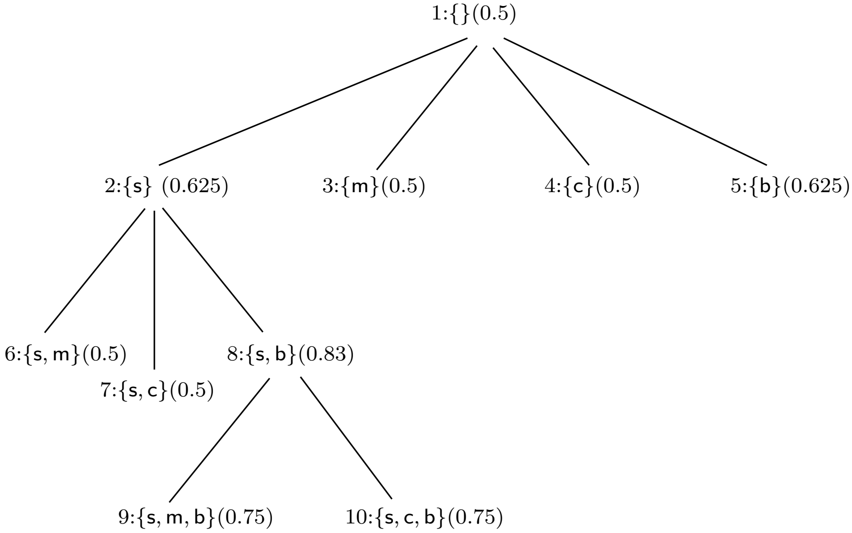


Fig. 3.10. Illustrating heuristic search

3.13 A Branch-and-Bound Algorithm

For some types of problem, a combination of the previous two algorithms – branch-and-bound – can be used. A *branch-and-bound* algorithm aims at finding the best hypothesis (or, best k hypotheses) w.r.t. a given function f, that is,

$$\mathcal{Q}(h, D, \mathcal{L}_h) = \big(h = \arg\max_{h' \in \mathcal{L}_h} f(h')\big) \tag{3.31}$$

Furthermore, branch-and-bound algorithms assume that, when working from general to specific, there is a bound $b(h)$ such that

$$\forall h' \in \mathcal{L}_h : h \preceq h' \rightarrow b(h) \geqslant f(h') \tag{3.32}$$

Given the current best value (or current k best values) v of the hypotheses investigated so far, one can safely prune all refinements of h provided that $v \geqslant b(h)$.

The branch-and-bound algorithm essentially combines the previous two algorithms: it performs a complete search but selects the hypotheses greedily (according to their f values) and prunes on the basis of the bounds. Furthermore, it computes a single best hypothesis (or k best hypotheses) as in the heuristic algorithm.

Example 3.29. Consider the function $f(h) = \mathit{freq}(h, P) - \mathit{freq}(h, N)$. The quality criterion aims at finding patterns for which the difference between the frequency in P and in N is maximal. For this function $f(h)$, the bound

$b(h) = freq(h, P)$ satisfies the requirement in Eq. 3.32 because specializing a hypothesis can only decrease the frequencies $freq(h, P)$ and $freq(h, N)$. Thus the maximum is obtained when $freq(h, P)$ remains unchanged and $freq(h, N)$ becomes 0.

The resulting search tree, applied to P and N of Ex. 3.20, is shown in Fig. 3.13. The values for the hypotheses $(b(h); f(h))$ are shown for each node. The order in which the nodes are visited is indicated. Nodes 3, 4 and 6 are pruned directly after generation and node 7 is pruned after node 8 has been generated. Finally, since node 5 cannot be further expanded, node 8 contains the optimal solution.

Let us also stress that branch-and-bound algorithms are used with statistical functions such as entropy and chi-square; cf. [Morishita and Sese, 2000].

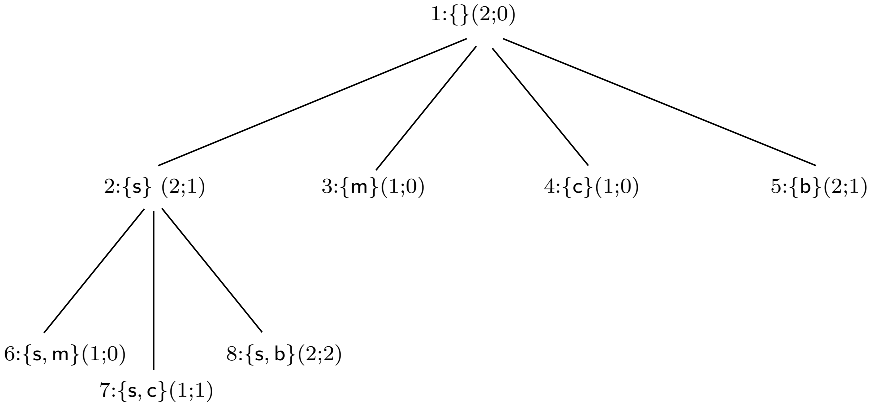


Fig. 3.11. Illustrating a branch-and-bound algorithm

Exercise 3.30. Specify the branch-and-bound algorithm formally.

3.14 A Specific-to-General Algorithm

To illustrate a specific-to-general algorithm, we consider an algorithm that implements a cautious approach to generalization in this section. Assume that the goal is to find the minimal generalizations of a set of hypotheses (or positive examples). The quality criterion can be specified as

$$\mathcal{Q}(h, D) = \big(h \in max(\{h \in \mathcal{L}_h \mid \forall d \in D : h \preceq d\}\big) \tag{3.33}$$

One possible algorithm to compute $Th(\mathcal{Q}, D, \mathcal{L}_h)$ is shown in Algo. 3.5. It starts from $\bot$ and repeatedly generalizes the present hypotheses until all examples or hypotheses in D are covered. Observe that the algorithm is an instance of the generic algorithm Algo. 3.10 (it is left as an exercise to the reader to verify this), and also that the efficiency of the algorithm can be improved by, for instance, incrementally processing the examples in D. This would eliminate our having to test the overall criterion $\mathcal{Q}$ over and over again.

Algorithm 3.5 A cautious specific-to-general algorithm

```
Queue := {$\bot$};
Th := $\emptyset$;
while Queue $\neq \emptyset$ do
   Delete a hypothesis $h$ from Queue
   if $\mathcal{Q}(h,D)$ = true then
      add $h$ to $Th$
   else
      select a hypothesis $d \in D$ such that $\neg(h \preceq d)$
      Queue := Queue $\cup$ $mgg(h, d)$
   end if
end while
return Th
```

Example 3.31. Consider the data set $D = \{\, \mathsf{s} \wedge \mathsf{m} \wedge \mathsf{c}, \mathsf{s} \wedge \mathsf{m} \wedge \mathsf{b}, \mathsf{s} \wedge \mathsf{m} \wedge \mathsf{c} \wedge \mathsf{b}\}$. When the examples are processed from right to left, the following hypotheses are generated: $\bot = false$, $\mathsf{s} \wedge \mathsf{m} \wedge \mathsf{c} \wedge \mathsf{b}$, $\mathsf{s} \wedge \mathsf{m} \wedge \mathsf{b}$, and $\mathsf{s} \wedge \mathsf{m}$.

Exercise 3.32. Design a data set in the domain of the strings where the S set (returned by Algo. 3.5) is not a singleton.

3.15 Working with Borders*

Algo. 3.5 can also be regarded as computing the S set w.r.t. the anti-monotonic constraint $rfreq(h, D) \geqslant 1$, which raises the question of how to compute, exploit and reason with border sets. This will be addressed in the present section.

3.15.1 Computing a Single Border

First, observe that a single border (either the S or G set w.r.t. an anti-monotonic or a monotonic constraint) can be computed in two dual ways: general-to-specific or specific-to-general using a simple adaptation of Algo. 3.3. To illustrate this point, assume that the goal is to find the G set w.r.t. a monotonic criterion $\mathcal{Q}$ and that we work from general to specific. Two modifications are needed to Algo. 3.3 for addressing this task (shown in Algo. 3.6): 1) refine only those hypotheses that do not satisfy the quality criterion $\mathcal{Q}$, and 2) move only those elements to Th that effectively belong to G. W.r.t.

1), note that if a hypothesis is a member of the G set, then all of its (proper) specializations satisfy $\mathcal{Q}$ even though they cannot be maximally general and therefore do not belong to G. W.r.t. 2), note that it is possible to test whether a hypothesis h that satisfies $\mathcal{Q}$ is maximally general by computing $\rho_i'(h)$ and testing whether elements of $\rho_i'(h)$ satisfy $\mathcal{Q}$. Only if none of them satisfies $\mathcal{Q}$can one conclude that $h \in G$. Here, ρ_i' denotes an ideal generalization operator, even though the general direction of the search is from general to specific.

Algorithm 3.6 Computing the G border general-to-specific.

```
Queue := {$\top$};
Th := $\emptyset$;
while not $Queue = \emptyset$ do
  Delete $h$ from Queue
  if $\mathcal{Q}(h,D)$ = true and $h \in G$ then
    add $h$ to $Th$
  else if $\mathcal{Q}(h,D)$ = false then
    Queue := Queue $\cup \rho_o(h)$
  end if
end while
return Th
```

The dual algorithm for computing G from specific to general can be obtained by starting from $\bot$ and by generalizing only those hypotheses h that satisfy Q (and do not belong to G). Even though the two algorithms compute the same result, the efficiency with which they do so may vary significantly. The direction that is to be preferred typically depends on the application.

By exploiting the dualities, one can devise algorithms for computing S as well as the negative borders.

Exercise 3.33. Compute the S set for the problem of Ex. 3.3 using the general-to-specific algorithm.

3.15.2 Computing Two Borders

Second, consider computing the borders of a version space as illustrated in Fig. 3.8. This could be the result of a quality criterion $\mathcal{Q}$ that is the conjunction of an anti-monotonic and a monotonic predicate. To compute these borders, we can proceed in several ways. One of these first computes one border (say the S set) using the techniques sketched above and then uses that set to constrain the computation of the other border (say the G set). When searching from general to specific for the G set and with the S set already given, hen all hypotheses h that are not more general than an element in the S set can safely be pruned in Algo. 3.6. By exploiting the various dualities, further algorithms can be obtained.

Example 3.34. Suppose S=$\{\mathsf{s} \wedge \mathsf{m} \wedge \mathsf{c}\}$. When computing G from general to specific, b and all its refinements can be pruned because $\neg(\mathsf{b} \preceq \mathsf{s} \wedge \mathsf{m} \wedge \mathsf{c})$.

3.15.3 Computing Two Borders Incrementally

Third, suppose that one is given already a version space (characterized by its S and G sets) and the task is to update it in the light of an extra constraint. This is the setting for which the original theory of version spaces was developed by Tom Mitchell. He considered concept-learning, which is concerned with finding all hypotheses satisfying $rfreq(h,P) = 1 \wedge rfreq(h,N) = 0$ w.r.t. sets of positive and negative examples P and N. This criterion can be rewritten as:

$$p_1 \in \mathbf{c}(h) \wedge \ldots \wedge p_k \in \mathbf{c}(h) \wedge n_1 \not\in \mathbf{c}(h) \wedge ... \wedge n_l \not\in \mathbf{c}(h) \tag{3.34}$$

where the p_i and the n_j are the members of P and N, respectively. Mitchell's candidate elimination algorithm processed the examples incrementally, that is, one by one, by updating S and G to accommodate the new evidence. His algorithm is shown in Algo. 3.7.

Algorithm 3.7 Mitchell's candidate elimination algorithm

```
$S := \{\bot\}$ ; $G := \{\top\}$
for all examples $e$ do
  if $e \in N$ then
    $S := \{s \in S \mid e \in \mathbf{c}(s)\}$
    for all $g \in G : e \in \mathbf{c}(g)$ do
      $\rho_g := \{g' \in ms(g,e) \mid \exists s \in S : g' \preceq s\}$
      $G := G \cup \rho_g$
    end for
    $G := min(G)$
  else if $e \in P$ then
    $G := \{g \in G \mid e \not\in \mathbf{c}(g)\}$
    for all $s \in S : e \not\in \mathbf{c}(s)$ do
      $\rho_s := \{s' \in mgg(s,e) \mid \exists g \in G : g \preceq s'\}$
      $S := S \cup \rho_s$
    end for
    $S := max(S)$
  end if
end for
```

The algorithm employs a new operation $ms(g,e)$, the minimal specialization w.r.t. e:

$$ms(g,e) = min(\{g' \in \mathcal{L}_h \mid g \preceq g' \wedge e \not\in \mathbf{c}(g')\}) \tag{3.35}$$

Example 3.35. Applied to item-sets over $\mathcal{I}$, this yields

$$ms(M_1, M_2) = \begin{cases} \{M_1\} & \text{if } \neg(M_1 \preceq M_2) \\ \{M_1 \cup \{i\} \mid i \in \mathcal{I} - (M_1 \cup M_2)\} & \text{otherwise} \end{cases} \tag{3.36}$$

For instance, $ms(\mathsf{s}, \mathsf{s} \wedge \mathsf{m}) = \{\mathsf{s} \wedge \mathsf{c}, \mathsf{s} \wedge \mathsf{b}\}$ and $ms(\mathsf{s}, \mathsf{s} \wedge \mathsf{m} \wedge \mathsf{b}) = \{\mathsf{s} \wedge \mathsf{s}\}$.

The candidate elimination algorithm works as follows. It starts by initializing the S and G sets to the $\bot$ and $\top$ elements, respectively. It then repeatedly updates these sets whenever the next example is not handled correctly by all the elements of S and G. Let us now sketch the different steps for a positive example (the ones for a negative one are dual). Whenever an element g of G does not cover the positive example e, the element g is too specific and is pruned away. This is because in order to cover e, the hypothesis g should be generalized, but this is not allowed since it would yield hypotheses that lie outside the current version space. Secondly, whenever an element s of S does not cover the positive example e, the mgg operator is applied on the elements g and e, yielding a set $mgg(e, g)$ of minimally general generalizations. From this set, only those elements are retained that lie within the version space, that is, those that are more specific than an element of G. Finally, only the maximal elements of S are retained in order to obtain a proper border and to remove redundancies. Without this step, the algorithm still works correctly in the sense that all elements of the version space will lie between an element of S and G. It may only be that some elements are redundant and do not lie at the proper border.

Example 3.36. Let us now employ the candidate elimination algorithm to the sets of examples $P = \{\{\mathsf{s}, \mathsf{m}, \mathsf{b}\}, \{\mathsf{s}, \mathsf{c}, \mathsf{b}\}\}$ and $N = \{\{\mathsf{b}\}, \{\mathsf{b}, \mathsf{c}\}\}$. So, the resulting S and G form the answer to Exer. 3.20. When first processing the positive examples and then the negative ones, the following sequence of S and G sets is obtained:

S=$\{\bot\}$	G=$\{\top\}$
S=$\{\mathsf{s} \wedge \mathsf{m} \wedge \mathsf{b}\}$	G=$\{\top\}$
S=$\{\mathsf{s} \wedge \mathsf{b}\}$	G=$\{\top\}$
S=$\{\mathsf{s} \wedge \mathsf{b}\}$	G=$\{\mathsf{s}\}$

Exercise 3.37. What happens to S and G when the examples are processed in a different order? Process the negative examples before the positive ones.

The use of version space representations for concept learning has some interesting properties. When the S and G sets become identical and contain a single hypothesis, one has converged upon a single solution, and when S or G becomes empty, no solution exists. Finally, the intermediate borders obtained using an incremental algorithm (such as the candidate elimination algorithm) can be used to determine whether a hypothesis h can still belong to the solution space. Furthermore, when learning concepts, the intermediate borders can be used to determine which examples contain new information. Indeed, under the assumption that a solution to the concept learning task exists within $\mathcal{L}_h$, any example covered by all elements in S must be positive, and any example covered by no element in G must be negative.

Exercise 3.38. * When $\mathcal{L}_h = \mathcal{L}_e$, the constraint $e \in \mathbf{c}(h)$ can often be rewritten as $h \preceq e$, and the dual one, $e \not\in \mathbf{c}(h)$, as $\neg(h \preceq e)$, where h is the target hypothesis and e a specific positive or negative example. Consider now the dual constraints $e \preceq h$ and $\neg(e \preceq h)$. Are these constraints monotonic or anti-monotonic? Also, can you illustrate the use of these constraints and compute the corresponding version space? Finally, can you extend the candidate elimination algorithm to work with these constraints?

Exercise 3.39. Try to learn the $\mathsf{father/2}$ predicate (in a relational learning setting) in the following context. Let the background theory B be the set of facts

$$\begin{array}{ll} \mathsf{male(luc)} \leftarrow & \mathsf{parent(luc, soetkin)} \leftarrow \\ \mathsf{female(lieve)} \leftarrow & \mathsf{parent(lieve, soetkin)} \leftarrow \\ \mathsf{female(soetkin)} \leftarrow & \end{array}$$

the hypothesis language $\mathcal{L}_h$ consist of single clauses, $\mathcal{L}_h = \{\mathsf{father(X,Y)} \leftarrow body \mid body \subseteq \{\mathsf{parent(X,Y)}, \mathsf{parent(Y,X)}, \mathsf{male(X)}, \mathsf{female(X)}, \mathsf{female(Y)}\}\}$, and let $P = \{\mathsf{father(luc, soetkin)}\}$, and $N = \{\mathsf{father(lieve, soetkin)}, \mathsf{father(luc, luc)}\}$

The candidate elimination algorithm illustrates how the borders of a version space can be computed incrementally. The candidate elimination algorithm works with constraints in the form of positive and negative examples. One remaining question is whether one can also devise algorithms that incrementally process a sequence of other types of monotonic and anti-monotonic constraints. One way of realizing this adapts Algo. 3.6. We discuss how to process an extra monotonic constraint $\mathcal{Q}$ w.r.t. to an already existing version space characterized by G and S.

The adaptation works as follows. First, the elements of S that do not satisfy $\mathcal{Q}$ are discarded. Second, the approach of Algo. 3.6 is taken but 1) all hypotheses that are not more general than an element of S are pruned, and 2) only those hypotheses that are more specific than an element of the original G set are tested w.r.t. $\mathcal{Q}$.

3.15.4 Operations on Borders

Another approach to incrementally process a conjunction of monotonic and anti-monotonic constraints intersects version spaces. Version space intersection employs the following operations:

$$int_s(S_1, S_2) = max\big(\{s \mid s \in mgg(s_1, s_2) \text{ with } s_1 \in S_1 \wedge s_2 \in S_2\}\big) \quad (3.37)$$

$$int_g(G_1, G_2) = min\big(\{g \mid g \in mgs(g_1, g_2) \text{ with } g_1 \in G_1 \wedge g_2 \in G_2\}\big) \quad (3.38)$$

The following property, due to Hirsh [1990], holds

Property 3.40. Let VS_1 and VS_2 be two version spaces with border sets S_1, G_2 and S_2, G_2, respectively. Then $VS = VS_1 \cap VS_2$ has border sets $int_s(S_1, S_2)$ and $int_g(G_1, G_2)$ respectively.

Version space intersection can now be used for learning concepts. To realize this, compute the version spaces that correspond to each of the single examples and incrementally intersect them.

Exercise 3.41. Solve the concept learning task in Exer. 3.20 by applying version space intersection.

3.16 Conclusions

This chapter started by formalizing data mining and machine learning tasks in a general way. It then focused on mechanisms for computing the set of solutions $Th(\mathcal{Q}, D, \mathcal{L}_h)$. The search space $\mathcal{L}_h$ was structured using the important *generality* relation. Various quality criteria were proposed and their properties, most notably monotonicity and anti-monotonicity, were discussed. It was shown that these properties impose borders on the set of solutions $Th(\mathcal{Q}, D, \mathcal{L}_h)$, and the notion of a version space was introduced. To compute $Th(\mathcal{Q}, D, \mathcal{L}_h)$ various algorithms were presented. They employ refinement operators which are used to traverse the search space. Ideal operators are especially useful for performing heuristic search and optimal ones for complete search. Some of the algorithms work from general to specific, other ones from specific to general. Finally, some algorithms work with the borders and there are algorithms (such as candidate elimination) that are bidirectional.

3.17 Bibliographical Notes

The formulation of the data mining task using $Th(\mathcal{Q}, D, \mathcal{L}_h)$ is due to Mannila and Toivonen [1997]. The notions of generality and border sets and their use for machine learning are due to Mitchell [1982, 1997] and, in a data mining context, to Mannila and Toivonen [1997]. The material presented here follows essentially the same lines of thought as these earlier works but perhaps presents a more integrated view on data mining and concept learning. The algorithms contained in this section are derived from [Mitchell, 1982, 1997, Mannila and Toivonen, 1997, De Raedt and Kramer, 2001, De Raedt and Bruynooghe, 1992a, Morishita and Sese, 2000]. Many of them belong to the folklore of machine learning and data mining. Algorithm 3.3 has a famous instantiation in the context of item-set and association rule mining; cf. [Agrawal et al., 1993]. Refinement operators were introduced in inductive logic programming in Ehud Shapiro's seminal Ph.D. thesis [1983]. Their properties were later studied by Nienhuys-Cheng and de Wolf [1997]. The notion of an ideal refinement operator introduced has been slightly adapted (w.r.t.

[Nienhuys-Cheng and de Wolf, 1997]) for educational purposes. The notion of an optimal refinement operator in relational learning is due to De Raedt and Bruynooghe [1993].

4

Representations for Mining and Learning

Choosing the right representation to model the problem is essential in artificial intelligence, and machine learning and data mining are not different in this respect. Since the early days of machine learning, researchers such as Gordon Plotkin, Stephen Vere, and Ryszard Michalski have considered the use of expressive representations to model complex learning problems. In this chapter, we introduce a hierarchy of representations used to represent machine learning and data mining problems. The hierarchy includes boolean, attribute-value, multi-instance, multi-relational and program representations. The various representations are formulated using the Prolog programming language and illustrated using various applications. We also show how sequences, trees and graphs can be formulated within the sketched relational and logical representations. Because the relations amongst and the need for the various representations are essential for understanding logical and relational learning, we present an integrated framework addressing these issues. This framework also leads us to advanced issues such as propositionalization and aggregation, the subjects of the last sections of this chapter.

4.1 Representing Data and Hypotheses

In the previous chapter, we employed two representation languages to model machine learning and data mining problems,

- a *language of examples* $\mathcal{L}_e$, whose elements are descriptions of examples, observations or data,
- a *language of hypotheses* $\mathcal{L}_h$, whose elements describe hypotheses about (or regularities within) the examples, observations or data,

and the covers relation $\mathbf{c} : \mathcal{L}_h \times \mathcal{L}_e$ to determine whether a hypothesis matches an example.

This book employs logical representations for representing examples, hypotheses and the covers relation, and studies the implications for machine learning and data mining. However, there exist different possible ways for choosing $\mathcal{L}_h$, $\mathcal{L}_e$ and $\mathbf{c}$ in logic, and they result in different settings for learning. The most popular settings are *learning from entailment* and *learning from interpretations.* Learning from entailment was used in Chapter 1, in the structure-activity relationship prediction problem, whereas learning from interpretations was the setting used throughout the previous chapter, when introducing machine learning and data mining principles. We now introduce these two settings formally.

In learning from entailment, the languages $\mathcal{L}_e$, $\mathcal{L}_h$ are logical formulae, typically subsets of clausal logic, and the covers relation $\mathbf{c}$ corresponds to logical entailment. More formally:

Definition 4.1. *When learning from entailment, $\mathcal{L}_e$ and $\mathcal{L}_e$ are logical formulae and $\mathbf{c}(H, e) = true$ if and only if $H \models e$.*

When working in clausal logic, examples typically correspond to single clauses and hypotheses to sets of clauses. Hence, $\mathcal{L}_e$ is a set of clauses, and $\mathcal{L}_h$ is a set of theories, that is, a set of sets of clauses. We write H using uppercase characters because it corresponds to a set of clauses, and e in lowercase because it denotes a single clause.

Example 4.2. Let us consider the domain of animals and assume that we have a blackbird that flies. This bird could be represented using the following clause e:

$$\mathsf{flies} \leftarrow \mathsf{black}, \mathsf{bird}, \mathsf{hasFeathers}, \mathsf{hasWings}, \mathsf{normal}, \mathsf{laysEggs}$$

Let H now be the theory:

$$\begin{array}{l} \mathsf{flies} \leftarrow \mathsf{bird}, \mathsf{normal} \\ \mathsf{flies} \leftarrow \mathsf{insect}, \mathsf{hasWings}, \mathsf{normal} \end{array}$$

Because $H \models e$, the hypothesis H covers the example e.

An alternative inductive logic programming setting that is frequently employed in data mining, and in computational learning theory as well, is that of *learning from interpretations.* When learning from interpretations, $\mathcal{L}_h$ is a set of logical formulae, most often subsets of clausal logic, and $\mathcal{L}_e$ a set of interpretations (or possible worlds). Furthermore, the covers relation corresponds to satisfiability. More formally:

Definition 4.3. *When learning from interpretations, $\mathcal{L}_e$ is a set of interpretations, $\mathcal{L}_h$ is a set of logical formulae, and $\mathbf{c}(H, e) = true$ if and only if e is a model of H.*

For practical reasons, $\mathcal{L}_e$ is typically restricted to Herbrand interpretations, which is also what we will assume from now on.

Example 4.4. The previous example can be represented using the interpretation e':

$$\{\mathsf{flies}, \mathsf{black}, \mathsf{bird}, \mathsf{hasFeathers}, \mathsf{hasWings}, \mathsf{normal}, \mathsf{laysEggs}\}$$

It is easily verified that this interpretation is a model for the theory H shown in the previous example. Thus the hypothesis H covers the example e' when learning from interpretations.

Although these two ways of representing the information about the bird are both natural, there is a subtle difference in meaning between the two representations. By representing the bird using an interpretation, it is assumed that all propositions not in the interpretation are false. Thus, in the example, the interpretation implies that the proposition insect is known to be false. This assumption is not made using the clausal representation of the bird. A further difference is that in the clausal representation, there is a distinguished predicate, the predicate flies, that is entailed by the set of conditions. In contrast, using interpretations, all predicates are treated uniformly. The former representation can be more natural when learning a specific concept as a predicate definition, such as the concept of flying things; the latter representation is more natural to describe a set of characteristics of the examples, such as the baskets bought in the supermarket.

The choice of which setting to employ in a specific application is often a matter of taste and tradition. There also exist other settings and variants of learning from entailment and from interpretations. Therefore, we employ, throughout this book, different representations and settings in a rather loose manner with the purpose of familiarizing the reader with the various options and conventions that are described in the literature.

In the remainder of this chapter, we investigate different types of data and their corresponding representation languages $\mathcal{L}_e$ and $\mathcal{L}_h$. Some of these data types, such as strings, trees and graphs, are among the best studied ones in computer science. Correspondingly, they possess several natural representations. Nevertheless, in order to enable a uniform and general treatment, we introduce adequate logical representations for these data types.

Many different types of representations are employed by contemporary machine learning and data mining systems. These include: boolean representations, attribute-value representations, multi-instance representations, relational representations, term-structured representations and programs. We now introduce these representations within the common framework of logical and relational learning. While doing so, for ease of explanation, we assume that no background theory is used and also that the hypotheses are (sets of) definite clauses. These restrictions will be lifted later. Because boolean representations were already introduced in Sect. 3.2, we start by introducing attribute-value learning.

4.2 Attribute-Value Learning

By far the most popular class of representation languages for data analysis is that of attribute-value representations. This type of language is employed within data mining, machine learning, neural networks, statistics, etc. Its popularity is to a large extent due to its simplicity and the fact that it can be represented in tabular form. This implies that data mining tools working with these representations can elegantly be coupled to other table-based software such as relational databases and spreadsheets.

Consider Table 4.1, which is due to Quinlan [1986]. It contains information about situations in which the weather is good (positive), or bad (negative) for playing tennis. In this table, each row corresponds to an example, and each column corresponds to an attribute. Furthermore, the examples have exactly one value specified for each of the attributes. In database terminology, this means that examples are *tuples* in a table (or relation). Therefore, the attribute-value representation makes the *single-table single-tuple* assumption. Each attribute A also has a *domain* $d(A)$, which specifies the set of values the attribute can take. In this book, we shall consider the following types of domains:

nominal: the domain consists of a finite set of discrete values,
ordinal: the values of the domain are enumerable and totally ordered,
continuous: the domain is the set of real numbers.

Table 4.1. Quinlan's playtennis example

Outlook	Temperature	Humidity	Windy	Class
sunny	hot	high	no	negative
sunny	hot	high	yes	negative
overcast	hot	high	no	positive
rain	mild	high	no	positive
rain	cool	normal	no	positive
rain	cool	normal	yes	negative
overcast	cool	normal	yes	positive
sunny	mild	high	no	negative
sunny	cool	normal	no	positive
rain	mild	normal	no	positive
sunny	mild	normal	yes	positive
overcast	mild	high	yes	positive
overcast	hot	normal	no	positive
rain	mild	high	yes	negative

It is easy to represent attribute-value problems using logic. Indeed, let us assume that the name of the table is r, the attributes are $A_1, \ldots, A_n$ and their corresponding domains are $d(A_1), \ldots, d(A_n)$. Then each example corresponds to an interpretation of the form $\{r(v_1, \ldots, v_n)\}$ where the v_i are values belonging to the domain $d(A_i)$ of attribute A_i. Alternatively, when learning from entailment, clauses of the form $class(v_n) \leftarrow r(v_1, \ldots, v_{n-1})$ can be used as examples. In the playtennis example, the last row corresponds to

$$\{\mathsf{playtennis(rain, mild, high, yes, negative)}\}$$

and

$$\mathsf{class(neg)} \leftarrow \mathsf{playtennis(rain, mild, high, yes)}.$$

For the playtennis example, the clause

$$\mathsf{class(pos)} \leftarrow \mathsf{playtennis(overcast, Temp, Hum, Wind)}$$

is a conjunctive concept, a single rule. *Disjunctive* hypotheses consist of a set of rules.

Alternatively, when learning from interpretations, one could employ clauses such as

$$\mathsf{Class} = \mathsf{positive} \leftarrow \mathsf{playtennis(overcast, Temp, Hum, Wind, Class)}$$

as constraint. This clause covers all examples in Table 4.1, because all examples in Table 4.1 are models of this clause. The clause thus describes a genuine property of the playtennis illustration. Many data mining and machine learning systems also allow the formulation of simple constraints on attributes. A simple constraint is a condition on a specific attribute. It can, for instance, state that the ith attribute may not be identical to value v, which can be stated as follows:

$$class(Class) \leftarrow r(t_1, \ldots, t_{i-1}, T_i, t_{i+1}, \ldots, t_n), T_i \neq v$$

In a similar fashion, simple constraints can specify that the value of an ordinal attribute should be larger than or smaller than a specified value. In this context, consider the rule

$$\mathsf{adult} \leftarrow \mathsf{person(Age, Sex, Profession)}, \mathsf{Age} \geqslant 18$$

which states that all persons aged over 18 are adults.

Example 4.5. The following disjunctive hypothesis covers all positive and none of the negative examples in the playtennis example:

$$\begin{aligned}
\mathsf{class(pos)} &\leftarrow \mathsf{playtennis(overcast, Temp, Hum, Wind)}\\
\mathsf{class(pos)} &\leftarrow \mathsf{playtennis(rain, Temp, Hum, no)}\\
\mathsf{class(pos)} &\leftarrow \mathsf{playtennis(sunny, Temp, normal, Wind)}
\end{aligned}$$

An alternative and simpler representation of Table 4.1 is possible when learning from entailment. In this representation, each example would be represented as a fact. An example is then covered by a hypothesis if the example is entailed by the hypothesis. Furthermore, the hypotheses could also be represented as facts containing variables.

Example 4.6. Using this representation, the first example could be represented using the fact $\mathsf{playtennis(sunny, hot, high, no)} \leftarrow$, and a hypothesis that covers all positive and no negative example would be

$$\begin{aligned}&\mathsf{playtennis(overcast, Temp, Hum, Wind)} \leftarrow\\&\mathsf{playtennis(rain, Temp, Hum, no)} \leftarrow\\&\mathsf{playtennis(sunny, Temp, normal, Wind)} \leftarrow\end{aligned}$$

While this representation is probably simpler and more intuitive for the playtennis example, it is harder to extend to the more general settings (such as multi-instance learning, the topic of the next section). This explains why we will focus on the other representations when learning from entailment throughout the rest of this chapter.

When employing attribute-value representations, it is often implicitly assumed that each example has exactly one value for each attribute. This assumption can be formalized using the following constraints for each attribute A_i with $d(A_i) = \{v_1, \ldots, v_m\}$:

$$X_i = Y_i \leftarrow r(X_1, \ldots, X_i, \ldots, X_n), r(Y_1, \ldots, Y_i, \ldots, Y_n) \tag{4.1}$$

$$X_i = v_1 \vee \ldots \vee X_i = v_m \leftarrow r(X_1, \ldots, X_i, \ldots, X_n) \tag{4.2}$$

The first constraint states that an example can only have one value for each attribute A_i; the second one that this value belongs to the domain $d(A_i)$. The reader may want to write down the constraints for the playtennis example.

Boolean and attribute-value representations are collectively referred to as *propositional* representations.

Exercise 4.7. How can one represent a boolean learning problem using attribute-value representations? Can one also represent attribute-value learning problems using boolean representations? If so, are the two representations equivalent? Or, is there something that is lost?

4.3 Multiple-Instance Learning: Dealing With Sets

An important extension of attribute-value learning is multi-instance learning. The multi-instance representation is important because it lies at the border of multi-relational learning and propositional learning.

Example 4.8. Assume you are a medical doctor investigating a particular genetic disease and your goal is to identify the genes that regulate this disease. Ultimately, you want to learn a decision rule that will allow you to predict whether a given person will eventually suffer from the disease or not. There is plenty of data available about the disease as many persons have undergone a series of genetic tests. The outcome of these tests for each person has been recorded in attribute-value format. However, the data is imperfect because it is not known exactly which persons may actually suffer from the disease (as this can only be determined for sure by means of expensive and elaborate tests, which have not been carried out on the persons in your investigation). All that is known is that certain families are carriers of the disease and others are not. Furthermore, persons belonging to a carrier family may, but need not necessarily suffer from the disease. Being aware of this problem, the data collectors have collected the data at the family level. There are two classes of families. The first is the carrier type family, where the disease has already occurred in the past, and the second, is the one where the disease has not yet occurred. For each person of the carrier families there is experimental data.

Thinking about the problem you realize that there is an asymmetry in it. Indeed, it is safe to assume that none of the persons belonging to non-carrier families suffers from the disease, whereas for the carrier families it is reasonable to assume that there exists at least one person who suffers from the disease. At this point, you decide it is better to analyze the data at the family level and to classify a person as being at risk with regard to the genetic disease if he or she belongs to a family that is classified as a carrier. The representation you propose describes each person using an attribute-value representation and each family as a set or bag of such persons, and classifies families according to whether they are carriers or not.

The question now is how to represent this type of problem. The main point is that the examples now consist of a set of tuples in a table. In Table 4.2, the examples are families, each of which corresponds to a set of persons, and each person is described by a feature vector. So, a multi-instance example can be represented using a single clause. For instance, the first example would correspond to the clause

$$\mathsf{class(neg)} \leftarrow \mathsf{person(aa, aa, aa, bb), person(aa, aa, aa, aa)}$$

or, alternatively, using the interpretation,

$$\{\mathsf{person(aa, aa, aa, bb), person(aa, aa, aa, aa)}\}.$$

Notice that the number of atoms in the examples for the predicate person does not have to be equal to 1, and also that it may differ according to the example. Thus, multi-instance learning examples are essentially *sets* of tuples. This corresponds to the *single-table multiple-tuple* assumption.

To represent hypotheses, various approaches can be taken. The simplest one is to employ the hypotheses language for attribute-value learning. Continuing Ex. 4.8, a family is classified as carrier if there exists one person (that is, one tuple) in the family for which the query succeeds. Under this assumption, all positive and no negative examples (represented as clauses) in Table 4.2 are covered by

$$\mathsf{class(pos)} \leftarrow \mathsf{person(ab, bb, G3, G4)}.$$

This is the traditional multi-instance learning setting, where examples are classified as positive if one of its instances satisfies some specified conditions. This results in clauses having exactly one literal in their condition part.

Alternatively, more expressive hypothesis languages can be employed. Consider clauses involving two conditions

$$class(Class) \leftarrow r(t_1, ..., t_n), r(u_1, ..., u_n)$$

in which no variable is shared by both literals. Such clauses cover an example if there exist two tuples in the example satisfying the two literals (or one tuple that satisfies both literals). Furthermore, the tuples are independent of one another as no variables are shared amongst the literals. Problems involving hypotheses languages involving two or more of such literals are sometimes called *multi-tuple* learning problems. One might also allow for variables to be shared among the literals in the hypotheses, and hence have tuples that depend on one another. In such cases, one talks about *multi-join* learning.

Finally, let us remark that the multi-instance representation arises quite naturally in some practical situations. Dietterich et al. [1997], who introduced the term multi-instance learning, were motivated by a practical problem in drug discovery. They describe an application where a candidate drug can be in one of a number of conformations and where each conformation is described by an attribute-value tuple. Furthermore, a candidate drug can be either active or inactive. As it can be active on the basis of a single conformation, its natural representation is multi-instance-based. Even though Dietterich et al. [1997] did not employ a relational representation for modeling multi-instance learning problems, the above exposition shows that this can easily be accomplished.

Table 4.2. A multi-instance example

Gene1	Gene2	Gene3	Gene4	Class
aa	aa	aa	bb	negative
aa	aa	aa	aa	
bb	aa	aa	bb	positive
ab	bb	aa	bb	
ab	ab	bb	bb	
ab	ab	bb	aa	
bb	ab	bb	aa	negative
aa	bb	aa	bb	
aa	ab	bb	bb	
ab	bb	bb	bb	positive
aa	bb	bb	aa	
bb	bb	aa	aa	
bb	aa	bb	bb	

Exercise 4.9. Assume that the maximum number of tuples within a multi-instance example is lower than k. Can one then represent a multi-instance learning problem using attribute-value representation? What are the problems associated with this? (The solution to this exercise will be discussed in Sect. 4.11.)

4.4 Relational Learning

From multi-instance learning, it is only a small step to relational learning. Indeed, the only difference is that now *multiple* tables or relations can occur in both examples and hypotheses. Such representations also form the basis of relational databases. This is important from a practical perspective as the data for many applications resides in relational databases. The database view is also important from a more theoretical perspective as it indicates the key advantage of using this type of representation: using relational representations one can elegantly represent complex objects (sometimes called entities) as well as relationships among the objects (as in the well-known Entity-Relationship model in databases).

Example 4.10. Consider the participant, subscription, course and company relations in Figs. 4.1 and 4.3. They contain information about different entities (participants, courses, and companies) involved in a summer school. Indeed, for each of the entities different attributes are contained in the database. For instance, for participants, this includes the job type, the name and whether or not the participant attends the party or not. In addition, two relationships are modeled: the subscription relationship, which indicates which participant takes which course, and the worksfor relationship, which indicates which participant works for which company. Because the former relation is of type **(n:m)**, it is modeled by a separate relation. The subscription relation is of type **(n:m)** because one participant can take multiple (**m**) courses, and vice versa, one course can be taken by multiple (**n**) participants. In contrast, the worksfor relation is of type **(n:1)** because a person can, in our restricted model, work for only one company, but a company can have multiple employees. In such situations, it is usual to model this type of relationship using an extra attribute in table participant.

The presence of **(n:m)** relations makes it very hard to model this type of data using multi-instance or attribute-value representations. Multiple relations are needed to elegantly and compactly represent the information contained in this problem. This is the so-called *multi-table multi-tuple* setting. Examples can again be modeled in various possible ways. Furthermore, this will typically depend on the entities of interest. In the summer school example, is one interested in participants, in courses, in companies, or in the summer school as a whole? In the last case, it is convenient to use a single interpretation (consisting of all tuples in the tables of the database). One can then look for clauses that represent genuine properties of the summer school database. Consider, for example, the clause

$$\mathsf{P} = \mathsf{no} \leftarrow \mathsf{participant}(\mathsf{N},\mathsf{J},\mathsf{C},\mathsf{P}), \mathsf{subscription}(\mathsf{N},\mathsf{C}'), \mathsf{course}(\mathsf{C}',\mathsf{L},\mathsf{advanced}).$$

As the interpretation just described is a model for the clause, the clause reflects a genuine property of the data.

Alternatively, if one is more interested in properties of the individuals or the entities, one could employ clauses of the form

$$\begin{aligned}&\mathsf{attendsParty(adams)} \leftarrow\\ &\qquad \mathsf{participant(adams, researcher, scuf)},\\ &\qquad \mathsf{subscription(adams, erm)},\\ &\qquad \mathsf{subscription(adams, so2)},\\ &\qquad \ldots\end{aligned}$$

Observe that in this type of representation the target attribute Party of participant is projected away as it is now represented by the target predicate attendsParty. Without projecting away the attribute Party, learning the relation attendsParty would be trivial. So, we are actually using the relations in Fig. 4.2.

It is not straightforward how to construct clauses of the above type. For completeness, all facts in the database should be included in the condition part of the clause. This is usually not done because it is impractical. There are two possible alternatives. First, syntactic restrictions on the clauses can be imposed such that only the information relevant to the example is included. This corresponds to imposing a *syntactic bias* on the examples. In the summer school illustration, the following clause for blake could be employed:

$$\begin{aligned}&\mathsf{attendsParty(blake)} \leftarrow\\ &\qquad \mathsf{participant(blake, president, jvt)},\\ &\qquad \mathsf{subscription(blake, erm)},\\ &\qquad \mathsf{subscription(blake, cso)},\\ &\qquad \mathsf{course(cso, 2, introductory)},\\ &\qquad \mathsf{course(erm, 3, introductory)},\\ &\qquad \mathsf{company(jvt, commercial)}.\end{aligned}$$

This clause includes only those tuples that are directly concerned with the participant blake, her company, the subscriptions she takes and the corresponding courses. Secondly, the globally relevant facts (such as those concerning the courses and the companies) could be assembled in the background knowledge, as will be explained in more detail in Sect. 4.9. One clause covering the example blake is $\mathsf{attendsParty(N)} \leftarrow \mathsf{participant(N, president, C)}$. So, typically, examples as well as hypotheses in multi-relational representations consist of both multiple relations and multiple tuples (or atoms).

Example 4.11. Consider the Bongard problem listed in Fig. 4.4. Bongard problems contain six positive and six negative examples, and the task is to find a description of the underlying concept. Can you guess what the concept is in Fig. 4.4? The different scenes in the Bongard problem can be represented using a relational representation. Indeed, two of the leftmost scenes can be represented using the following clauses:

$$\begin{aligned}&\mathsf{pos} \leftarrow \mathsf{circle(c), triangle(t), in(t, c)}\\ &\mathsf{pos} \leftarrow \mathsf{circle(c1), triangle(t1), in(t1, c1), triangle(t2)}\end{aligned}$$

So, the idea is to name the objects by constants and to explicitly represent the relations that hold among them. Furthermore, a hypothesis that covers all positive examples and no negative example is

Name	Job	Company	Party
adams	researcher	scuf	no
blake	president	jvt	yes
king	manager	pharmadm	no
miller	manager	jvt	yes
scott	researcher	scuf	yes
turner	researcher	pharmadm	no

(a) participant

Name	Course
adams	erm
adams	so2
adams	srw
blake	cso
blake	erm
king	cso
king	erm
king	so2
king	srw
miller	so2
scott	erm
scott	srw
turner	so2
turner	srw

(b) subscription

Course	Length	Type
cso	2	introductory
erm	3	introductory
so2	4	introductory
srw	3	advanced

(c) course

Company	Type
jvt	commercial
scuf	university
pharmadm	university

(d) company

Fig. 4.1. The summer school database (adapted from [De Raedt et al., 2001])

Name	Job	Company
adams	researcher	scuf
blake	president	jvt
king	manager	pharmadm
miller	manager	jvt
scott	researcher	scuf
turner	researcher	pharmadm

(a) person

Name
blake
miller
scott

(b) attendsParty

Fig. 4.2. A variant of the summer school database

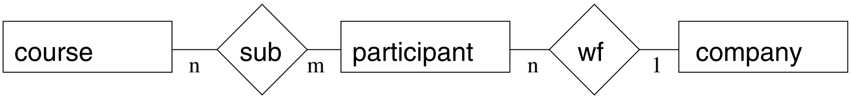


Fig. 4.3. The Entity-Relationship model for the summer school database

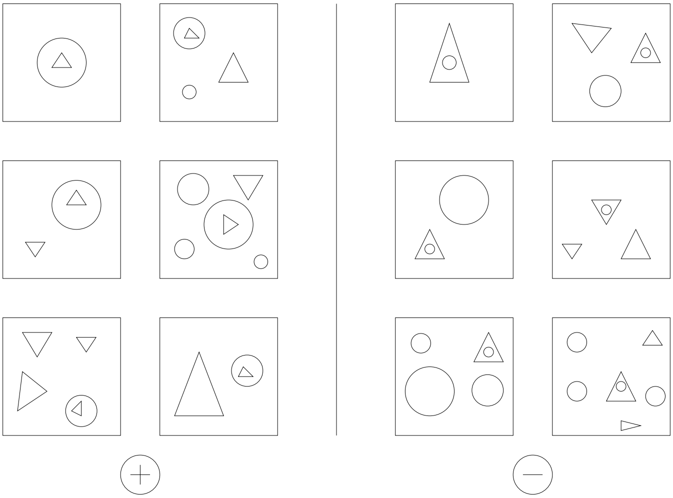


Fig. 4.4. A Bongard problem after [Bongard, 1970]. Reprinted with permission from [Van Laer and De Raedt, 2001]

$$\mathsf{pos} \leftarrow \mathsf{circle}(\mathsf{C}), \mathsf{triangle}(\mathsf{T}), \mathsf{in}(\mathsf{T}, \mathsf{C}).$$

Finally, let us remark that even though Bongard problems are essentially toy problems, they share many characteristics with real-life problems, for instance, for analyzing molecules; cf. Ex. 4.22.

Exercise 4.12. Represent the famous train problem sketched in Fig. 4.5 using a relational representation.

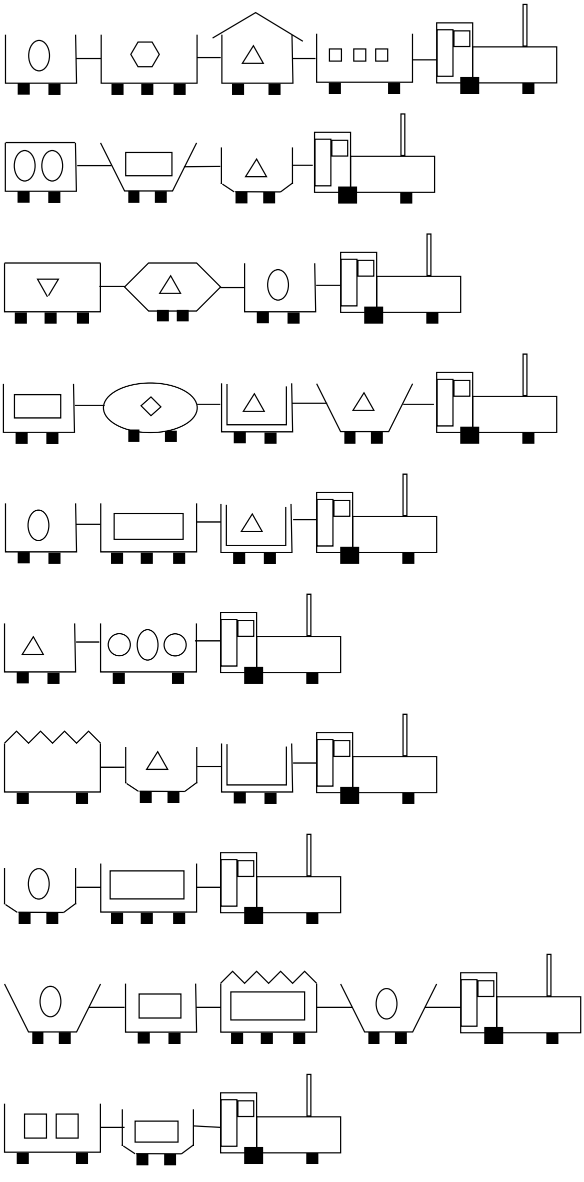


Fig. 4.5. The train problem of Stepp and Michalski [1986]. Reprinted from [Stepp and Michalski, 1986], page 53, ©1986, with permission from Elsevier

4.5 Logic Programs

Since the early days of computer science, programmers have dreamed of intelligent computers that would not have to be programmed by hand but that somehow could be automatically programmed. This question has been studied within the field of *automatic programming.* One approach pioneered by researchers such as Alan Bierman, Philip Summers and Ehud Shapiro has studied *program synthesis from examples.* Here, the program synthesizer or learning system is given examples of the input-output behavior of the target program and has to define the target program. If one chooses the logic programming language Prolog as the representation language for examples and programs, then inductive logic programming can be regarded as an approach to program synthesis from examples. In the early 1990s, this setting received quite some attention within the field of inductive logic programming.

Example 4.13. Consider the facts $\mathsf{sort}([1,3,4],[1,3,4]) \leftarrow$, $\mathsf{sort}([3,5,4],[3,4,5]) \leftarrow$ and $\mathsf{sort}([5,4],[4,5]) \leftarrow$ as positive examples in the learning from entailment setting. A program (implementing insertion sort) in the form of a set of definite clauses that covers all these examples is:

```
insertsort(List, Sorted) ← isort(List, [], Sorted)
isort([], Acc, Acc) ←
isort([H|T], Acc, Sorted) ← insert(H, Acc, NAcc), isort(T, NAcc, Sorted)

insert(X, [], [X]) ←
insert(X, [Y|T], [Y|NT]) ← X > Y, insert(X, T, NT)
insert(X, [Y|T], [X, Y|T]) ← X =< Y.
```

This illustration shows that, in principle, the learning from entailment setting can be used for synthesizing programs from examples. Positive examples then describe the desired input-output behavior of the unknown target program, and negative ones specify a wrong output for a given input. As a consequence of this view, many of the techniques developed in this book are relevant for program synthesis, despite the fact that the dream of automated programming has, so far, not become a reality. One reason why program synthesis from examples is hard is that the typical program consists of a number of subroutines. For instance, the program $\mathsf{insertsort}$ employs a number of auxiliary predicates such as insert and isort. To learn the top-level program, one must also learn the subroutines, and at the same time, in a real-life setting, one cannot expect examples for these intermediate predicates or routines to be provided. Another reason for the difficulty of synthesizing programs from examples is that the space of possible programs is enormous. At the same time, the number of examples the user needs to provide in any realistic setting must be small. These are contradictory requirements. Issues related to program synthesis from examples will be explored further in Chapter 7 that addresses theory revision, which is also concerned with learning auxiliary and intermediate predicates. The Model Inference System, developed in the early 1980s by Ehud Shapiro, will be employed there. It should be regarded as an inductive logic programming system avant la lettre.

Another point to stress is that the key difference between the relational representation used in the previous section and the logic program representation used for synthesizing programs is that the latter allows for the use of functors and structured terms. This difference is akin to that between a database and a programming perspective. Without functors and structured terms, one would not have the expressive power of a programming language (or Turing machine). At the same time, using functors and structured terms in an unrestricted fashion is problematic for most inductive logic programming techniques for a variety of reasons. One is that there is a combinatorial explosion in the search space; another is concerned with the undecidability of coverage tests. Even though theoretical aspects of functors and structured terms are relatively well understood, one often has to restrict their use for pragmatic reasons.

In the following three sections, we provide illustrations of specific situations where functors and structured terms can be helpful. At the same time, these three sections are concerned with traditional data structures and types, in particular, strings and sequences, trees and graphs and networks. For these three cases, we shall show how logic-programming-based representations can be used to elegantly capture these data structures.

4.6 Sequences, Lists, and Grammars

Many observations can be naturally represented as strings or sequences. This is particularly true in computational biology databases, where proteins, genes, and DNA are typically represented using strings. Also, natural language and streams of data (for instance, alarms that arise in a network) are strings of symbols. As already stated in Chapter 3, a *string* $s_1 s_2 ... s_n$ is a sequence of symbols s_i taken from an alphabet Σ. A *language* $\mathcal{L}$ is then a set of strings over an alphabet Σ. For instance, the string 10101000 over the alphabet $\Sigma = \{0, 1\}$ represents a number in binary notation.

Because of their simplicity and applicability, strings and sequences have been widely studied in computer science and many results are known. The most frequently used representation to define and manipulate formal languages, that is, sets of strings, is that of grammars. Various types of grammars are known, and in this book, we concentrate on two of them: the context-free and the definite clause grammars. Roughly speaking, grammars employ two disjoint sets of symbols: the set of terminal symbols Σ (that is, the alphabet of the formal language the grammar defines), and the set of non-terminal symbols N. Furthermore, one non-terminal symbol is the so-called starting symbol. These symbols are then used to define rules. For context-free grammars, the rules are of the form $n \rightarrow s$ where n is a non-terminal and s is a string over $N \cup \Sigma$.

Example 4.14. The following grammar G uses the terminals $\Sigma = \{\mathsf{the}, \mathsf{dog}, \mathsf{bites}\}$ and the non-terminals $N = \{\mathsf{S}, \mathsf{NP}, \mathsf{VP}, \mathsf{Verb}, \mathsf{Art}, \mathsf{Noun}\}$:

$$\begin{array}{ll}
\mathsf{S} \rightarrow \mathsf{NP\ VP} & \mathsf{Art} \rightarrow \mathsf{the} \\
\mathsf{VP} \rightarrow \mathsf{Verb} & \mathsf{Noun} \rightarrow \mathsf{dog} \\
\mathsf{NP} \rightarrow \mathsf{Art\ Noun} & \mathsf{Verb} \rightarrow \mathsf{bites}
\end{array}$$

The language that this grammar accepts contains the single string the dog bites.

Note that grammar rules are very similar to clauses. The two differences are that the symbol "→" is used instead of "←" and that the order of the symbols on the righthand side of the rules is important. The similarity between clauses and grammar rules is also useful for reasoning with grammars. Indeed, deriving strings (such as the dog bites) from a grammar can be done using a resolution-like mechanism.

Example 4.15. Starting from the grammar G, one can derive the following sequence of rules:

$$\begin{array}{l}
\mathsf{S} \rightarrow \mathsf{NP\ VP} \\
\mathsf{S} \rightarrow \mathsf{Art\ Noun\ VP} \\
\mathsf{S} \rightarrow \mathsf{the\ Noun\ VP} \\
\mathsf{S} \rightarrow \mathsf{the\ dog\ VP} \\
\mathsf{S} \rightarrow \mathsf{the\ dog\ Verb} \\
\mathsf{S} \rightarrow \mathsf{the\ dog\ bites}
\end{array}$$

Given that S is the starting symbol, one can conclude that the string the dog bites is accepted by the grammar G. Furthermore, this is the only string that is accepted by our simple grammar.

The next rule $n \rightarrow b_1 \ldots b_{i-1} a_1 \ldots a_n b_{i+1} \ldots b_m$ in the above sequence of rules is obtained from the rules $n \rightarrow b_1 \ldots b_m$ and $b_i \rightarrow a_1 \ldots a_n$ where b_i is the rightmost non-terminal symbol in $b_1 \ldots b_m$. This closely mirrors the resolution inference rule. Abusing logical notation, we shall write

$$n \rightarrow b_1 \ldots b_m \wedge b_i \rightarrow a_1 \ldots a_n \models n \rightarrow b_1 \ldots b_{i-1} a_1 \ldots a_n b_{i+1} \ldots b_m \quad (4.3)$$

This analogy between clausal logic and grammars can now be exploited for data mining purposes. Assume we are given examples such as S → the dog bites and S → the cat runs; then the grammar G together with the two rules Verb → runs and Noun → cat covers the examples. So, a setting that is very close to learning from entailment is obtained for working with sequences and strings. This implies that the principles underlying the mining of sequences are similar to those of relational data mining.

An alternative for employing grammars to represent formal languages is to employ relational or logical notations. This is illustrated in the following example.

Example 4.16. Reconsider the grammar G introduced in Ex. 4.14. This grammar can be represented in relational form as follows:

$$\begin{array}{ll}
\mathsf{art(P1, P2, the)} \leftarrow \mathsf{succ(P2, P1)} & \mathsf{s(P1, P2)} \leftarrow \mathsf{np(P1, P3), vp(P3, P2)} \\
\mathsf{noun(P1, P2, dog)} \leftarrow \mathsf{succ(P2, P1)} & \mathsf{np(P1, P2)} \leftarrow \mathsf{art(P1, P3, Art), noun(P3, P2, Noun)} \\
\mathsf{verb(P1, P2, bites)} \leftarrow \mathsf{succ(P2, P1)} & \mathsf{vp(P1, P2)} \leftarrow \mathsf{verb(P1, P2, Verb)}
\end{array}$$

where the variables starting with a P denote positions, and atoms such as $\mathsf{vp(P1, P2)}$ succeed when the part of the sentence from position $\mathsf{P1}$ to $\mathsf{P2}$ is a verb phrase (vp). The clause

$$\begin{array}{l}
\mathsf{s(0, 3)} \leftarrow \\
\qquad \mathsf{art(0, 1, the),\ noun(1, 2, dog),\ verb(2, 3, bites),} \\
\qquad \mathsf{succ(1, 0),\ succ(2, 1),\ succ(3, 2).}
\end{array}$$

is covered by the grammar and it represents the sentence $\mathsf{the\ dog\ bites}$. Instead of using a relational formalism, the programming language Prolog can be employed, yielding the following clauses:

$$\begin{array}{ll}
\mathsf{art([the|X], X)} \leftarrow & \mathsf{s(P1, P2)} \leftarrow \mathsf{np(P1, P3), vp(P3, P2)} \\
\mathsf{noun([dog|X], X)} \leftarrow & \mathsf{np(P1, P2)} \leftarrow \mathsf{art(P1, P3), noun(P3, P2)} \\
\mathsf{verb([bites|X], X)} \leftarrow & \mathsf{vp(P1, P2)} \leftarrow \mathsf{verb(P1, P2)}
\end{array}$$

Because of the direct correspondence between this program and the context-free grammar G of Ex. 4.14, this type of logic program is sometimes called a *definite clause grammar.* The example sentence would now be represented as $\mathsf{s([the, dog, bites], [])} \leftarrow$. At this point, the reader familiar with Prolog may want to verify that this example is entailed by the program.

To summarize, it is possible to represent sequential data and hypotheses using relational or first-order logic. Furthermore, even the more traditional grammar representations employ rules which are very close to clausal notation. This implies that techniques and principles for learning relational representations also apply to mining sequential data.

4.7 Trees and Terms

Several applications in domains such as web and document mining as well as bioinformatics involve complex objects as examples. One of the most traditional data structures for representing such examples are trees. The natural representation of trees is as logical terms because there is a one-to-one mapping from trees to terms.

Example 4.17. Natural language corpora often involve examples in the form of parse trees. Some examples are listed in Fig. 4.6. They can be represented using the following clauses or facts for the predicate $\mathsf{parsetree}$:

$$\begin{array}{l}
\mathsf{parsetree(s(np(art(the), noun(dog)), vp(verb(eats), np(article(the), noun(rabbit)))} \leftarrow \\
\mathsf{parsetree(s(np(art(the), noun(cat)), vp(verb(bites)))} \leftarrow
\end{array}$$

The following hypothesis covers the first example but does not cover the second one. It is illustrated in Fig. 4.7.

$$\mathsf{parsetree(s(np(art(A), noun(N1)), vp(verb(V), np(art(A), noun(N2)))} \leftarrow$$

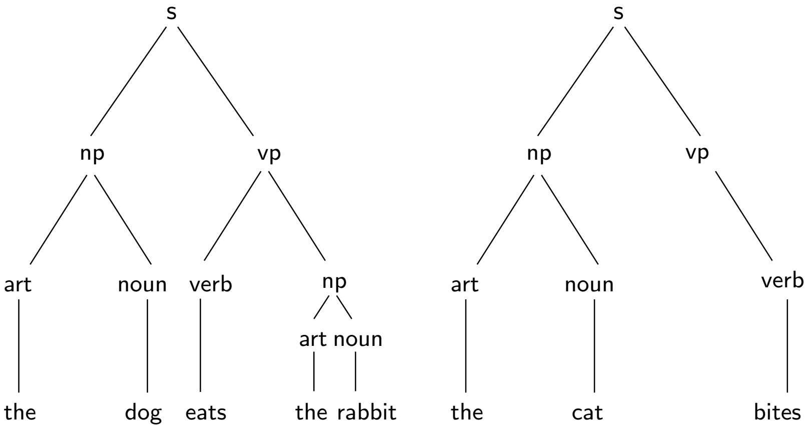


Fig. 4.6. Parse trees as examples

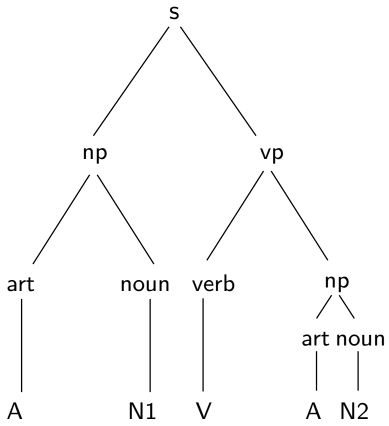


Fig. 4.7. A generalized parse tree

The above example clearly shows that tree-structured examples can naturally be represented using logical terms. One of the advantages of using logical terms for representing trees is that unification can be employed for pattern matching, which allows one to impose restrictions requiring that two subtrees be identical. For example, the hypothesis in Ex. 4.17 only covers examples in which the two articles are identical. One disadvantage of the use of standard logical terms for representing trees is that unification imposes conditions from the root towards the leaves of the trees, which makes it harder to express conditions such as "there exists a subtree noun(dog)" in the tree. Because definite clause logic can be used as a programming language, such hypotheses can still be represented. Indeed, the clause parsetree(P) ← occursin(noun(dog), P) together with an appropriate definition of the predicate occursin/2 covers all parse trees in which noun(dog) occurs. Typically, such additional predicates have to be defined by the user and are part of the background knowledge; cf. Sect. 4.9. Obviously, sequences can be represented using terms as well. This is realized using lists.

Exercise 4.18. Implement the predicate occursin/2 in Prolog.

Exercise 4.19. Represent the example sequences and hypothesis from Ex. 4.14 using terms.

Exercise 4.20. Suppose you have data about sequences of events. How can you represent a sequence as a logical term? Can you represent, using a query of the above type, the concept of sequences in which a certain event e occurs?

Exercise 4.21. Can one represent trees using a relational representation, that is, without using terms? (The solution to this exercise is discussed in Sect. 4.11.)

4.8 Graphs

Another powerful data structure that belongs to the folklore of computer science is that of graphs. Formally, a graph (V, E) consists of a set V of vertices and a set E of edges, where E is a relation over $V \times V$. Several variants exist that take into account labels of edges and vertices. Data in the form of graphs arises in two quite different and yet natural settings. The first setting is where each data point corresponds to a graph, and this setting is typically known as *graph mining*. One quite popular application of graph mining is that of mining sets of molecules. The second setting is where the data set itself is one large graph. As an illustration, consider the world wide web. Each document is a node in the graph and there are links among these nodes. The terms *link analysis* and *link discovery* have recently become popular to describe (aspects of) the second setting [Getoor, 2003]. Because of the importance of and current interest in graph data, let us discuss these settings in more detail and illustrate them using real examples.

Example 4.22. Suppose you are a chemist faced with the problem of predicting which compounds (or molecules) have a certain toxic effect (for instance, carcinogenicity, that is, causing cancer). The structure of the compounds under consideration is diverse but crucial in causing the effect. The two-dimensional structure of a simple molecule, named methane (CH_4), is depicted in Fig. 4.8 on the left. This two-dimensional structure is essentially a graph. If one names the vertices, that is, the atoms, in the graph, a relational representation can be used to represent the graph (cf. the right of Fig. 4.8). For our example molecule, we obtain the following description

$$\begin{array}{ll}
\mathsf{atom(a1, c)} & \mathsf{bond(a1, a2, s)} \\
\mathsf{atom(a2, h)} & \mathsf{bond(a1, a3, s)} \\
\mathsf{atom(a3, h)} & \mathsf{bond(a1, a4, s)} \\
\mathsf{atom(a4, h)} & \mathsf{bond(a1, a5, s)} \\
\mathsf{atom(a5, h)} &
\end{array}$$

for representing the atoms and bonds. The s indicates that there is a single bond among the atoms. Substructures can easily be represented by clauses. For instance, the following substructure requires that there be two atoms of the same type that are bonded to a carbon (c) atom:

$$\begin{array}{l}
\mathsf{substructure} \leftarrow \\
\quad\quad \mathsf{atom(X, c),\ bond(X, Y, T),} \\
\quad\quad \mathsf{atom(Y, T),\ bond(X, Z, W),} \\
\quad\quad \mathsf{atom(Z, T),\ Y \neq Z.}
\end{array}$$

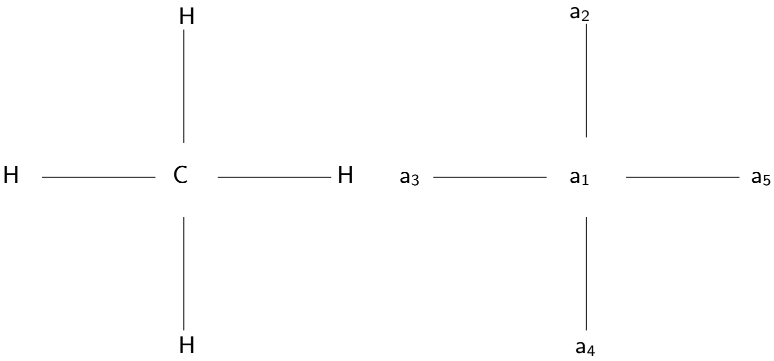


Fig. 4.8. A molecule as a graph

Once the graphs are represented using relations, they can be treated as any other multi-relational data set. Nevertheless, for some purposes, it may still be useful to work with alternative representations for graphs, such as adjacency matrices, directly. Such alternative representations may allow one to perform certain optimizations which may not be available for relational representations.

Extracting information from the web and from documents is becoming quite popular. To illustrate the domain of link analysis and discovery, let us consider the task of analyzing a web site.

Example 4.23. Consider a university web site. It has pages on departments, courses, lecturers, students, projects, etc. These pages can be represented as labeled nodes in a graph. Furthermore, there are hyperlinks between these pages, which correspond to the edges in the graph. In the literature on link analysis and link discovery, one distinguishes several interesting mining tasks. First, there is the task of classifying the nodes in the graph, that is, of finding the labels of the nodes. In our university web site example, the classes would be of type department, course, etc. Second, one can try to predict whether there exists a link (or edge) between two nodes in the graph, and if so, the nature of the underlying relationship. For the university web site, one can distinguish several types of relationships. This includes faculty supervises student, lecturer teaches course, department offers course, etc. Third, one can try to find the authorities and hubs in the network. Hub pages typically contain a lot of links to other pages; they correspond to guides and resource lists. Authorities are important pages that are pointed at by hubs (and other pages). The data and hypotheses for this type of link mining can again easily be represented using relations. The university web site could include relations such as department/1, course/1, lecturer/1, student/1, project/1, supervises/2, teaches/2, offers/2, ...

In addition to web mining, there exist other challenging and important application areas for link discovery. Indeed, consider the task of analyzing a database of scientific papers and authors such as Citeseer or Google Scholar, customers and products at Internet-based companies such as Amazon, and regulatory or protein networks in computational biology. Many of these tasks can again be represented and addressed using relational representations.

Exercise 4.24. Discuss how to represent a database of scientific papers and authors using a relational representation. Identify also interesting link discovery tasks. (Some answers to this question are delivered by Getoor [2003].)

4.9 Background Knowledge

So far, two simplifying assumptions have been made. The first assumption is that the data is available as a set of clauses or interpretations. The second one is that the hypothesis has to be constructed from scratch, that is, that no further knowledge is available to the mining or learning system. Both assumptions are often unrealistic and unpractical. The first assumption only holds after the data have been preprocessed. Typically, however, one starts from a given database such as the summerschool database specified in Fig. 4.1. To overcome the need to preprocess these examples in clauses or interpretations, the typical approach in multi-relational data mining and inductive logic programming uses a different notion of coverage and example. It employs a covers relation that takes into account a body of background knowledge B specified in a language $\mathcal{L}_b$. In the presence of background knowledge, this leads to the following definitions.

Definition 4.25. *When learning from entailment in the presence of background knowledge, $\mathcal{L}_e$ is a set of clauses, $\mathcal{L}_h$ and $\mathcal{L}_b$ are sets of logical formulae, and $\mathbf{c}(H, B, e)$ = true if and only if $H \wedge B \models e$.*

As before, logical theories (sets of clauses) are used as formulae.

Example 4.26. Reconsider the summer school example from Figs. 4.1 and 4.2 and represent the tuples that belong to these relations as facts. For instance, the first tuple of the relation course becomes $\mathsf{course(cso, 2, introductory)} \leftarrow$. Now refer to the ensemble of facts corresponding to the summer school database as B. The observation that blake attends the party can now be represented by the fact $\mathsf{attendsParty(blake)} \leftarrow$. This example is covered by the clause $\mathsf{attendsParty(P)} \leftarrow \mathsf{participant(P, president, C)}$ in the presence of the database B specified in Fig. 4.2 and Fig. 4.1, where we employ the $\mathsf{participant}$ table of Fig. 4.2.

Example 4.27. As another example, reconsider the Bongard problem of Ex. 4.11. To represent the data corresponding to these Bongard problems in a relational database, we apply an identifier or name to each of the example scenes. Instead of using the clause $\mathsf{pos} \leftarrow \mathsf{circle(c1), triangle(t1), in(t1, c1), triangle(t2)}$, we can then employ a database B containing the facts $\mathsf{circle(e2, c1)} \leftarrow$, $\mathsf{triangle(e2, t1)} \leftarrow$, $\mathsf{in(e2, t1, c1)} \leftarrow$, and $\mathsf{triangle(e2, t2)} \leftarrow$, and the fact $\mathsf{pos(e2)} \leftarrow$ as the example clause. Here, $\mathsf{e2}$ is the identifier for the second leftmost scene in the Bongard problem. The reader may want to verify that $\mathsf{pos(e2)} \leftarrow$ is indeed covered by $\mathsf{pos(E)} \leftarrow \mathsf{circle(E, C), triangle(E, T), in(E, C, T)}$.

The key advantage of using the database to store *global* information about all the examples is that this is often easier if one starts from a given relational database. The key disadvantage is, however, that deciding whether a given hypothesis covers an example is computationally more expensive. To see why this is the case, consider that the summer school database contains information about a million participants. To check coverage with respect to one example, one must access the whole database (which also contains information about 999,999 uninteresting examples). This is likely to be computationally more expensive than working with the more *local* representation that was discussed earlier. Using the earlier clausal representation of examples, one can simply retrieve the example and check coverage. Given that the size of this example is much smaller than that of the whole database, this is bound to be more efficient. This type of local approach to representing the examples is also sometimes called an *individual-centered* representation [Flach et al., 1998]. Note also that for some types of problems, such as the Bongard problems and the mining of molecules, the local representations are quite natural. On the other hand, for some applications, such as those concerning link mining and the summer school database, the global representations are more natural as it is unclear how to turn the initial global representation into a local one.

The second assumption that was made implicitly is that the user possesses no background knowledge about the domain of interest. This assumption violates one of the first lessons of artificial intelligence, which states that knowledge is central to intelligence and learning. Therefore, knowledge must be used whenever possible. It is often argued that one of the main advantages of relational representations is that background knowledge is supported in a natural way in the learning or mining process.

True background knowledge, as opposed to the database mentioned above, then takes the form of clauses or rules that define additional predicates or relations. In database terminology, background knowledge closely corresponds to the *intensional* part of the database, and the facts concerning the examples correspond to the *extensional* part.[1] So, one can also consider background predicates as *view* predicates. The idea is that these predicates can then be used as any other predicate in the mining or learning process. This is possible within both learning from entailment and learning from interpretations. This time, however, we illustrate it on the latter setting.

Definition 4.28. *When learning from interpretations in the presence of background knowledge, $\mathcal{L}_e$, $\mathcal{L}_b$ and $\mathcal{L}_h$ are sets of sets of clauses and $\mathbf{c}(H, B, e) =$ true if and only if the minimal Herbrand model $M(e \wedge B)$ is a model of H.*

Example 4.29. Many variants have been developed of the item-set representation presented in Ex. 3.2. One of them concerns the use of a taxonomy to make an abstraction of specific items or brands. Part of such a taxonomy is shown in Fig. 4.9. It can easily be encoded as a set of clauses. For instance, the leftmost branch of the taxonomy corresponds to the clauses:

$$\begin{array}{ll}
\mathsf{product} \leftarrow \mathsf{food} & \mathsf{food} \leftarrow \mathsf{drink} \\
\mathsf{drink} \leftarrow \mathsf{softdrink} & \mathsf{softdrink} \leftarrow \mathsf{coke} \\
\mathsf{coke} \leftarrow \mathsf{pepsi} & \ldots
\end{array}$$

These clauses can now be used to complete examples. For example, the item-set $\{\mathsf{hoegaarden}, \mathsf{duvel}, \mathsf{camembert}\}$, the actual basket containing two famous Belgian beers and French cheese, can now be completed by

$$\{\mathsf{cheese}, \mathsf{drink}, \mathsf{alcohol}, \mathsf{beer}, \mathsf{food}, \mathsf{product}\}$$

in the minimal Herbrand of the item-set. Using the completed examples enables the discovery of association rules at various levels of abstraction, such as $\mathsf{beer} \leftarrow \mathsf{cheese}$. Note that the taxonomy employed need not be tree-structured.

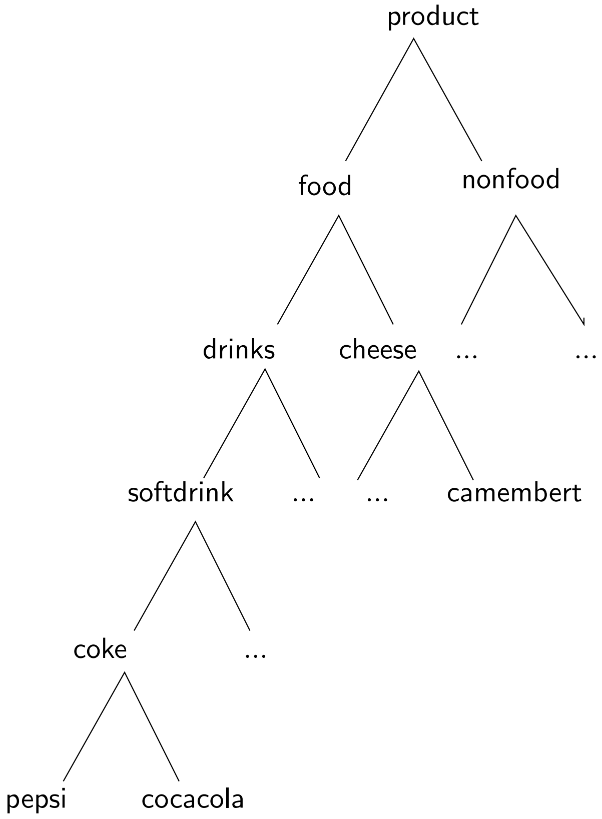


Fig. 4.9. A taxonomy of products in a supermarket

This example also illustrates one of the main points of the use of expressive representation languages and background knowledge. It becomes possible to emulate and represent many of the special cases and variants of existing approaches that have been developed. In this context, the example illustrates that a general relational association rule miner would be able to emulate the traditional association rule mining setting with taxonomies. Of course, the more general learner may pay a (computational) price for this generality; cf. Chapter 6.

Example 4.30. Reconsider the Bongard problem or Ex. 4.11. There one could employ clauses such as

[1] Some of the early inductive logic programming systems were called extensional because they were not able to employ intensional clauses.

$$
\begin{aligned}
&\mathsf{polygon(P)} \leftarrow \mathsf{triangle(P)}\\
&\mathsf{polygon(P)} \leftarrow \mathsf{square(P)}\\
&\ldots\\
&\mathsf{inside(O1, O2)} \leftarrow \mathsf{in(O1, O2)}\\
&\mathsf{inside(O1, O2)} \leftarrow \mathsf{in(O1, O3), inside(O3, O2).}
\end{aligned}
$$

Using these clauses, the example

$$\{\mathsf{circle(c1)},\ \mathsf{triangle(t1)},\ \mathsf{in(t1, c1)},\ \mathsf{triangle(t2)},\ \mathsf{in(t2, t1)}\}$$

could be completed with the following facts

$$\{\mathsf{inside(t1, c1)},\ \mathsf{inside(t2, t1)},\ \mathsf{inside(t2, c1)},\ \mathsf{polygon(t1)}, \mathsf{polygon(t2)}\}.$$

Example 4.31. In molecular applications, such as the one presented in Ex. 4.22, researchers have applied complex predicate definitions to define functional groups and ring structures. One (slightly simplified) example of such a rule defines a nitro group used in the mutagenicity application [Srinivasan et al., 1996]:

$$
\begin{aligned}
&\mathsf{nitro(Molecule, Atom0, Atom1, Atom2, Atom3)} \leftarrow\\
&\qquad \mathsf{atom(Molecule, Atom1, n, 38),}\\
&\qquad \mathsf{bond(Molecule, Atom0, Atom1, 1),}\\
&\qquad \mathsf{bond(Molecule, Atom1, Atom2, 2),}\\
&\qquad \mathsf{atom(Molecule, Atom2, o, 40),}\\
&\qquad \mathsf{bond(Molecule, Atom1, Atom3, 2),}\\
&\qquad \mathsf{atom(Molecule, Atom3, o, 40).}
\end{aligned}
$$

The last arguments of the relations bond and atom specify the bond and atom type, respectively.

4.10 Designing It Yourself

Despite the several different types of representation introduced above, it might still be hard to fit a given data set into one of these formats. In such cases, it may be worthwhile to develop your own representation using the principles of relational logic. This section presents a case study using relational sequences. It shows that variants of the standard relational representations grounded in the same methodology may well be beneficial for specific types of data and applications. At the same time, it shows that the specific representations employed within relational data mining are less important than the underlying methodology.

Assume that you are interested in modeling users of computer systems. Two types of data are of interest. On the one hand, there are sequences of Unix commands that you want to analyze, and on the other hand, there are sequences of web pages that specific users visit during a single session. The raw data for these sequences appear as:

emacs chapter2.tex, latex chapter2.tex, bibtex chapter2.tex,
xdvi chapter2.dvi, dvips chapter2.dvi, lpr chapter2.ps, ...

and

www.uni-freiburg.de,
www.informatik.uni-freiburg.de,
www.informatik.uni-freiburg.de/ml/,
www.informatik.uni-freiburg.de/kersting.htm,
www.machine-learning.org,...

At this point, there are various possibilities for representing these sequences. There are also several levels of abstraction that can be used. At the lowest level of abstraction, one might use the commands and web addresses as atomic descriptions directly. This is, however, problematic because there are very many of these atoms, and more importantly because the relations among the different components of various commands or web pages get lost. For the command sequences, the names of the files are important and are common across different commands in the sequence, and for the web pages, the same can be said about the domains. The next two alternatives are at a higher level of abstraction.

The first is to directly encode the sequences in a relational format. For instance, the command sequence could be encoded as facts:

emacs(1,chapter2,tex)←, latex(2,chapter2,tex)←, bibtex(3,chapter2,tex)←,
xdvi(4,chapter2,dvi)←, dvips(5,chapter2,dvi)←, lpr(6,chapter2,ps)←, ...

Here, the sequential aspect is captured by the first argument of the predicates, containing the number of the command in the sequence. Working with this relational representation requires the use of relations, such as the successor succ/2 predicate, in the background knowledge to work with these numbers. Within this representation, the pattern of an emacs command followed by a latex command for the same file would be represented by a clause such as pattern(F) ← emacs(N, F, tex), latex(M, F, tex), succ(M, N). Although interesting patterns can be represented, the sequential nature of the data is only represented implicitly and is rather complex to handle. Instead of the expected two conditions in the pattern, one for emacs and one for latex, there are three: one additional one for taking care of the sequential aspect.

The third option is to represent the sequences using lists in Prolog:

seq([emacs(chapter2, tex), latex(chapter2, tex), bibtex(chapter2, tex),
xdvi(chapter2, dvi), dvips(chapter2, dvi), lpr(chapter2, ps), ...])

Whereas this is again possible and alleviates the need to employ numbers, it also complicates the description of natural patterns. For instance, the above pattern would be represented using the following set of clauses:

pattern(F, [emacs(F, tex), latex(F, tex)|Rest]) ←
pattern(F, [Head|Tail]) ← pat(F, Tail).

This definition is not only recursive, it is also quite complex, which implies that it will be hard to generate using an inductive logic programming system.

So, why not employ a slightly different representation for sequences and hypotheses? Let us use the notation

$$\mathsf{emacs(chapter2, tex)\ latex(chapter2, tex)\ bibtex(chapter2, tex)}$$
$$\mathsf{xdvi(chapter2, dvi)\ dvips(chapter2, dvi)\ lpr(chapter2, ps)...}$$

to denote sequences, and

$$\leftarrow \mathsf{emacs(F, tex)\ latex(F, tex)}$$

to denote a pattern. Furthermore, a pattern $p_1 \dots p_n$ covers a sequence $s_1 \dots s_m$ if and only if there exists a substitution θ and a number $i \in \{1, \dots, m\}$ such that $p_1\theta = s_i \wedge \dots \wedge p_n\theta = s_{i+n-1}$. This setting will be referred to as that of logical sequences. Note that the above pattern covers the above sequence of commands with $i = 1$ and $\theta = \{\mathsf{F} \leftarrow \mathsf{chapter2}\}$. Employing the logical sequence representation is not only much simpler and natural than the ones previously discussed, it is also much more efficient. It can be shown that one can decide whether a sequential pattern covers a sequence of atoms in polynomial time.

Exercise 4.32. * Design the equivalent of clauses to encode background knowledge in the logical sequence setting. (Hint: use a representation based on grammar rules.)

4.11 A Hierarchy of Representations*

In the first few sections of this chapter, various representation formalisms have been introduced. The most important ones were: boolean representations (BL), attribute-value representations (AV), multi-instance representations (MI), relational representations (RR) and, finally, logic programs (LP). In this section, we investigate the representational power of these formalisms, and show that they form a hierarchy in the sense that any data set (and corresponding hypothesis) that can be represented using attribute-value representations, can also be represented in multi-instance format, and correspondingly for the other representations.

To investigate the relationships among these representations in more detail, we assume that learning tasks are characterized by their set of examples E, that the learning from interpretation settings is being used and that the hypotheses in $\mathcal{L}_h$ are sets of clauses of the form:

$$h \leftarrow l_1, \dots, l_n \tag{4.4}$$

Further requirements on $\mathcal{L}_h$ and $\mathcal{L}_e$ are imposed by the different representational languages:

- Boolean representations (BL) require that all predicates be of arity 0.[2]
- Attribute-value representations (AV) require that $n = 1$, that no functors be used, and that each example contain exactly one fact.
- Multi-instance representations (MI) require that $n = 1$ and that no functors be used. For multi-tuple and multi-join problems, n can be larger than 1.
- Relational representations (RL) require that no functors be used and that the number of constants (the domain) be finite.
- Logic programs (LP) do allow for functors and structured terms.

In addition, for BL, AVL, MI and RL, clauses will typically contain atoms of the form class(value) whereas for LP they will typically take arguments as well. We shall treat the former case by referring to the binary case, that is, by employing the atom positive in the conclusion part.

We say that a learning problem, characterized by the set of positive and negative examples $E \subseteq \mathcal{L}_{e_R}$, is *representable* using the representation R if there exists a hypothesis $h \in \mathcal{L}_{h_R}$ that covers all positive examples of E and none of its negative examples. For instance, the playtennis illustration of Ex. 4.5 is representable using AV and the carrier example of Ex. 4.2 is representable using MI. It is now easy to see that the playtennis example is also representable using MI (because $\mathcal{L}_{e_{AV}} \subseteq \mathcal{L}_{e_{MI}}$ and $\mathcal{L}_{h_{AV}} \subseteq \mathcal{L}_{h_{MI}}$). However, because we are also interested in investigating the other direction, it is worthwhile to introduce the notion of a reduction. Reductions are used in the theory of computer science for investigating the relationships amongst various representations and computational problems.

Given two representations X and Y, we say that X *is reducible to* Y, notation $X \sqsubseteq Y$, if and only if there exist two functions $f_e : \mathcal{L}_{e_X} \rightarrow \mathcal{L}_{e_Y}$ and $f_h : \mathcal{L}_{h_X} \rightarrow \mathcal{L}_{h_Y}$ such that h covers e if and only if $f_h(h)$ covers $f_e(e)$. Thus the function f_e maps examples from representation X to Y, and the function f_h does the same for hypotheses. The two functions f_e and f_h are together called a *reduction*. The definition actually implies the following property:

Property 4.33. If a learning problem E is representable using X, and X is reducible to Y, then $f_e(E)$ is representable using Y, where $f_e(E) = \{f_e(e)|e \in E\}$.

This property is important because it allows one to solve a learning problem in X using a learning algorithm for Y. This involves transforming the set of examples E into $f_e(E)$ and generating a solution $h_Y \in \mathcal{L}_{h_Y}$. The hypothesis h_Y will then cover all positive examples in $f_e(E)$ and none of the negative examples. It can also be used to predict whether unseen examples e (in $\mathcal{L}_{e_X}$) are covered by the unknown target concept by testing whether h_y covers $f_e(e)$. Notice that this notion of reducibility does not require f_h to be a bijection. If

[2] In boolean representations, it is also typical to allow for negative literals, that is, negated conditions, in clauses.

f_h were a bijection, it would be possible to map the found solution h_Y back to a hypothesis $f_h^{-1}(h_Y)$ represented in X.

As one trivial example of a reduction, consider the reducibility from MI to RL by using the identity functions for f_e and f_h. As similar reductions exist for the other representations in the hierarchy:

Property 4.34. $BL \sqsubseteq AV \sqsubseteq MI \sqsubseteq RR \sqsubseteq LP$.

Because reductions $\sqsubseteq$ are transitive, Proposition 4.34 implies that any boolean mining problem can be solved using an inductive logic programming system. This confirms the intuition that one can indeed use an algorithm with a richer representation than required to solve a given learning problem. However, in practice, this is not always a good idea. The reason for this is one of the basic lessons from computer science: expressiveness comes at a computational cost. Indeed, there is a trade-off between the two. It states that more expressive formalisms are typically computationally more expensive as well. This is to be expected because simpler representations can usually be better optimized. Therefore, it is often more effective and efficient to work within the original representations if possible. Nevertheless, the reader should keep in mind that systems working within the richer representations can often serve as benchmarks.

Defining a hierarchy of representations is useful. However, it is also useful to know whether the hierarchy is proper, that is, whether there exist representations X and Y in our hierarchy that X not only reduces to Y but Y also reduces to X. This issue has received quite some attention (and even controversy) in the machine learning community for AV and RL. The reason for this is that if RL were reducible to AV then there would be no need to develop learning and mining systems that directly work with relational representations. Then one could still formulate relational problems and solve them using traditional algorithms. Let us therefore try to shed some light on the answers to these questions using the notion of reducibility just introduced. In doing so, we do not resort to formal proofs and definitions, but rather try to illustrate the key issues, arguments and transformations by example. The reductions introduced also form the basis for many of the popular and practical approaches to *propositionalization*. Propositionalization is the process of generating a number of useful attributes or features starting from relational representations and then using traditional propositional algorithms for learning and mining. So, propositionalization techniques employ a kind of *incomplete* reduction. They are discussed in more detail in the next section.

4.11.1 From AV to BL

First, there is the question of whether $AV \sqsubseteq BL$; cf. Ex. 4.7. Provided that all the attributes are discrete, this is indeed the case as indicated in the next example, which also explains why both AV and BL are commonly referred to as propositional representations.

Example 4.35. Reconsider the playtennis example of Table 4.1. Each attribute Att with its corresponding domain $d(Att)$ can be mapped onto a set of boolean variables that indicate whether $Att = value$. For instance, the first example in Table 4.1 corresponds to the interpretation

$$\{'\mathsf{outlook} = \mathsf{sunny}',\ '\mathsf{temp} = \mathsf{hot}',\ '\mathsf{humid} = \mathsf{high}',\ '\mathsf{windy} = \mathsf{no}'\}.$$

where $'\mathsf{outlook} = \mathsf{sunny}'$ denotes the propositional variable yielding the value true when the attribute outlook has the value sunny, and false otherwise. All other propositional variables in this example do not occur in the interpretation, and hence are false. In a similar vein, one can transform rules from attribute-value format to boolean format.

It is easy to see that the two mappings just introduced form a reduction from AV to BL. When applying the reductions in a practical setting, one must realize that the constraints shown in Eqs. 4.1 and 4.2 are no longer automatically satisfied, and hence are lost in the reduction. These constraints specified that for each attribute and example there is exactly one value. When looking at, for instance, the attribute outlook with domain $\{\mathsf{sunny}, \mathsf{rainy}, \mathsf{overcast}\}$, the boolean representation no longer guarantees that when the proposition $'\mathsf{outlook} = \mathsf{sunny}'$ is true the propositions $'\mathsf{outlook} = \mathsf{overcast}'$ and $'\mathsf{outlook} = \mathsf{rainy}'$ must be false.

4.11.2 From MI to AV

Secondly, let us consider how to reduce a multi-instance problem to an attribute-value representation, that is, how $MI \sqsubseteq AV$. A reduction for this case must map every multi-instance example onto a single tuple in a single table.

The first naive approach to reducing MI to AV imposes an upper bound on the number of possible instances in an example.

Example 4.36. Consider the multi-instance example

$$\{\mathsf{object}(\mathsf{red}, \mathsf{triangle}), \mathsf{object}(\mathsf{blue}, \mathsf{circle})\}$$

The number of instances for this example is 2 and assume that this is also an upper bound. At this point, one might map this example onto

$$\{\mathsf{obj2}(\mathsf{red}, \mathsf{triangle}, \mathsf{blue}, \mathsf{circle})\}$$

The reduced example has twice the number of attributes as the original one.

There are several problems with this approach. First, the reduced example is not unique. Indeed, in the above example, it was implicitly assumed that the red triangle was the first instance and the blue circle the second one. Multi-instance examples are, however, unordered, and therefore an equivalent reduced example would be

$$\{\mathsf{obj2}(\mathsf{blue}, \mathsf{circle}, \mathsf{red}, \mathsf{triangle})\}$$

In general, the number of such equivalent representations will be exponential in the number of possible instances. Furthermore, this effect will carry over to the level of hypotheses, for example, the clauses $\mathsf{pos} \leftarrow \mathsf{obj2}(\mathsf{blue}, \mathsf{X}, \mathsf{Y}, \mathsf{Z})$ and $\mathsf{pos} \leftarrow \mathsf{obj2}(\mathsf{X}, \mathsf{Y}, \mathsf{blue}, \mathsf{Z})$. Secondly, if the number of instances in an example varies, then one needs to introduce null values for some of the attributes. A null value indicates that the value of the attribute is not defined. For instance, the example containing a single object, a red square, could be represented as $\{\mathsf{obj2}(\mathsf{red}, \mathsf{square}, \mathsf{nil}, \mathsf{nil})\}$. These problems show why the naive approach is problematic in practice.

The second idea always applies provided that the hypothesis space is known and finite. In this case, the idea is to explicitly enumerate all the rules in the hypothesis space and to test whether they cover the example or not. This implies that for each rule one introduces a boolean variable, which is true if the rule covers the example, and false otherwise.

Example 4.37. Reconsider the multi-instance example

$$\{\mathsf{object}(\mathsf{red}, \mathsf{triangle}), \mathsf{object}(\mathsf{blue}, \mathsf{circle})\}$$

Depending on the domain of the attributes of the relation object, propositional features corresponding to the following queries are possible:

$$\begin{array}{ll}
\leftarrow \mathsf{object}(\mathsf{X}, \mathsf{Y}) & \leftarrow \mathsf{object}(\mathsf{red}, \mathsf{triangle}) \\
\leftarrow \mathsf{object}(\mathsf{red}, \mathsf{X}) & \leftarrow \mathsf{object}(\mathsf{red}, \mathsf{circle}) \\
\leftarrow \mathsf{object}(\mathsf{blue}, \mathsf{X}) & \leftarrow \mathsf{object}(\mathsf{blue}, \mathsf{triangle}) \\
\leftarrow \mathsf{object}(\mathsf{Y}, \mathsf{circle}) & \leftarrow \mathsf{object}(\mathsf{blue}, \mathsf{circle}) \\
\leftarrow \mathsf{object}(\mathsf{Y}, \mathsf{triangle}) &
\end{array}$$

These can now be employed to describe the original example. Indeed, each of these attributes has the value true or false. Hence, one can describe the example using the following propositional interpretation:

$$\begin{array}{l}
\{'\mathsf{object}(\mathsf{X}, \mathsf{Y})', \ \ '\mathsf{object}(\mathsf{red}, \mathsf{X})', \ \ '\mathsf{object}(\mathsf{blue}, \mathsf{X})', \\
'\mathsf{object}(\mathsf{Y}, \mathsf{circle})', \ \ '\mathsf{object}(\mathsf{red}, \mathsf{triangle})', \ \ '\mathsf{object}(\mathsf{blue}, \mathsf{circle})'\}
\end{array}$$

where $'\mathsf{object}(\mathsf{X}, \mathsf{Y})'$ denotes the propositional variable corresponding to the query $\leftarrow \mathsf{object}(\mathsf{X}, \mathsf{Y})$.

Observe also that every multi-instance rule can now be represented as a rule of the form *class* ← *query* where *query* is one of the generated features. Rules with conjunctive conditions actually correspond to multi-tuple hypotheses.

So, the above example illustrates a true reduction. On the other hand, it is clear that the reduction is computationally expensive and the size of the resulting table explodes. Therefore the reduction is not efficient and is to be avoided in practice. Nevertheless, some propositionalization approaches exist that heuristically generate a useful (but limited) set of queries in this way; cf. the next section. A further significant problem with this reduction is that the structure on the hypothesis space – the *is more general than relation*, used by most mining and learning systems is lost; cf. Chapter 5.

Exercise 4.38. In Ex. 4.37, if one only employs the leftmost queries, does one still have a reduction?

4.11.3 From *RL* to *MI*

The next question to address is whether relational learning problems can be transformed into multi-instance learning problems. The main difference between these two types of representations, is that relational representations allow – as their name indicates – for multiple relations. So how can one then reduce these multiple tables to a single table? The theory of databases has introduced the notion of the *universal relation* for this purpose. The *universal relation* consists of the Cartesian product of the underlying relations. This concept can directly be applied for obtaining a reduction.

Example 4.39. Let us illustrate this concept on a reduced version of the summer school database, in which there is only one participant, scott, two courses, erm and srw, and two companies, scuf and pharmadm. Let us assume that all tuples involving any other course, company or participant have been deleted from Fig. 4.1. The resulting relations as well as the universal relation over these relations are shown in Fig. 4.10. The relational example participant(scott, researcher, scuf) is now turned into $1 \times 2 \times 2 \times 2 = 8$ tuples for a single relation. Also, any clause over the relations in the original database can be expanded into a clause over the universal relation. For instance, the clause

$$\begin{aligned}&\mathsf{attendsParty(P)} \leftarrow\\ &\qquad \mathsf{participant(P, researcher, C)},\\ &\qquad \mathsf{subscription(P, C')},\\ &\qquad \mathsf{course(C', L, advanced)}\end{aligned}$$

corresponds to

$$\mathsf{attendsParty(P)} \leftarrow \mathsf{ur(P, researcher, C, P, C', C', L, advanced)}.$$

Reducing clauses involving more than one occurrence of a predicate may require a multi-tuple hypothesis rather than a multi-instance one.

The previous example can easily be generalized from the summer school database context. It shows that in general it is possible to reduce a relational learning problem to multi-instance (or multi-tuple) learning one. Nevertheless, it should also be clear (from database theory as well as the above example) that this reduction is combinatorially explosive and therefore not to be used in practice. For instance, for applications in computational chemistry, such as that illustrated in Ex. 4.22, small molecules have 40 atoms and 80 bonds. Whereas the original relational representation would involve about 120 tuples, the Cartesian product would already contain 3,200. This is computationally prohibitive. Nevertheless, the above reduction is sometimes adapted for propositionalization, as will be explained in the next section.

Notice that because $RL \sqsubseteq MI$ and $MI \sqsubseteq AV$, RL can also be reduced to AV by applying the two reductions in sequence.

Name	Job	Company
scott	researcher	scuf

(a) participant

Name	Course
scott	erm
scott	srw

(b) subscription

Course	Length	Type
erm	3	introductory
srw	3	advanced

(c) course

Company	Type
scuf	university
pharmadm	university

(d) company

Name	Job	Company	Name	Course	Course	Length	Type	Company	Type
scott	res.	scuf	scott	erm	erm	3	intro.	scuf	univ.
scott	res.	scuf	scott	srw	erm	3	intro.	scuf	univ.
scott	res.	scuf	scott	erm	srw	3	adv.	scuf	univ.
scott	res.	scuf	scott	srw	srw	3	adv.	scuf	univ.
scott	res.	scuf	scott	erm	erm	3	intro.	pdm	univ.
scott	res.	scuf	scott	srw	erm	3	intro.	pdm	univ.
scott	res.	scuf	scott	erm	srw	3	adv.	pdm	univ.
scott	res.	scuf	scott	srw	srw	3	adv.	pdm	univ.

(e) the universal relation

Fig. 4.10. The short summer school database

4.11.4 From *LP* to *RL*

The remaining question concerning our hierarchy is whether $LP \sqsubseteq RL$. Upon a first investigation, this seems implausible, because LP is a programming language and RL is not. Nevertheless, there exists a useful transformation from LP to RL that is, however, not a proper reduction. It is the so-called flattening operation introduced by Rouveirol [1994].

There are basically two important differences between logic programs and relational expressions. First, logic programs may contain recursive definitions, such as the successor, ancestor, member or append relations. As there also exist extensions of relational databases, such as Datalog, that support recursion, we will not further deal with this aspect of the representation here. Dealing with recursive clauses in an inductive logic programming setting will be discussed in Chapter 7. Secondly, logic programs may contain structured terms that represent the underlying data structures such as lists and trees; cf.

Sect. 4.7. The flattening transformation takes as input a single clause involving structured terms and generates a flattened clause (without functors) and a number of facts. To flatten a clause involving structured terms, introduce for each functor f of arity n a new predicate p_f of arity $n+1$. The predicate p_f is then defined by the fact $p_f(X_1, ..., X_n, f(X_1, ..., X_n)) \leftarrow$. This type of predicate will be called a *functor predicate.* More formally:

Definition 4.40. *Let c be a clause; then $flat(c) =$*

if c contains a structured term $f(t_1, ..., t_n)$, then return $flat(c')$ where c' is c with all occurrences of $f(t_1, ..., t_n)$ replaced by V and with $p_f(t_1, ..., t_n, V)$ added to the condition part of c;
return also the fact $p_f(X_1, ..., X_n, f(X_1, ..., X_n)) \leftarrow$;
otherwise return c.

Example 4.41. Flattening the member program, that is, the two clauses

$$\begin{array}{l}\mathsf{member(X, cons(X, Y))} \leftarrow \\ \mathsf{member(X, cons(Y, Z))} \leftarrow \mathsf{member(X, Z).}\end{array}$$

(where we use $\mathsf{cons(X, Y)}$ to denote the list $\mathsf{[X|Y]}$) yields

$$\begin{array}{l}\mathsf{member(X, V)} \leftarrow \mathsf{pcons(X, Y, V).} \\ \mathsf{member(X, V)} \leftarrow \mathsf{pcons(Y, Z, V),\ member(X, Z).} \\ \mathsf{pcons(X, Y, cons(X, Y))} \leftarrow\end{array}$$

The inverse operation is called *unflattening* and is defined as follows:

Definition 4.42. *Let c be a clause, then $unflat(c) =$*

if c contains a literal referring to a functor predicate p_f, then return $unflat(c')$ where c' is the resolvent of the fact $p_f(X_1, ..., X_n, f(X_1, ..., X_n)) \leftarrow$ with the clause c;
otherwise, return c.

Unflattening a flattened clause yields the original clause again:

Property 4.43. If c is a clause then $unflat(flat(c)) = c$, provided that the functor predicates are known.

Flattening also preserves entailment:

Property 4.44. When P is a definite clause program, c a clause, and c and P do not contain any functor predicates, then $P \models c$ if and only if $flat(P) \models flat(c)$.

Despite these properties, the flattening clauses do not yield theories in relational form. The reason is that the flattening removes functors from clauses only to add them back through the definition of the functor predicates. Thus, flattening is only useful to manipulate single clauses; it does not really help at the program level. In order to be able to transform full programs into relational form, many practitioners of inductive logic programming have often employed a variant of the flattening operation that works on sets of ground facts or interpretations.

Definition 4.45. *Let $q(t_1, ..., t_n)$ be a ground fact. The flattened interpretation $flat_i(I)$ of an interpretation I is*

$$flat_i(I) = \bigcup_{q(t_1,...,t_n) \in I} (\{q(c_{t_1}, ..., c_{t_n})\} \cup$$
$$\{f_p(c_{u_1}, ..., c_{u_m}, c_{f(u_1,...,u_m)}) | f(u_1, ..., u_m) \textit{is a sub-term of some } t_i\})$$

where each c_t denotes a unique constant.

Example 4.46. Flattening the interpretation I

$$\{\mathsf{append}([1],[2],[1,2]);\ \mathsf{partition}(2,[1],[1],[]);\ \mathsf{sort}([],[]);$$
$$\mathsf{sort}([1],[1]);\ \mathsf{sort}([2,1]),[1,2])\}$$

yields $flat_i(I)$:

$$\{\mathsf{append}(\mathsf{c}_{[1]},\mathsf{c}_{[2]},\mathsf{c}_{[1,2]}),\ \mathsf{partition}(\mathsf{c}_2,\mathsf{c}_{[1]},\mathsf{c}_{[1]},\mathsf{c}_{[]}),\ \mathsf{sort}(\mathsf{c}_{[]},\mathsf{c}_{[]}),$$
$$\mathsf{sort}(\mathsf{c}_{[1]},\mathsf{c}_{[1]}),\ \mathsf{sort}(\mathsf{c}_{[2,1]},\mathsf{c}_{[1,2]})\ \mathsf{consp}(\mathsf{c}_1,\mathsf{c}_{[2]},\mathsf{c}_{[1,2]}),$$
$$\mathsf{consp}(\mathsf{c}_1,\mathsf{c}_{[]},\mathsf{c}_{[1]}),\ \mathsf{consp}(\mathsf{c}_2,\mathsf{c}_{[1]},\mathsf{c}_{[2,1]}),\ \mathsf{consp}(\mathsf{c}_2,\mathsf{c}_{[]},\mathsf{c}_{[2]})\}.$$

This operation has been used in many attempts to synthesize programs from examples using systems that employ relational representations only. These systems started by flattening the examples, such as $\mathsf{sort}([2,3,1],[1,2,3])$, and then adding the resulting interpretation to the relational database. When applying this transformation, the reader should keep in mind that it is not a proper reduction and also that the transformation is combinatorially explosive. The reason for this is that the converse of the following property does not hold, as illustrated in the example below.

Property 4.47. If I is an interpretation that is a model for the clause c a clause, then $flat_i(I)$ is a model of flat(c).

Example 4.48. The reader may first want to verify that the property holds for the interpretation specified in Ex. 4.46 and the clause

$$\begin{aligned}&\mathsf{sort}([\mathsf{A}|\mathsf{B}],\mathsf{S}) \leftarrow\\ &\qquad \mathsf{partition}(\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D}),\mathsf{sort}(\mathsf{C},\mathsf{E}),\\ &\qquad \mathsf{sort}(\mathsf{D},\mathsf{F}),\mathsf{append}(\mathsf{E},[\mathsf{B},\mathsf{F}],\mathsf{S}).\end{aligned}$$

Example 4.49. Consider the clause $\mathsf{nat}(\mathsf{s}(\mathsf{X})) \leftarrow \mathsf{nat}(\mathsf{X})$, stating that the successor of X is a natural number if X is a natural number. Although the interpretation $\{\mathsf{nat}(\mathsf{s}(0)),\ \mathsf{nat}(0)\}$ is not a model for the clause, it is easily verified that the flattened interpretation $\{\mathsf{nat}(\mathsf{c}_{\mathsf{s}(0)}),\ \mathsf{nat}(\mathsf{c}_0),\ \mathsf{sp}(\mathsf{c}_0,\mathsf{c}_{\mathsf{s}(0)})\}$ *is* a model of the flattened clause $\mathsf{nat}(\mathsf{Y}) \leftarrow \mathsf{sp}(\mathsf{X},\mathsf{Y}),\ \mathsf{nat}(\mathsf{X})$.

4.12 Propositionalization

The previous section introduced a hierarchy of representations and studied the relationships among the various representational formalisms. It was shown that some representational formalisms can be reduced to one another. For instance, RL problems can be reduced to MI form, and MI problems can be reduced to AV format. On the other hand, it was also argued that these reductions cannot be applied on real data sets because of the enormous computational costs and combinatorics involved. At this point, the question arises as to whether it might be possible to control the combinatorial explosion in some way. When it is impossible to compute the reduced data set, one might still attempt to approximate it. An approximation to a complete reduction in AV form is called a *propositionalization*. It might be an approximation in the sense that the functions f_e and f_h are not a reduction in the formal sense, that is, for some examples e and hypotheses h, $\mathbf{c}(h, e) \neq \mathbf{c}(f_h(h), f_e(e))$; in other words, the coverage of h w.r.t. e might be different than that for $f_h(h)$ and $f_e(e)$, which would imply that some information is lost in the transformation. If such inconsistencies occur only very rarely, the transformation might still be useful, because the propositionalized problem might still capture many essential features of the problem, and a traditional propositional learner may be applied successfully to the propositionalized problem.

One advantage of such *propositionalization* is that the whole set of traditional learning algorithms, including neural networks, statistics, support vector machines and so on, can be applied to the propositionalized problem. The disadvantage is that the propositionalized problem might be incomplete and that some information might get lost in the propositionalization process. In the past few years, many approaches to propositionalization have been developed. The majority of these directly transform a relational description into an attribute-value one, though some also consider the intermediate level of multi-instance descriptions; cf. [Zucker and Ganascia, 1998]. In line with the philosophy of the book, we sketch in the remainder of this section two different types of propositionalization and discuss some general issues, rather than provide a detailed survey of the many specific approaches to propositionalization that have been developed over the past few years.

4.12.1 A Table-Based Approach

The first approach to propositionalization builds an approximation of the universal relation sketched in Sect. 4.11.3. It typically employs the learning from entailment setting, and starts from a set of positive and negative examples (in the form of ground facts) as well as from a clause $h \leftarrow b_1, \ldots, b_m$ defining the unknown target predicate. It then computes all substitutions θ that ground the clause and for which $h\theta$ is an example and all the $b_i\theta$ are true. The resulting substitutions are then listed in a table together with the class of the example $h\theta$. Let us demonstrate this technique on the summer school database.

Example 4.50. Consider the $\mathsf{attendsParty}$ predicate of Fig. 4.2. The resulting table for the clause

$$\mathsf{attendsParty(P)} \leftarrow \mathsf{participant(P, J, C), company(C, T)}$$

is shown in Table 4.3. Each example (person in this case) is represented by a single row in that table. Furthermore, the class of the example corresponds to the Party attribute. This table can directly serve as input for an attribute-value learner. The only further preprocessing still needed is that the argument of the $\mathsf{attendsParty}$ predicate, that is, the attribute Name (the column for the variable P), be dropped. An attribute-value rule learner might then come up with a rule such as $'\mathsf{Party = yes}' \leftarrow '\mathsf{Job = researcher}'$ which could be transformed backwards into the clause

$$\mathsf{attendsParty(P)} \leftarrow \mathsf{participant(P, researcher, C), company(C, T)}.$$

P	J	C	T	Party
adams	researcher	scuf	university	no
blake	president	jvt	commercial	yes
king	manager	pharmadm	university	no
miller	manager	jvt	commercial	yes
scott	researcher	scuf	university	yes
turner	researcher	pharmadm	university	no

Table 4.3. A propositionalized version of the summer school database

One refinement of the table-based approach is concerned with removing identifiers. In the previous examples, the attribute Name has been removed because it corresponds to an identifier that is specific to the example and that is of no further interest. However, there may be further relations in the database that contain such identifiers, for instance, the novel relation $\mathsf{isMarriedTo/2}$, which could motivate the use of the clause

$$\mathsf{attendsParty(P)} \leftarrow \mathsf{participant(P, J, C),\ company(C, T),\ isMarriedTo(P, W)}.$$

As one is typically not interested in the names of the persons, but only in whether the person in question is married or not, instead of adding an attribute Wife one might add the attribute $\mathsf{IsMarried}$, which is true if the query $\leftarrow \mathsf{isMarried(P, W)}$ succeeds. Notice that the dimensions (and therefore its complexity) of the table are controlled by the clause one starts from. The clause thus constitutes a *syntactic bias* as it restricts the syntax of the hypotheses that can be generated.

The table-based approach as sketched so far only works when the obtained table satisfies the single-tuple assumption: every example should be mapped onto a single tuple. In general, this will not be the case. For example, if the clause

$$\mathsf{attendsParty(P)} \leftarrow \mathsf{participant(P,J,C)},\ \mathsf{company(C,T)},\ \mathsf{subscribes(P,C')}$$

is used, there are multiple tuples for some of the participants in the table. So, a multi-instance problem would be obtained rather than an attribute-value one, and therefore, a multi-instance learning algorithm could be applied. As far as the author knows, the use of multi-instance learners in propositionalization has not yet been investigated systematically. If on the other hand, an attribute-value representation is targeted, then one should only follow relations of type **(n:1)** and **(1,1)** when starting from the target relation. This concept has been formalized in the inductive logic programming literature under the term *determinacy*; cf. Muggleton and Feng [1992], which guarantees that the resulting problem is in attribute-value format (cf. also Sect. 10.2).

Exercise 4.51. Assume that the target relation in the summer school database is subscription. Specify a feasible clause that results in an attribute-value representation, and specify one that yields a proper multi-instance representation.

4.12.2 A Query-Based Approach

The second approach has already been demonstrated in Ex. 4.37 when discussing the reduction from multi-instance to attribute-value learning. The idea there can be generalized. Essentially, one starts from a language of queries $\mathcal{L}_q$ which specifies the set of possible queries. A complete transformation then employs all queries expressible in this language. Whereas the language in Ex. 4.37 allowed for a single occurrence of the predicate object, in a general relational setting, one will typically employ more complex relational queries involving multiple literals. For instance, in the Bongard problem illustrated in Ex. 4.11, one might employ, for instance, the following queries:

$$\begin{array}{l}\leftarrow \mathsf{triangle(T)}\\ \leftarrow \mathsf{circle(C)}\\ \leftarrow \mathsf{triangle(T), in(T,C)}\\ \leftarrow \mathsf{triangle(T), in(C,T)}\end{array}$$

The naive approach to query-based propositionalization generates all queries that are expressible within the language $\mathcal{L}_q$. To keep the number of such queries within reasonable limits, the user must either carefully engineer the language $\mathcal{L}_q$ or else filter the features so that only the most interesting ones are retained. Even though there is a wide range of query-based propositionalization techniques available, the underlying principles often remain the same.

To illustrate engineering the language $\mathcal{L}_q$, consider the choices made in the first propositionalization system, LINUS, by Lavrač et al. [1991]. If the goal is to learn a target predicate p/n, Linus learns clauses of the form $p(V_1, \ldots, V_n) \leftarrow b_1, \ldots, b_m$ where the variables V_i are the only possible arguments of the atoms b_i. The queries in the language $\mathcal{L}_q$ then correspond to all the atoms b_i that can be constructed in this way. For instance, when learning the predicate $\mathsf{daughter}(\mathsf{X}, \mathsf{Y})$ in terms of parent, male and female, the language $\mathcal{L}_q$ contains queries corresponding to the following atoms (where we have abbreviated the predicate names):

$$\mathsf{p}(\mathsf{X},\mathsf{X}), \mathsf{p}(\mathsf{Y},\mathsf{Y}), \mathsf{p}(\mathsf{X},\mathsf{Y}), \mathsf{p}(\mathsf{Y},\mathsf{X}), \mathsf{m}(\mathsf{X}), \mathsf{m}(\mathsf{Y}), \mathsf{f}(\mathsf{X}), \mathsf{f}(\mathsf{Y}).$$

Two approaches exist to filtering the features. One approach imposes hard constraints on the features of interest. A popular constraint is to require that the query succeeds for at least x instances, where x is a user-set threshold. This corresponds to imposing a minimum frequency threshold in frequent pattern mining; cf. Sects. 3.4 and 6.5. The other approach heuristically searches the space of possible queries and evaluates each query according to some measure of interestingness. This search process resembles that of rule learning (cf. Sect. 6.3) and indeed heuristic query-based approaches are often variants of rule learners, or they post-process rules generated by a multi-relational rule learner; cf. [Srinivasan and King, 1999a].

The existing approaches to propositionalization fall into two categories. The *static* ones first propositionalize and then run the attribute-value learner. The *dynamic* ones intervene the propositionalization and mining processes. They incrementally generate a number of features and use these for learning. If the quality of the learned hypothesis is not satisfactory, new features are generated and this process is repeated until there is no more improvement. Most of the present propositionalization approaches are static, but see [Landwehr et al., 2007, Popescul and Ungar, 2007] for two dynamic propositionalization approaches.

4.13 Aggregation

When querying a database, the specific tuples that belong to a certain table may be less interesting than the aggregated information over the whole table. Aggregation is related to propositionalization because it also reduces the available information, and presents it in a more compact form, most often resulting in a loss of information. Aggregated attributes are typically functions of either all tuples in a table or of a specific attribute in the table. Example aggregate functions include:

- COUNT, which counts the number of tuples in the table, or the number of values for an attribute,
- SUM, which computes the sum of the values for a specific attribute,
- AVG, which computes the average of the values for a specific attribute,
- MIN and MAX, which compute the minimum and maximum values for a specific attribute.

Aggregate functions can easily be integrated into our logical framework. The syntax and the formalization used is based on the work of Vens et al. [2006] though slightly modified for our purposes. We gradually introduce it by example.

Example 4.52. Consider the summer school database listed in Fig. 4.1. The expression

$$\mathsf{COUNT}\{\mathsf{Course}|\mathsf{subscription}(\mathsf{adams},\mathsf{Course})\}$$

first constructs the table listing all answer substitutions for the variables Course in the query $\mathsf{subscription}(\mathsf{adams},\mathsf{Course})$, resulting in Fig. 4.11a. It then applies the aggregate Count to the tuples in this relation. As adams subscribes to three courses, the value of the expression is 3.

In Prolog, it is common to write queries such as $\leftarrow \mathsf{M}\ \mathsf{is}\ 5+3$, where the expression is denotes that the first argument (M) unifies with the result of evaluating the second expression, which is the result $5+3$, that is, the value 8. When working with aggregates, we also allow the second argument to be expressions involving aggregates. The results can then be used as any other expression in Prolog. In this way, it is possible to simulate a kind of group-by statement, as in the relational database language SQL.

Example 4.53. Consider the clause

$$\mathsf{result}(\mathsf{Part},\mathsf{C}) \leftarrow \mathsf{C}\ \mathsf{is}\ \mathsf{COUNT}\{\mathsf{Course}|\mathsf{subscription}(\mathsf{Part},\mathsf{Course})\}$$

which corresponds to the SQL statement

$$\mathsf{SELECT\ Part,\ COUNT(Course)\ FROM\ subscription\ GROUP\ BY\ Part}$$

Because there are multiple participants, the aggregate COUNT first computes the table of answer substitutions for the query $\leftarrow \mathsf{subscription}(\mathsf{Part},\mathsf{Course})$ in Fig. 4.12a, and then applies the function COUNT to each of the different answer substitutions grouped by Part. In this way, the number of courses per participant are counted. The resulting predicate is shown in extensional form in Fig. 4.12b. We shall employ the convention that the answer substitutions are grouped by those variables that appear also outside the query in the set. In our example, Part also appears in the conclusion part of the clause defining the result predicate.[3]

The aggregated predicate definitions can now be used like any other predicate. For example, the clause

$$\mathsf{attendsParty}(\mathsf{Part}) \leftarrow \mathsf{COUNT}\{\mathsf{Course}|\mathsf{subscription}(\mathsf{Part},\mathsf{Course})\} \leqslant 2$$

[3] This convention is similar to the semantics of the well-known $\mathsf{bagof/3}$ and $\mathsf{setof/3}$ predicates in Prolog.

Course
erm
so2
srw

(a)

COUNT
3

(b)

Fig. 4.11. Computing $\mathsf{COUNT}\{\mathsf{Course}|\mathsf{subscription}(\mathsf{adams}, \mathsf{Course})\}$

Part	Course
adams	erm
adams	so2
adams	srw
blake	cso
blake	erm
king	cso
king	erm
king	so2
king	srw
miller	so2
scott	erm
scott	srw
turner	so2
turner	srw

(a) Answers.

Part	C
adams	3
blake	2
king	4
miller	1
scott	2
turner	2

(b) Result.

Fig. 4.12. Computing $\mathsf{result}(\mathsf{Part}, \mathsf{C})$

states that participants will attend the party if they take at most two courses.

Aggregates are popular in relational data mining because they can be used to reduce a multi-instance learning problem to an attribute-value learning problem, and in this way form an important tool for propositionalization.

Example 4.54. Consider the table-based propositionalization approach with regard to the query

$$\mathsf{attendsParty}(\mathsf{P}) \leftarrow \mathsf{participant}(\mathsf{P}, \mathsf{J}, \mathsf{C}), \mathsf{subscribes}(\mathsf{P}, \mathsf{C}')$$

This results in the table listed in Fig. 4.13a, which corresponds to a multi-instance learning problem. When using the clause

$$\mathsf{attendsParty}(\mathsf{P}) \leftarrow \mathsf{participant}(\mathsf{P}, \mathsf{J}, \mathsf{C}), \mathsf{V} \text{ is } \mathsf{COUNT}\{\mathsf{C}'|\mathsf{subscription}(\mathsf{P}, \mathsf{C}')\}$$

one obtains the attribute-value learning problem in Fig. 4.13b.

Exercise 4.55. Can you provide an example that illustrates the use of aggregation and that does not involve a reduction to an attribute-value learning problem?

P	J	C	C’
adams	researcher	scuf	erm
adams	researcher	scuf	so2
adams	researcher	scuf	srw
blake	president	jvt	cso
blake	president	jvt	erm
king	manager	pharmadm	cso
king	manager	pharmadm	erm
king	manager	pharmadm	so2
king	manager	pharmadm	srw
miller	manager	jvt	so2
scott	researcher	scuf	erm
scott	researcher	scuf	srw
turner	researcher	pharmadm	so2
turner	researcher	pharmadm	srw

(a) Multiple instances

P	J	C	V
adams	researcher	scuf	3
blake	president	jvt	3
king	manager	pharmadm	4
miller	manager	jvt	1
scott	researcher	scuf	2
turner	researcher	pharmadm	2

(b) Aggregated instances

Fig. 4.13. Multiple and aggregated instances

As the example illustrates, aggregation is a very useful operation in order to reduce sets of tuples to single tuples. In this way, aggregation attempts to summarize the information in the table. It is, however, important to realize that summarization always results in a loss of information. Depending on the application this can be harmless or problematic.

The other point to keep in mind is that the search space rapidly explodes when aggregation is allowed because one has to search for the right aggregation functions to apply to the right queries, and for a single query there are various ways in which aggregation can be applied. Furthermore, as we will see in Chapter 5, a further complication is that the generality relation becomes quite involved.

4.14 Conclusions

This chapter started by introducing two alternative logical settings for learning: learning from entailment and learning from interpretations.

We then presented a hierarchy of representations that are used in data mining and machine learning. The key representations are: boolean, attribute-value, multi-instance, relational and logic program representations. Other representations that are quite popular in data mining and computer science are sequences, trees and graphs. It was also shown that these traditional data structures can be elegantly represented using relational representations or logic programs.

We then investigated the relationship among these different representations; more specifically we have shown that functors can be eliminated from logic programs, that (bounded) relational representations can be reduced to multi-instance and attribute-value learning problems. One important insight concerns the identification of multi-instance learning as one of the central issues in relational learning. It is central because it is relatively easy to reduce relational learning to multi-instance learning, but much harder to reduce multi-instance learning to attribute-value learning. Even though the reductions presented show that the richer representations are, under certain assumptions, not needed, the sketched transformations are computationally too expensive to be applied in practice, except perhaps in specific circumstances. Therefore, it is advantageous to stay as close as possible to the genuine representation required by the problem at hand. When using simpler representations, one either confronts computational difficulties or loses some information; when using rich representations, computationally more expensive learning engines must be employed. An intermediate solution is based on heuristic propositionalization and aggregation approaches.

4.15 Historical and Bibliographical Remarks

The learning from entailment setting is due to Plotkin [1970] and is the most popular and prominent setting within inductive logic programming [Muggleton and De Raedt, 1994, Muggleton et al., 1992]. The learning from interpretations is due to De Raedt and Džeroski [1994], which in turn was strongly influenced by Helft [1989] and the boolean representations used in computational learning theory [Valiant, 1984, Kearns and Vazirani, 1994]. The relation among these and other logical settings for learning were investigated by De Raedt [1997]. The description of the hierarchy of representations and reductions is based on De Raedt [1998]. The flattening operation for clauses was introduced by Rouveirol [1994], and that for interpretations by De Raedt and Džeroski [1994], which in turn formalized common practice in various publications such as [Quinlan, 1993b].

Multi-instance learning was introduced by Dietterich et al. [1997] and was motivated by the musk problem in drug discovery. More information on the relation between logic and grammars and a deeper introduction to definite clause grammars can be found in any textbook on the Prolog programming language; cf. [Flach, 1994, Sterling and Shapiro, 1986, Bratko, 1990]. The case study with logical sequences follows Lee and De Raedt [2004] and is motivated by an application in user modeling due to Jacobs and Blockeel [2001].

Propositionalization has been the topic of many investigations in the past few years, and many different techniques have been developed [Kramer et al., 2001]. Nevertheless, the underlying principles are the table-based or query-based methods, possibly enhanced with aggregation, that we presented. The table-based approach described the work on the LINUS and DINUS systems by Lavrač and Džeroski [1994], and the query-based approach follows Srinivasan and King [1999a]. Aggregation has been used for quite a while in machine learning [Michalski, 1983], but has only recently become quite popular within relational learning [Geibel and Wysotzki, 1997, Knobbe et al., 2001, Perlich and Provost, 2003, Vens et al., 2006] and probabilistic relational models [Getoor et al., 2001a].

5

Generality and Logical Entailment

In this chapter, we study the generality *relation from a logical perspective. A central property in the theory of logical learning is that the generality relation coincides with logical entailment. This property results in an operational framework for studying and defining operators for generalization and specialization. Each of these operators originates from a deductive framework. We study several of the most important frameworks for generality, including θ-subsumption (and some of its variants) and inverse resolution, and then discuss some advanced issues such as the influence of background knowledge and aggregation.*

5.1 Generality and Logical Entailment Coincide

Recall from Sect. 3.6 Def. 3.4 that a hypothesis g is more general than a hypothesis s, notation $g \preceq s$, if and only if, $\mathbf{c}(s) \subseteq \mathbf{c}(g)$, that is, all examples covered by s are also covered by g. Analyzing this definition using the learning from entailment setting leads to the following property (where we add the subscript e to the symbol $\preceq$ to denote that we are learning from entailment):

Property 5.1. When learning from entailment, $g \preceq_e s$ if and only if $g \models s$.

Proof. * We prove the claim when no background knowledge is employed. It can easily be adapted to account for background knowledge. We also assume that all clauses can be used both as examples and as parts of hypotheses.

$\Longrightarrow$ if $g \preceq_e s$ then it follows that for all $e \in \mathcal{L}_e : e \in \mathbf{c}(s) \rightarrow e \in \mathbf{c}(g)$. This is equivalent to

$$\forall e : (s \models e) \rightarrow (g \models e) \tag{5.1}$$

Since s is a conjunction of clauses $s_1 \wedge \ldots \wedge s_n$ and $s \models s_i$, Eq. 5.1 and $s_i \in \mathcal{L}_e$ (because all clauses can be used as examples) imply that $\forall i : g \models s_i$, which in turn implies that $g \models s$.

$\Longleftarrow$ if $g \models s$, then, for all examples e, whenever $s \models e$ (that is, s covers e), $g \models e$ because of the transitivity of logical entailment. Thus $g \preceq_e s$. □

The property can be generalized to take into account a background theory B. We then say that g *is more general than* s *relative to the background theory* B, notation $g \preceq_{e,B} s$, if and only if $B \wedge g \models s$.

Property 5.2. When learning from interpretations, $g \preceq_i s$ if and only if $s \models g$.

Proof. This is a direct consequence of the definition of logical entailment, which states that $s \models g$ if and only if all models of s are a model of g. □

When learning from interpretations relative to a background theory B, we write that g is more general than s relative to B, notation $g \preceq_{i,B} s$, if and only if $B \wedge s \models g$.

Example 5.3. Consider the hypotheses h_1

$$\begin{array}{l}\mathsf{grandparent(GP, GC)} \leftarrow \mathsf{father(GP, C), parent(C, GC)} \\ \mathsf{father(F, C)} \leftarrow \mathsf{male(F), parent(F, C)}\end{array}$$

and h_2

$$\mathsf{grandparent(GP, GC)} \leftarrow \mathsf{male(GP), parent(GP, C), parent(C, GC)}$$

for which $h_1 \models h_2$. One can easily check that $h_1 \wedge \neg h_2$ is inconsistent (or that h_2 is the resolvent of the two clauses in h_2), hence $h_1 \models h_2$. Note that the reverse does not hold because, for instance,

$$\{\mathsf{father(jef, paul), parent(paul, an)}\}$$

is a model of h_2, but not of h_1. By Property 5.1 this means that h_1 is (strictly) more general than h_2 when learning from entailment ($h_1 \preceq_e h_2$). Thus there exist examples, such as

$$\mathsf{grandparent(leo, luc)} \leftarrow \mathsf{father(leo, rose), parent(rose, luc)}$$

that are covered by h_1 but not by h_2. On the other hand, when learning from interpretations, the generality relation reverses. According to Property 5.2 h_2 is strictly more general than h_1 ($h_2 \preceq_i h_1$). The above interpretation involving jef and paul is an example that is covered by h_2 but not by h_1.

Because the learning from entailment setting is more popular than that of learning from interpretations within traditional inductive logic programming, it is also the default setting, which explains why, traditionally, a hypothesis g is said to be more general than s if and only if $g \models s$. From now on, when the context is clear, we will also follow this convention.

The above two properties lie at the heart of the theory of inductive logic programming and generalization because they directly relate the central notions of logic with those of machine learning [Muggleton and De Raedt, 1994].

They are also extremely useful because they allow us to directly transfer results from logic to machine learning.

This can be illustrated using traditional deductive inference rules, which start from a set of formulae and derive a formula that is entailed by the original set. For instance, consider the *resolution* inference rule for propositional definite clauses:

$$\frac{h \leftarrow g, a_1, \ldots, a_n \text{ and } g \leftarrow b_1, \ldots, b_m}{h \leftarrow b_1, \ldots, b_m, a_1, \ldots, a_n} \tag{5.2}$$

As discussed in Chapter 2, this inference rule starts from the two rules above the line and derives the so-called *resolvent* below the line. This rule can be used to infer, for instance,

$$\mathsf{flies} \leftarrow \mathsf{blackbird}, \mathsf{normal}$$

from

$$\mathsf{flies} \leftarrow \mathsf{bird}, \mathsf{normal}$$
$$\mathsf{blackbird} \leftarrow \mathsf{normal}$$

An alternative deductive inference rule adds a condition to a rule:

$$\frac{h \leftarrow a_1, \ldots, a_n}{h \leftarrow a, a_1, \ldots, a_n} \tag{5.3}$$

This rule can be used to infer that

$$\mathsf{flies} \leftarrow \mathsf{blackbird}$$

is more general than

$$\mathsf{flies} \leftarrow \mathsf{blackbird}, \mathsf{normal}$$

In general, a deductive inference rule can be written as

$$\frac{g}{s} \tag{5.4}$$

If s can be inferred from g and the operator is *sound*, then $g \models s$. Thus applying a deductive inference rule realizes specialization, and hence deductive inference rules can be used as specialization operators. A *specialization operator* maps a hypothesis onto a set of its specializations; cf. Chapter 3. Because specialization is the inverse of generalization, *generalization operators* — which map a hypothesis onto a set of its generalizations — can be obtained by inverting deductive inference rules. The inverse of a deductive inference rule written in the format of Eq. 5.4 works from bottom to top, that is from s to g. Such an inverted deductive inference rule is called an *inductive* inference rule. This leads to the view of induction as the inverse of deduction. This view is operational as it implies that each deductive inference rule can be inverted into an inductive one, and that each inference rule provides an alternative framework for generalization.

An example generalization operator is obtained by inverting the *adding condition* rule in Eq. 5.3. It corresponds to the well-known *dropping condition* rule. As we will see soon, it is also possible to invert the resolution principle of Eq. 5.2.

Before deploying inference rules, it is necessary to determine their properties. Two desirable properties are *soundness* and *completeness.* These properties are based on the repeated application of inference rules in a proof procedure. Therefore, as in Chapter 2, we write $g \vdash_r s$ when there exists a sequence of hypotheses $h_1, \cdots, h_n$ such that

$$\frac{g}{h_1}, \frac{h_1}{h_2}, \cdots, \frac{h_n}{s} \text{ using } r \tag{5.5}$$

A proof procedure with a set of inference rules r is then *sound* whenever $g \vdash_r s$ implies $g \models s$, and *complete* whenever $g \models s$ implies $g \vdash_r s$. In practice, soundness is always enforced while completeness is an ideal that is not always achievable in deduction. Fortunately, it is not required in a machine learning setting. When working with incomplete proof procedure, one should realize that the generality relation $\vdash_r$ is weaker than the logical one $\models$.

The formula $g \models s$ can now be studied under various assumptions. These assumptions are concerned with the class of hypotheses under consideration and the operator $\vdash_r$ chosen to implement the semantic notion $\models$. The hypotheses can be single clauses, sets of clauses (that is, clausal theories), or full first-order and even higher-order theories. Deductive operators that have been studied include θ-subsumption (and its variants such as OI-subsumption) among single clauses, implication among single clauses and resolution among clausal theories. Each of these deductive notions results in a different framework for specialization and generalization. The most important such frameworks are presented below, including θ-subsumption (and some of its variants), inverse implication, and inverse resolution. Due to its relative simplicity, θ-subsumption is by far the most popular framework. As we shall see in the following chapters, the large majority of contemporary logical and relational learning systems employ operators under θ-subsumption in one form or another. We will therefore provide an in-depth presentation of this framework and provide shorter discussions of alternative frameworks. In addition, θ-subsumption will be gradually introduced. We first consider propositional (or boolean) clauses only, then investigate the structure on logical atoms, and finally, combine these two notions in order to obtain the full-fledged θ-subsumption framework.

5.2 Propositional Subsumption

When performing operations on clauses, it is often convenient to represent the clauses by the sets of literals they contain. For instance, the clause $\mathsf{flies} \leftarrow \mathsf{bird}, \mathsf{normal}$ can be represented as $\{\mathsf{flies}, \neg\mathsf{bird}, \neg\mathsf{normal}\}$. This implies that we are now using the same notation for both clauses, which are disjunctions, and for item-sets, which are conjunctions. Therefore, this should only be done when the context is clear. The reason for overloading the set notation will become clear soon.

Using this notion for two propositional clauses c_1 and c_2,

$$c_1 \text{ subsumes } c_2 \text{ if and only if } c_1 \subseteq c_2 \tag{5.6}$$

Example 5.4. The clause $\mathsf{flies} \leftarrow \mathsf{bird}, \mathsf{normal}$ subsumes the clause $\mathsf{flies} \leftarrow \mathsf{bird}, \mathsf{normal}, \mathsf{pigeon}$.

Observe that propositional subsumption is *sound*, which means that whenever c_1 subsumes c_2, it is the case that $c_1 \models c_2$, and *complete*, which means that whenever $c_1 \models c_2$, c_1 also subsumes c_2.

The resulting search space is shown in Fig. 5.1. At this point, the reader may observe that this search space coincides with that used for monomials in Fig. 3.5 of Chapter 3. Whereas this may be surprising at first sight because monomials are conjunctions and clauses are literals, there is a simple explanation. When working with clauses, a clause c is said to cover an example when $c \models e$, that is, when $c \subseteq e$. On the other hand, when working with item-sets and learning from interpretations, an item-set m covers an interpretation e when $m \subseteq e$. Therefore, when using sets to represent clauses and item-sets, the generality relation coincides. Therefore, for both item-sets and clauses, we have that hypothesis h_1 is more general than hypothesis h_2 if and only if $h_1 \subseteq h_2$. This also explains why we employ set notation for both clauses and monomials. The reader should keep in mind though that, at the logical level, item-sets and clauses are dual to one another, because conjunction is complementary to disjunction. Combined with the duality of the generality relation between learning from entailment (used for clauses) and learning from interpretations (used for item-sets), the generality relation coincides in both cases with the subset relation.

Because the generality relation for clauses and item-sets coincides when the set notation is used, the operators defined for item-sets are the same as those for clauses. This implies for instance that the *lgg* of $\mathsf{flies} \leftarrow \mathsf{blackbird}, \mathsf{normal}$ and $\mathsf{flies} \leftarrow \mathsf{pigeon}, \mathsf{normal}$ is $\mathsf{flies} \leftarrow \mathsf{normal}$. Observe also that the space of propositional clauses forms a lattice and possesses optimal as well as ideal operators, which makes them easy to use. Recall that ideal operators generate all children (or parents) of a hypothesis in a Hasse diagram, whereas optimal operators ensure that there is exactly one path from the most general hypothesis to any specific hypothesis through the refinement graph; cf. Chapter 3.

5.3 Subsumption in Logical Atoms

When working with atoms a_1 and a_2,

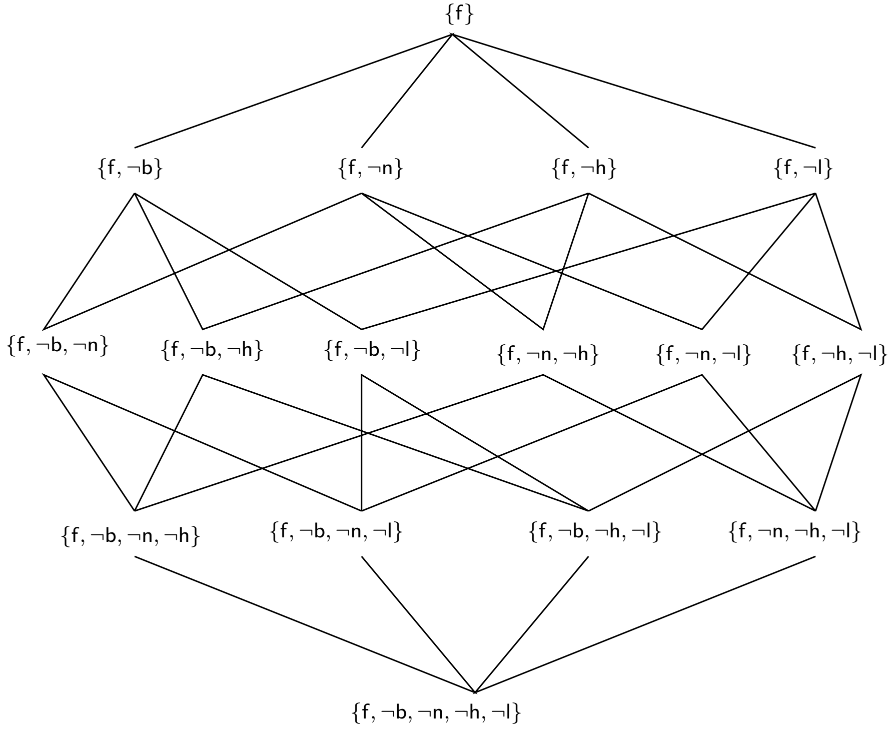


Fig. 5.1. The lattice of propositional definite clauses with head atom f. We use the following abbreviations: f = flies, b = bird, n = normal, h = hasWings and l = laysEggs

$$a_1 \text{ subsumes } a_2, \text{if and only if}, \exists \text{ substitution } \theta : a_1\theta = a_2 \qquad (5.7)$$

Example 5.5. p(X, Y, a) subsumes p(a, b, a) (with substitution θ ={X/a, Y/b} but does not subsume p(a, Y, Z).

The resulting structure on the space of atoms is depicted in Fig. 5.3. It still has very nice properties. Nevertheless, compared to propositional subsumption, two complications arise. First, as illustrated in the figure, when using functors, infinite chains occur in the subsumption lattice. Indeed, consider p(X), p(f(X1)), p(f(f(X2)), ... Second, there exist syntactic variants. Recall that two hypotheses are syntactic variants when they are syntactically different but cover the same set of instances. For example, p(X) and p(Y) are syntactic variants because there exist substitutions θ_1={X/Y} and θ_2={Y/X} such that p(X)θ_1 = p(Y) and p(Y)θ_2 = p(X). Therefore, p(X) and p(Y) are equivalent with regard to subsumption, and hence the set of instances covered by p(X) and p(Y) is the same. Fortunately, one can show that two atoms can be syntactic variants only if they are a variable renaming of one another. An expression e is a variable renaming of e' if and only if there exist substitutions

$\theta = \{V_1/W_1, ..., V_n/W_n\}$ and $\theta' = \{W_1/V_1, ..., W_n/V_n\}$ where the V_i and W_i are different variables appearing in e and e', such that $e\theta = e'$ and $e'\theta' = e$. It can be shown that the resulting structure on the search space is again a lattice up to variable renaming (when adding a bottom $\perp$ element).

Let us now introduce the different operators for working with atoms.

5.3.1 Specialization Operators

An Ideal Specialization Operator

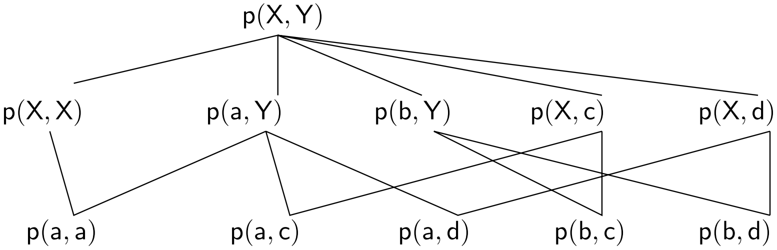


Fig. 5.2. Part of the lattice on atoms

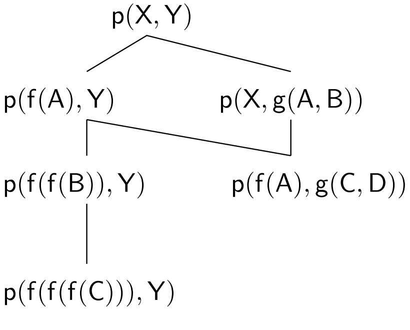


Fig. 5.3. Part of the lattice on atoms

Let A be an atom. Then

$$\rho_{s,a,i}(A) = \{A\theta \mid \theta \text{ is an elementary substitution}\} \tag{5.8}$$

where an elementary substitution θ is of the form

$$\theta = \begin{cases} \{X/f(X_1, ..., X_n)\} & \text{with } f \text{ a functor of arity } n \text{ and} \\ & \quad \text{the } X_i \text{ variables not occurring in } A \\ \{X/c\} & \text{with } c \text{ a constant} \\ \{X/Y\} & \text{with } X \text{ and } Y \text{ variables occurring in } A \end{cases} \tag{5.9}$$

The subscript s in the operator stands for specialization, a for atoms, and i for ideal. It is relatively easy to see that $\rho_{s,a,i}$ is an ideal specialization operator for atoms.

An Optimal Specialization Operator*

Obtaining an optimal operator for atoms is a little bit harder. The reason for this is illustrated in Figs. 5.4, 5.5 and 5.6. As one can see, even if one naively applies one type of elementary substitution there exist many different ways of generating one particular atom. These redundancies can be avoided by further restricting the elementary substitutions.

The operator $\rho_{s,a,o}$ is an optimal specialization operator for atoms A, defined as follows:

$$\rho_{s,a,o}(A) = \{A\theta \mid \theta \text{ is an optimal elementary substitution}\} \tag{5.10}$$

where an elementary substitution θ is an *optimal elementary substitution* for an atom A if and only if it is of the form

$$\theta = \begin{cases} \{X/f(X_1, ..., X_n)\} & \text{with } f \text{ a functor of arity } n \text{ and} \\ & \quad \text{the } X_i \text{ variables not occurring in } A \\ & \quad \text{and all terms occurring to the right of} \\ & \quad \text{the leftmost occurrence of } X \\ & \quad \text{variables or constants} \\ \{X/c\} & \text{with } c \text{ a constant and no term} \\ & \quad \text{to the right of the leftmost occurrence of } X \\ & \quad \text{containing constants} \\ \{X/Y\} & \text{with } X \text{ and } Y \text{ variables occurring in } A \\ & \quad X \text{ occurring once, } Y \text{ occurring left of } X \\ & \quad \text{and all variables to the right of } X \\ & \quad \text{occurring only once in } A \end{cases} \tag{5.11}$$

At this point the reader may want to verify that these optimal elementary substitutions avoid the problems sketched in the figures. However, one more problem exists. If any sequence of elementary substitutions is allowed, further redundancies exist.

Example 5.6. Consider the atom $\mathsf{p(X, Y)}$. Now, one can apply the substitutions $\theta_1 = \{\mathsf{X/Y}\}$ and $\theta_2 = \{\mathsf{Y/a}\} \leftarrow$ resulting in $\mathsf{p(a, a)}$. On the other hand, one could also apply the substitutions $\theta'_1 = \{\mathsf{X/a}\}$ and $\theta'_2 = \{\mathsf{Y/a}\}$, which give the same result.

This problem disappears when requiring that all optimal elementary substitutions of type $\{X/f(X_1,...,X_n)\}$ be applied before those of type $\{X/c\}$, which be applied before those of type $\{X/Y\}$.

Exercise 5.7. * If one orders the optimal elementary substitutions differently, is the resulting operator still optimal?

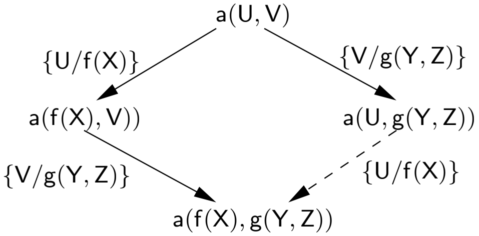


Fig. 5.4. Example of duplicate avoidance for substitutions of type $\{X/f(X_1,\ldots,X_n)\}$ (adapted from [Lee, 2006])

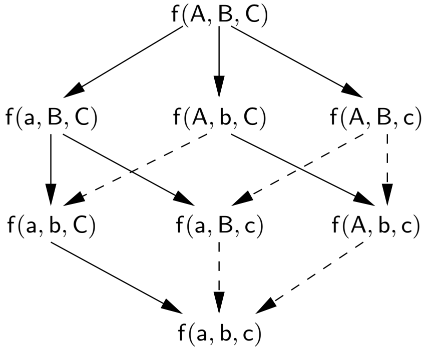


Fig. 5.5. Example of duplicate avoidance for substitutions of type $\{X/c\}$ from [Lee, 2006]

5.3.2 Generalization Operators*

In order to be able to introduce a generalization operator, we need the important notion of an *inverse* substitution θ^{-1}. As the term suggests, an inverse substitution inverts a substitution. This is in line with the previously introduced view of induction as the inverse of deduction. Applying a substitution

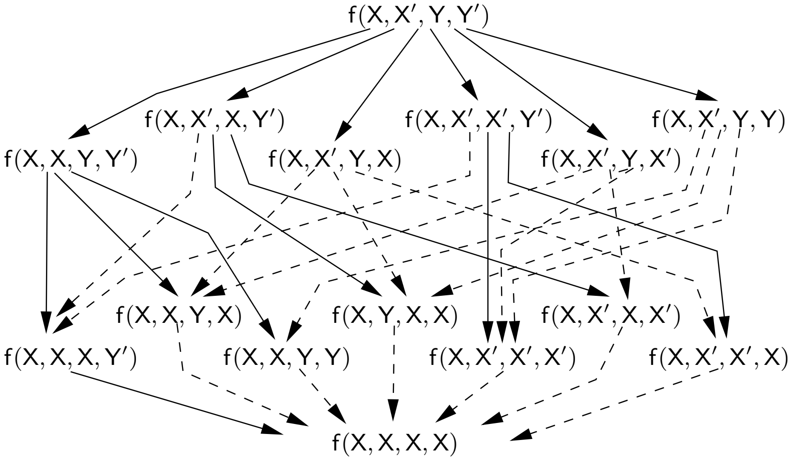


Fig. 5.6. Example of duplicate avoidance for substitutions of type $\{X/Y\}$ (adapted from [Lee, 2006])

is a deductive operation; applying an inverse substitution is an inductive operation.

An inverse substitution θ^{-1} (corresponding to the substitution θ) satisfies

$$A\theta\theta^{-1} = A \tag{5.12}$$

Whereas regular substitutions substitute terms for variables, inverse substitutions substitute variables for terms. However, because a term can occur more than once in an atom (or a logical expression), we need to distinguish the different occurrences of a term. To this end, one employs *positions* (sometimes also called *places*). With each occurrence of a term t in an atom $p(t_1, ..., t_n)$, a position is associated as follows:

$$\begin{array}{ll} < i > & \text{if } t = t_i \\ < i, i_1, ..., i_m > & \text{if } t \text{ is a sub-term of } t_i \text{ at position } < i_1, ..., i_m > \end{array} \tag{5.13}$$

Example 5.8. Consider $\mathsf{p(f(a), a, a)}$. The term a occurs at positions $< 1, 1 >$, $< 2 >$ and $< 3 >$ in the atom.

An *inverse substitution* θ^{-1} now substitutes terms at specified positions with variables. Thus, formally speaking, an *inverse substitution* is of the form

$$\{p_1/V_1, ...p_k/V_k\} \text{ with positions } p_i \text{ and variables } V_i \tag{5.14}$$

Example 5.9. Let $A = \mathsf{p(f(a), a, a)}$. Applying inverse substitution

$$\theta^{-1} = \{< 1, 1 > /\mathsf{X}, < 3 > / \ \mathsf{Y}\}$$

to A yields $A\theta^{-1} = \mathsf{p(f(X), a, Y)}$.

By now we are able to introduce the ideal generalization operator $\rho_{g,a,i}$ that inverts the previously introduced $\rho_{s,a,i}$ for logical atoms.

Let A be an atom. Then

$$\rho_{g,a,i}(A) = \{A\theta^{-1} \mid \theta^{-1} \text{ is an elementary inverse substitution}\} \tag{5.15}$$

where an elementary inverse substitution θ^{-1} is of the form $\{p_1/Y, ..., p_n/Y\}$ where Y is a variable not occurring in A and the term t occurring at positions $p_1, ..., p_n$ in A is of the following form:

$$t = \begin{cases} f(Y_1, ..., Y_m) & \text{where } Y_1, ..., Y_m \text{ are distinct variables only occurring} \\ & \quad \text{in } t \text{ and } t \text{ occurring exactly } n \text{ times in } A; \\ c & \text{where } c \text{ is a constant} \\ X & \text{where } X \text{ is variable occurring at least } n+1 \text{ times in } A; \end{cases} \tag{5.16}$$

Example 5.10. Applying the elementary inverse substitutions to $\mathsf{p(a, f(b))}$ yields $\mathsf{p(X, f(b))}$ with $\{< 1 > /\mathsf{X}\}$ and $\mathsf{p(a, f(X))}$ with $\{< 2, 1 > /\mathsf{X}\}$.

Exercise 5.11. * Assume you are given an atom A and a substitution θ. How can one, in general, obtain the inverse substitutions $A\theta^{-1}$?

The operators $\rho_{g,a,i}$ and $\rho_{s,a,i}$ are ideal for the class of atoms (up to variable renaming). The dual of the operator $\rho_{s,a,o}$, that is, the operator $\rho_{g,a,o}$, is not presented because such operators are not commonly applied.

5.3.3 Computing the lgg and the glb

We claimed above that subsumption at the level of atoms induces a complete lattice (up to variable renaming) on the search space. By definition, a complete lattice has a unique least upper bound and a unique greatest lower bound for any two elements. The *greatest lower bound* $glb(a_1, a_2)$ of two atoms a_1 and a_2 starting with the same predicate symbol p is

$$glb(a_1, a_2) = a_1\theta = a_2\theta \text{ where } \theta = mgu(a_1, a_2) \tag{5.17}$$

So, the *glb* corresponds to *unification*, the operation introduced in Sect. 2.4

The dual operation is the least general generalization (*lgg*) operation. It is sometimes referred to as *anti-unification*. The *least general generalization* $lgg(a_1, a_2)$ of two atoms a_1 and a_2 starting with the same predicate symbol p is a generalization a of a_1 and a_2 such that for all other generalizations b of a_1 and a_2 there exists a substitution θ such that $b\theta = a$ and $b\theta$ and a are not variable renamings.

It is well known that the set of all atoms, partially ordered by the subsumption relation (and extended by special $\top$ and $\bot$ elements), forms a complete lattice as the *lgg* and the *glb* exist and are unique for all pairs of atoms. The special elements $\top$ and $\bot$ are used for those cases where the *glb* or *lgg* of two atoms are not defined, that is, $\bot$ is the result of the *glb* operator when the two atoms are not unifiable, and $\top$ is the result of the *lgg* operator when the two atoms do not start with the same predicate symbol.

Example 5.12. Consider the atoms $\mathsf{p(a, b, f(a))}$ and $\mathsf{p(c, b, f(c))}$. The atom $\mathsf{p(X, b, f(X))}$ is the least general generalization of these two terms, as it is a generalization of the first atom (with substitution $\{\mathsf{X/b}\}$) and of the second atom (with substitution $\{\mathsf{X/c}\}$), and furthermore, for all other generalizations, for instance, $\mathsf{p(U, b, f(W))}$, there exists a substitution $\theta = \{\mathsf{U/X, W/X}\}$ for which $\mathsf{p(U, b, f(W))}\theta = \mathsf{p(X, b, f(X))}$.

The computation of the least general generalization bears some similarities with the unification algorithm shown in Algo. 2.3. The anti-unification algorithm is depicted in Algo. 5.1.

Algorithm 5.1 Computing the $lgg(a_1, a_2)$ of two atoms a_1 and a_2 following [Nienhuys-Cheng and de Wolf, 1997]

```
$\theta_1 := \emptyset$; $\theta_2 := \emptyset$;
while $a_1 \neq a_2$ do
    find the leftmost position $p$ where $a_1$ and $a_2$ differ
    let $s$ and $t$ be the terms at position $p$ in $a_1$ and $a_2$
    if $\{V/s\}$ occurs in $\theta_1$ and $\{V/t\}$ occurs in $\theta_2$ then
        replace the terms $s$ and $t$ at position $p$ in $a_1$ and $a_2$ by $V$
    else
        replace the terms $s$ and $t$ at position $p$ in $a_1$ and $a_2$ by a new variable $W$
        $\theta_1 := \theta_1\{W/s\}$; $\theta_2 := \theta_2\{W/t\}$
    end if
end while
return $a_1$
```

The algorithm repeatedly finds the position p where the two atoms a_1 and a_2 disagree, and replaces the terms s and t at the corresponding positions by a variable. If the terms s and t were already encountered before and replaced by a variable V, the same variable is used; otherwise a new variable W not yet occurring in the atoms is used.

Example 5.13. To illustrate the operation of the algorithm, consider the terms $\mathsf{p(a, b, f(a), c)}$ and $\mathsf{p(c, b, f(c), d)}$. Then the variables at the end of each iteration obtain the following values:

1. $\theta_1 = \emptyset$, $\theta_2 = \emptyset$, $a_1 = \mathsf{p(a, b, f(a), c)}$, and $a_2 = \mathsf{p(c, b, f(c), d)}$;
2. $p = \ < 1 >$, $s = \mathsf{a}$, $t = \mathsf{b}$, and hence $a_1 = \mathsf{p(V, b, f(a), c)}$, $a_2 = \mathsf{p(V, b, f(c), d)}$, $\theta_1 = \{\mathsf{V/a}\}$ and $\theta_2 = \{\mathsf{V/b}\}$;
3. $p = \ < 1, 3 >$, $s = \mathsf{a}$, $t = \mathsf{b}$, and hence $a_1 = \mathsf{p(V, b, f(V), c)}$, $a_2 = \mathsf{p(V, b, f(V), d)}$, $\theta_1 = \{\mathsf{V/a}\}$ and $\theta_2 = \{\mathsf{V/b}\}$;

4. $p = \quad < 4 >$, $s = \mathsf{c}$, $t = \mathsf{d}$, and hence $a_1 = \mathsf{p(V,b,f(V),W)}$, $a_2 = \mathsf{p(V,b,f(V),W)}$, $\theta_1 = \{\mathsf{V/a,W/c}\}$ and $\theta_2 = \{\mathsf{V/b,W/d}\}$;
5. at this point $a_1 = a_2$, and hence the algorithm terminates and outputs $\mathsf{p(V,b,f(V),W)}$.

5.4 *Θ*-Subsumption

Θ-subsumption is the most important framework for generalization and specialization in inductive logic programming. Almost all inductive logic programming systems use it in one form or another.

Θ-subsumption combines propositional subsumption with subsumption at the level of logical atoms. It is defined as follows. Clause c_1 *θ-subsumes* clause c_2 if and only if

$$\exists \text{ substitution } \theta : c_1\theta \subseteq c_2 \tag{5.18}$$

Example 5.14. The clause

$$\mathsf{father(X,john) \leftarrow male(X), male(john), parent(X,john)}$$

is θ-subsumed (with substitution $\{\mathsf{Y/X, Z/john}\}$) by

$$\mathsf{father(Y,Z) \leftarrow male(Y), parent(Y,Z))}$$

The clause

$$\mathsf{nat(s(X)) \leftarrow nat(X)}$$

θ-subsumes (with substitution) $\{\mathsf{X/s(0)}\}$

$$\mathsf{nat(s(s(0))) \leftarrow nat(s(0))}$$

and

$$\mathsf{p(X,Y,X) \leftarrow q(Y)}$$

θ-subsumes (with substitution $\{\mathsf{X/U, Y/U}\}$)

$$\mathsf{p(U,U,U) \leftarrow q(U), r(a)}$$

Finally, the clause

$$\mathsf{q(X) \leftarrow p(X,Y), p(Y,X)}$$

θ-subsumes (with substitution $\{\mathsf{X/A, Y/A}\}$)

$$\mathsf{q(A) \leftarrow p(A,A).}$$

The last example is surprising in that the clause $\mathsf{q(X) \leftarrow p(X,Y), p(Y,X)}$ is both more general and longer than the clause $\mathsf{q(A) \leftarrow p(A,A)}$. In machine learning, it is typical that longer clauses are more specific. When working with θ-subsumption, this is not always the case.

There are various interesting properties of θ-subsumption, which we will now discuss.

5.4.1 Soundness and Completeness

Proposition 5.1 stated that a hypothesis g is more general than a hypothesis s (when learning from entailment) when $g \models s$. Θ-subsumption considers hypotheses that consist of single clauses. A natural question that arises in this context is to what extent is θ-subsumption *sound* and *complete* w.r.t. logical entailment among single clauses. First, θ-subsumption is sound w.r.t. logical entailment, that is,

$$\forall c_1, c_2 : (c_1 \theta\text{-subsumes } c_2) \rightarrow (c_1 \models c_2) \tag{5.19}$$

The reverse property, that is, completeness, does not hold. Indeed, as shown in the example below, θ-subsumption is incomplete. Fortunately, it is only incomplete for self-recursive clauses. These are clauses which resolve with themselves.

Example 5.15. Consider c_1: $\mathsf{nat(s(Y)) \leftarrow nat(Y)}$ and c_2: $\mathsf{nat(s(s(X))) \leftarrow nat(X)}$. Then $c_1 \models c_2$ but c_1 does not θ-subsume c_2. Indeed, $c_1 \models c_2$ can be proved by resolution; cf. Fig. 5.7. Furthermore, there is no substitution that makes c_1 θ-subsume c_2. The first literal indicates that Y should be substituted by $\mathsf{s(X)}$ but the second literal requires Y to be substituted by X. These constraints are unsatisfiable.

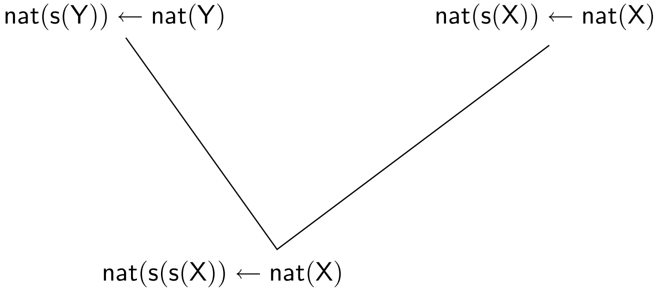


Fig. 5.7. Proving $c_1 \models c_2$ using resolution with substitution $\{\mathsf{Y/s(X)}\}$

5.4.2 Deciding Θ-Subsumption

To decide whether a clause c_1 θ-subsumes c_2, Algo. 5.2 can be be employed. The algorithm[1] works as follows: if c_1 is the empty clause, it clearly subsumes c_2, and hence returns success; otherwise, it first skolemizes the clause c_2; that is, it replaces all variables in c_2 by constants not occurring in c_2 (and c_1);

[1] Algo. 5.2 has a short, though tricky implementation in Prolog, where the clauses are represented as lists.

and then tries to eliminate a literal l_1 from c_1 that subsumes a literal in c_2; if this is possible it recursively calls the function again; otherwise, it backtracks to earlier choice points. This process continues until either the empty clause is generated (meaning that all literals from the initial clause c_1 have been matched with literals from c_2) and success is returned, or all choice points have been considered and failure is returned.

Example 5.16. To test whether clause c_1

$$\mathsf{p(X,Y,X)} \leftarrow \mathsf{q(X)}$$

θ-subsumes the clause c_2

$$\mathsf{p(X,a,X)} \leftarrow \mathsf{q(X)}$$

the algorithm first skolemizes c_2, yielding $c_2\sigma$

$$\mathsf{p(sk,a,sk)} \leftarrow \mathsf{q(sk)}.$$

It then tests whether the literal $\mathsf{q(X)} \in c_1$ subsumes a literal in $c_2\sigma$. Since it does (with substitution θ=$\{\mathsf{X/sk}\}$), the procedure is called recursively; now with the clause $\mathsf{p(sk,Y,sk)} \leftarrow$. The algorithm then considers the literal $\mathsf{p(sk,Y,sk)}$ and tests whether it subsumes a literal in $c_2\sigma$. Since it does (with substitution $\{\mathsf{Y/a}\}$), the resulting clause is empty, and hence the algorithm concludes that c_1 θ-subsumes c_2.

Due to the backtracking step in Algo. 5.2, testing for θ-subsumption is computationally expensive. Indeed, θ-subsumption is an NP-complete problem [Garey and Johnson, 1979]. A related result that is often overlooked, and that can be considered an immediate consequence of the NP-completeness of θ-subsumption, is that coverage testing in relational data mining is also computationally expensive. Indeed, the problem of deciding whether a (non-recursive) clause $p \leftarrow q_1, \ldots, q_n$ covers a particular example, say the fact $p\theta$, in an extensional relational database consisting of the facts $f_1, \ldots, f_m$ only, can be reduced to deciding whether $p\theta \leftarrow q_1\theta, ..., q_n\theta$ θ-subsumes $p\theta \leftarrow f_1, \ldots, f_m$. Vice versa, the problem of deciding whether the clause $p_1 \leftarrow q_1, \ldots, q_n$ θ-subsumes the clause $p \leftarrow f_1, \ldots, f_m$ can be reduced to the problem of deciding whether the clause $p_1 \leftarrow q_1, \ldots, q_n$ covers the example $p\theta$ in the extensional database $f_1\theta, \ldots, f_m\theta$, where θ is a substitution skolemizing the clause $p \leftarrow f_1, \ldots, f_m$. This provides an alternative to Algo. 5.2 for deciding θ-subsumption.

```
thetasubsumes(Clause1, Clause2) ←
   not not (numbervars(Clause2, 999, N), subsetof(Clause1, Clause2)).
```

The double negation is employed in order to avoid binding the variables in the clauses; numbervars is a built-in predicate that performs the skolemization, numbering the variables from 0 to 999, and subsetof succeeds when all elements of Clause1 occur in Clause2.

Algorithm 5.2 The function **θ-subsumes**(c_1, c_2: clauses), which decides whether c_1 θ-subsumes c_2

```
if $c_1 = \emptyset$ then
  return success
else
  skolemize $c_2$
  let $l_1$ be the first literal in $c_1$
  for each literal $l_2$ in $c_2$ that is an instance of $l_1$ (choice point) do
    let $\theta$ be the substitution such that $l_1\theta = l_2$
    if $\theta$-subsumes($c_1\theta - \{l_1\theta\}, c_2$) succeeds then
      return success ($c_1$ $\theta$-subsumes $c_2$)
    else
      backtrack to the last choice point
    end if
  end for
  return failure ($c_1$ does not $\theta$-subsume $c_2$)
end if
```

Example 5.17. Let us illustrate this on a simple example. Consider testing whether the example pos(e1) ← is covered by the clause

$$\mathsf{pos(E) \leftarrow triangle(E,T), circle(E,C), in(E,T,C)}$$

given the extensional background theory containing the following facts:

$$\begin{array}{ll} \mathsf{triangle(e1,t1) \leftarrow} & \mathsf{triangle(e1,t2) \leftarrow} \\ \mathsf{circle(e1,c1) \leftarrow} & \mathsf{square(e1,s1) \leftarrow} \\ \mathsf{in(e1,t2,c1) \leftarrow} & \ldots \end{array}$$

The example pos(e1) is covered by the clause if and only if the clause

$$\mathsf{pos(e1) \leftarrow triangle(e1,T), circle(e1,C), in(e1,T,C)}$$

θ-subsumes

$$\begin{array}{l} \mathsf{pos(e1) \leftarrow} \\ \qquad \mathsf{triangle(e1,t1), triangle(e1,t2), circle(e1,c1),} \\ \qquad \mathsf{square(e1,s1), in(e1,t2,c1), \ldots} \end{array}$$

Vice versa, testing whether

$$\mathsf{pos(E) \leftarrow triangle(E,T), circle(E,C), in(E,T,C)}$$

θ-subsumes

$$\mathsf{pos(E) \leftarrow triangle(E,t1), circle(E,C), in(E,t1,C), square(E,S)}$$

can be realized by testing whether the example pos(ske) is covered by the clause

$$\mathsf{pos(E) \leftarrow triangle(E,T), circle(E,C), in(E,T,C)}$$

given the background theory

$$\begin{array}{ll} \mathsf{triangle(ske,t1)} \leftarrow & \mathsf{circle(ske,skc)} \leftarrow \\ \mathsf{in(ske,t1,skc)} \leftarrow & \mathsf{square(ske,sks)} \leftarrow \end{array}$$

The above discussion shows that θ-subsumption and extensional coverage testing are intimately related and belong to the same complexity class. Because θ-subsumption is NP-complete, so is extensional coverage testing.[2] Because the two problems are closely related, the number of such tests carried out by a logical or relational learning system can be used as a measure of the efficiency of the learning system. It should be clear that the number of such tests should be as small as possible. At the same time, it is important that the implementations of such tests be optimized where possible. Various techniques for realizing this are presented in Chapter 10.

5.4.3 Equivalence Classes

When analyzing the properties of θ-subsumption, it is quite easy to see that θ-subsumption is both reflexive (take the empty substitution) and transitive (compose the substitutions). Unfortunately, it is not anti-symmetric because there exist syntactic variants, which are not restricted to variable renamings.

Example 5.18. The following clauses, syntactically different, are equivalent under θ-subsumption, and are therefore syntactic variants.

$$\begin{array}{l} \mathsf{parent(X,Y)} \leftarrow \mathsf{mother(X,Y)} \\ \mathsf{parent(X,Y)} \leftarrow \mathsf{mother(X,Y), mother(X,Z_1)} \\ \mathsf{parent(X,Y)} \leftarrow \mathsf{mother(X,Y), mother(X,Z_1), mother(X,Z_2)} \end{array}$$

All these clauses are equivalent under θ-subsumption and therefore (due to the soundness of θ-subsumption) also logically equivalent. This can be quite problematic when defining operators. Some of the early inductive logic programming systems get into infinite loops because of this problem. They start from the first clause, refine it into the second, the third, and so on, and might never recover.

Because θ-subsumption is reflexive and transitive, but not anti-symmetric, it is a quasi-order. The standard mathematical approach to turn a quasi-order into a partial one is to define equivalence classes and study the quotient set. This is also what Gordon Plotkin did for θ-subsumption. Let us denote by $[c]$ the set of clauses equivalent to c under θ-subsumption. The quotient set is then the set of all equivalence classes $[c]$. The θ-subsumption relation induced a relation on the quotient set, and this quotient relation has various interesting properties. The first such property concerns the existence of a

[2] Furthermore, given that the typical size of the relations in a database is much larger than that of the typical clause, testing coverage is typically more expensive than testing generality under θ-subsumption.

unique representative (up to variable renaming) of each equivalence class. This representative is termed the *reduced* clause. The reduced clause r of a clause c is defined as the shortest clause $r \subseteq c$ that is equivalent to c. It is unique up to variable renaming. Algo. 5.3 starts from a given clause c and computes the reduced clause of the equivalence class $[c]$.

Algorithm 5.3 Reducing a clause under θ-subsumption

```
$r := c$
for all literals $l \in r$ do
  if $r$ $\theta$-subsumes $r - \{l\}$ then
    $r := r - \{l\}$
  end if
end for
return $r$
```

Example 5.19. Assume the algorithm is called with the clause

$$\mathsf{parent}(\mathsf{X},\mathsf{Y}) \leftarrow \mathsf{mother}(\mathsf{X},\mathsf{Y}), \mathsf{mother}(\mathsf{X},\mathsf{Z_1}), \mathsf{mother}(\mathsf{X},\mathsf{Z_2})$$

and assume the literals are processed from right to left. Then the algorithm will first delete the literal $\mathsf{mother}(\mathsf{X},\mathsf{Z_2})$ and then the literal $\mathsf{mother}(\mathsf{X},\mathsf{Z_1})$, after which no further literals can be deleted while staying within the same equivalence class.

A second property of the resulting quotient relation (as well as of the original relation) is that there exist infinite ascending and descending chains in the partial order.

Example 5.20. The following clauses form an ascending chain (where $n \geqslant 2$); cf. [Nienhuys-Cheng and de Wolf, 1997]:

$$c_n = \{\neg \mathsf{p}(\mathsf{X_i},\mathsf{X_j}) \mid i \neq j \text{ and } 1 \leqslant i,j \leqslant n\} \tag{5.20}$$

For instance,

$$\begin{array}{l} c_2 = \leftarrow \mathsf{p}(\mathsf{X_1},\mathsf{X_2}), \mathsf{p}(\mathsf{X_2},\mathsf{X_1}) \\ c_3 = \leftarrow \mathsf{p}(\mathsf{X_1},\mathsf{X_2}), \mathsf{p}(\mathsf{X_2},\mathsf{X_1}), \mathsf{p}(\mathsf{X_1},\mathsf{X_3}), \mathsf{p}(\mathsf{X_3},\mathsf{X_1}), \mathsf{p}(\mathsf{X_2},\mathsf{X_3}), \mathsf{p}(\mathsf{X_3},\mathsf{X_2}) \\ \ldots \end{array}$$

One can prove that $c_i \prec c_{i+1}$ (that is, c_i θ-subsumes c_{i+1} and c_{i+1} does not θ-subsume c_i) for all $i \geqslant 2$; cf. [Nienhuys-Cheng and de Wolf, 1997]. This means that the c_i belong to different equivalence classes. Furthermore, each c_i θ-subsumes $c = \leftarrow \mathsf{p}(\mathsf{X},\mathsf{X})$. So, we have an infinite ascending chain in the partial order because $c_2 \prec c_3 \prec \ldots c_\infty \prec c$.

There also exist infinite descending chains in the partial order though the corresponding series are even more complicated. For this direction, one can construct, for instance, an infinite descending chain starting at $d = \leftarrow$ $\mathsf{p}(\mathsf{X}_1, \mathsf{X}_2), \mathsf{p}(\mathsf{X}_2, \mathsf{X}_1)$; cf. [Nienhuys-Cheng and de Wolf, 1997].

The third property is a consequence of the second. Due to the existence of these infinite ascending and descending chains which are bounded from below or from above, there exist neither ideal nor optimal refinement operators for θ-subsumption. The problem is due to clauses such as c and d above. For example, consider that one wants to minimally generalize c. Given the infinite chain, one of the minimal generalizations of c would be c_∞. Now, c_∞ is not a clause because it has an infinite number of literals. Even if it were a clause, it could not be computed by an algorithm. As a consequence there exist neither computable ideal nor optimal refinement operators for θ-subsumption.

In order to deal with this problem, one somehow has to relax the conditions imposed on ideal and optimal refinement operators. One can either work with a finite hypothesis language (as this excludes the existence of such infinite chains) or relax the condition that the refinements $\rho(c)$ of a clause c must be proper, minimal or complete.

Most logical and relational learning systems address this problem in a more pragmatic manner (see also Sect. 5.5.1 on object identity). Simple refinement operators are then used, for example:

$$\rho_{s,i,\theta}(c) = \begin{cases} c\theta & \text{with } \theta \text{ an elementary substitution} \\ c \cup \{l\} & \text{with } l \text{ an elementary literal} \end{cases} \tag{5.21}$$

A literal l is *elementary* w.r.t. c if and only if it is of the form

$$p(X_1, ..., X_n) \text{ with the } X_i \text{ different variables not occurring in } c \tag{5.22}$$

There exist many variants of such operators. Most of them take into account certain syntactic restrictions, that is, a *syntactic bias*, on the form of the clauses to be obtained. Syntactic biases are discussed in depth in Sect. 6.6. In Fig. 5.8, we show part of the refinement graph obtained using a refinement operator that only considers *linked* clauses (these are clauses where each literal in the clause contains at least one variable already introduced earlier in the clause).

One can now also invert the operator $\rho_{s,i,\theta}$, in order to obtain a generalization operator. We leave this as an exercise to the reader.

Exercise 5.21. Invert the pragmatic specialization operator.

The fourth property of the quotient relation is that it forms a complete lattice. Its structure is summarized in Fig. 5.9. Because the quotient set is a lattice, there exist for any two equivalence classes $[c_1]$ and $[c_2]$ a unique *lgg* and *glb* in the quotient set. A representative of $lgg([c_1], [c_2])$ and $glb([c_1], [c_2])$ can be computed starting from the clauses c_1 and c_2 themselves. These operations are the notions of *lgg* and *glb* typically found in the inductive logic programming literature.

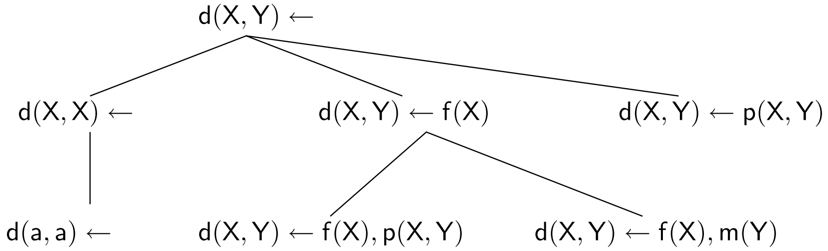


Fig. 5.8. Applying a pragmatic refinement operator under θ-subsumption

$$lgg(c_1, c_2) = \{lgg(l_1, l_2) \mid l_1 \in c_1 \wedge l_2 \in c_2 \wedge lgg(l_1, l_2) \neq \top\} \tag{5.23}$$

This definition implicitly assumes that if two terms s and t are to be replaced by a variable V at position p in literals l_1 and l_2, and if they occur at position p' in literals l'_1 and l'_2, then they will be replaced by the same variable V throughout; cf. Algo. 5.1.

Example 5.22. Let c_1 and c_2 be

$$\mathsf{father(jef, ann) \leftarrow parent(jef, ann), female(ann), male(jef)}$$
$$\mathsf{father(jef, tom) \leftarrow parent(jef, tom), male(jef), male(tom)}$$

Then $lgg(c_1, c_2)$ is

$$\mathsf{father(jef, AT) \leftarrow parent(jef, AT), male(jef), male(JT)}$$

This clause can be further reduced to yield

$$\mathsf{father(jef, AT) \leftarrow parent(jef, AT), male(jef)}$$

From the definition of the lgg it follows that, in the worst case, the number of literals in the lgg can be $O(n \times m)$ where n and m are the size of the original clauses. When computing the lgg with regard to k examples of size n the lgg can be of size $O(n^k)$, which is exponential in the number of examples, and therefore problematic from a computational point of view. Another problem is that the lgg of two clauses, as computed above, is itself a clause, which may not be reduced. Therefore, after each computation of the lgg, one may want to reduce the result with regard to θ-subsumption. However, as we have mentioned earlier, this is again computationally expensive due to the need to carry out a number of θ-subsumption tests.

The operation that is dual to the lgg is the glb. The glb of two clauses c_1 and c_2 (which do not share any variables) is defined as:

$$glb(c_1, c_2) = c_1 \cup c_2 \tag{5.24}$$

It is mainly important for theoretical reasons and not so much used in practice. One of the reasons for not using it in practice is that the glb of two

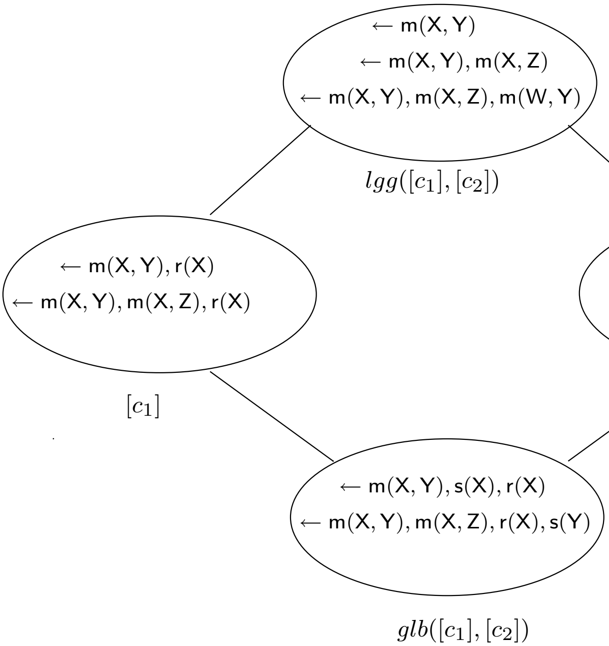


Fig. 5.9. The θ-subsumption lattice on clauses.

definite clauses (these are clauses with exactly one literal in the head) may be non-Horn (with more than one literal in the head), even after reduction.

Given the importance of the θ-subsumption framework, it is no surprise that several variants have been introduced. We study two such variants in the next section, which can be skipped without loss of continuity. These variants address particular shortcomings of θ-subsumption. First, OI-subsumption eliminates syntactic variants from the search space and possesses optimal and ideal refinement operators. Second, the framework of (inverse) implication addresses the incompleteness of θ-subsumption with regard to logical entailment.

5.5 Variants of Θ-Subsumption*

5.5.1 Object Identity*

A recent and promising variant of θ-subsumption that solves some of the problems with θ-subsumption is given by the object identity framework. One of the difficulties with θ-subsumption is that a longer clause c_1 can θ-subsume a shorter one c_2. This may occur when there exist substitutions θ for which several literals in c_1 collapse into one; see Ex. 5.14 for an example. Object identity prevents this because it requires that all terms in a clause be different.

Let us now introduce OI-subsumption formally. For simplicity, we define it for functor-free clauses only. When working under OI-subsumption, each clause c is *completed* to the clause $com(c)$,

$$com(c) = c \cup \{t_i \neq t_j \mid t_i \text{ and } t_j \text{ are two distinct terms in } c\} \tag{5.25}$$

Example 5.23. The completion of the clause

$$\mathsf{p(X, X) \leftarrow q(X, X), r(Y, a)}$$

is

$$\mathsf{p(X, X) \leftarrow q(X, X), r(Y, a), X \neq Y, X \neq a, Y \neq a}$$

Given two clauses g and s, g *OI-subsumes* s if and only if there exists a substitution θ such that

$$com(g)\theta \subseteq com(s) \tag{5.26}$$

We list some interesting properties of OI-subsumption.

$$\forall g, s : g\ OI\text{-subsumes } s \rightarrow g\theta\text{-subsumes } s \tag{5.27}$$

This property states that OI-subsumption is a weaker form of subsumption than θ-subsumption. It is actually strictly weaker, as illustrated by the next example.

Example 5.24. Consider the clauses

$$\begin{aligned} c_1 =& \mathsf{p(X, X) \leftarrow q(X, X), r(Y, a)} \\ c_2 =& \mathsf{p(X, X) \leftarrow q(X, X), r(Y, a), t(V)} \end{aligned}$$

Clearly c_1 subsumes c_2 with regard to both OI-subsumption and θ-subsumption. However, the clause

$$c_3 = \mathsf{p(X, X) \leftarrow q(X, T), q(T, X)}$$

θ-subsumes c_1 but does not OI-subsume it (due to the constraint $\mathsf{X} \neq \mathsf{T}$ in $com(c_3)$).

The second property states that two clauses g and s are equivalent under OI-subsumption if and only if g and s are variable renamings. This property actually implies that the only syntactic variants are variable renamings. This eliminates the need for reducing clauses, which is an expensive process under θ-subsumption. At the same time, it enables one to define ideal and optimal refinement operators under OI-subsumption. The operator $\rho_{s,i,OI}$ is ideal under OI-subsumption when working with functor-free Datalog clauses:

$$\rho_{s,i,OI}(c) = \begin{cases} c\theta & \text{with } \theta \text{ a substitution of the form } \{X/a\}, \\ & \quad X \text{ a variable in } c, \text{ and} \\ & \quad a \text{ a constant not occurring in } c \\ c \cup \{l\} & \text{with } l \notin c \text{ a negated atom} \end{cases} \tag{5.28}$$

So, the only operators allowed substitute a variable with a constant or add an atom to the body of the clause. Unifying two variables is not allowed as this violates the object identity the assumption. From this operator, various other operators under OI-subsumption can be derived, including an optimal one.

Whereas the key advantage of OI-subsumption is that the structure on the search space is simpler, there are also some complications that arise. First, OI-subsumption is weaker than θ-subsumption, and is therefore also a weaker approximation of logical implication. Second, evaluating whether a certain hypothesis covers an example will typically be more expensive because the process of computing the completion $com(c)$ of a clause c introduces further constraints that must be satisfied. As the efficiency of coverage tests depends on the length of the clauses, working with OI-subsumption may lead to less efficient covers tests. For example, consider the clause $\mathsf{p(X)} \leftarrow \mathsf{q(X), r(Y)}$. Under θ-subsumption, the literals $\mathsf{q(X)}$ and $\mathsf{r(Y)}$ can be handled independently of one another, whereas under OI-subsumption, the clause not only becomes longer, but also introduces the extra dependency $\mathsf{Y} \neq \mathsf{X}$. Finally, the lgg of two clauses may no longer be unique, as shown in the next example.

Example 5.25. Consider the two clauses

$$\begin{array}{l} \mathsf{pos} \leftarrow \mathsf{circle(a), blue(a), small(a)} \\ \mathsf{pos} \leftarrow \mathsf{circle(b), blue(b), large(a), circle(c), green(c), small(c)} \end{array}$$

The lgg under θ-subsumption is

$$\mathsf{pos} \leftarrow \mathsf{circle(X), blue(X), circle(Y), small(Y)}$$

but this clause does not OI-subsume the first clause. Instead, there are now two minimal general generalizations of the two clauses:

$$\begin{array}{c} \mathsf{pos} \leftarrow \mathsf{circle(X), blue(X)} \\ \mathsf{pos} \leftarrow \mathsf{circle(X), small(X)} \end{array}$$

This situation is akin to the older work on structural matching [Ganascia and Kodratoff, 1986, Hayes-Roth and McDermott, 1978, De Raedt et al., 1997].

5.5.2 Inverse Implication*

The incompleteness of θ-subsumption with respect to logical entailment has been addressed under the framework of inverse implication. Rather than inverting θ-subsumption this framework attempts to invert the implication relation among clauses. A clause c_1 implies a clause c_2 if $c_1 \models c_2$. The implication framework differs from θ-subsumption only for recursive clauses. The computational properties of the framework are worse than those of θ-subsumption because testing whether one Horn clause logically implies another one is only semi-decidable [Marcinkowski and Pacholski, 1992].

Research within inverse implication has focused around the lgg operation. We illustrate this operation on an example.

Example 5.26. Consider

$$\begin{array}{c}\mathsf{nat(s(s(s(a)))} \leftarrow \mathsf{nat(a)} \\ \mathsf{nat(s(s(b))} \leftarrow \mathsf{nat(b)}\end{array}$$

If one requires the *lgg* to be a Horn clause, then the *lgg* is not necessarily unique. The minimally general generalizations in the form of Horn clauses are:

$$\begin{array}{c}\mathsf{nat(s(s(X)))} \leftarrow \mathsf{nat(Y)} \\ \mathsf{nat(s(X))} \leftarrow \mathsf{nat(X)}\end{array}$$

Both of these clauses entail the example clauses. Neither of these generalizations entails the other generalization.

If one lifts the requirement that the *lgg* is a Horn clause, then the following *lgg* is obtained:

$$\mathsf{nat(s(X)); nat(s(s(Y)))} \leftarrow \mathsf{nat(X)}$$

Because of these difficulties, the framework of inverse entailment is seldom used in practice. Furthermore, given the completeness of θ-subsumption for *non-recursive* clauses, it is only relevant for applications requiring recursion, such as program synthesis or grammar induction; cf. Chapter 7.

5.6 Using Background Knowledge

The frameworks for generality introduced so far are concerned with the relationship among *single* clauses. More formally, they define a clause g to be more general than a clause s, notation $g \preceq_o s$, (w.r.t. a partial or quasi-order o) if and only if

$$g \vdash_o s \tag{5.29}$$

where the generality relationship is induced by a deductive inference operator $\vdash_o$; cf. the discussion in Sect. 5.1. Various relationships that take into account single clauses s and g have been studied above, but they do not employ any further knowledge that might be available to the learner. Researchers since Plotkin [1970] have realized the limitations of this approach and have considered generality orders in which background knowledge can be taken into account. In these models, a clause g is more general than a clause s *relative to the background theory* B, notation $g \preceq_{o,B} s$, if and only if

$$g \wedge B \vdash_o s \tag{5.30}$$

This definition complements the semantic notion of generality, which stated that $g \wedge B \models s$; cf. Sect. 5.1.

Example 5.27. Let

$$s = \mathsf{pos(X)} \leftarrow \mathsf{red(X), triangle(X)};$$
$$g = \mathsf{pos(Y)} \leftarrow \mathsf{red(Y), polygon(X)}, \text{ and}$$
$$B = \{\ \mathsf{polygon(X)} \leftarrow \mathsf{triangle(X)}\}.$$

Then g is more general than s (w.r.t. B) because s can be derived from g and B using SLD-resolution.

Again various frameworks have been developed: relative subsumption [Plotkin, 1971], relative implication [Nienhuys-Cheng and de Wolf, 1997] and generalized subsumption [Buntine, 1988]. These differ mainly in the form of inference rule and derivation that is allowed starting from g and B. Rather than focusing on the differences,[3] which are rather technical, we will focus on the similarities among these frameworks. Furthermore, for simplicity, we will assume that all clauses in the background theory and hypotheses space are definite range-restricted clauses (that is, each variable occurring in the head of a clause also occurs in the body of the clause).

Because these frameworks generalize θ-subsumption or implication, many of the problems for these simpler frameworks carry over to the relative notions of generality, such as the existence of syntactic variants, of infinite descending and ascending chains, of the non-existence of ideal refinement operators, etc. Rather than discussing the complex structures imposed on the search space by the various frameworks of relative generalization, we shall in the remainder of this section focus on alternative uses of relative generalization.

There are three concepts that have received quite some attention with regard to relative generalization: 1) the *saturation* of clauses w.r.t. the background theory, 2) the least general generalization operator relative to the background theory, the so-called *relative lgg*, and 3) the design of specialization operators that avoid the generation of redundant clauses. We discuss each of these topics in turn in the next three subsections.

5.6.1 Saturation and Bottom Clauses

The first approach, incorporated in techniques such as saturation or bottom clauses, attempts to find the most specific clause (within the hypothesis space $\mathcal{L}$) that covers an example with regard to the background theory. This clause is then called the *bottom* or *starting* clause.

[3] More specifically, Plotkin's relative subsumption allows the clause g to be used at most once in a deduction whereas relative implication does not impose such a restriction. Thus the differences between relative subsumption and relative implication are akin to those among the subsumption and implication order. Furthermore, Buntine's generalized subsumption requires that g and s have the same predicate symbol in the head and allows only SLD-derivations in which g is used once and in which all other clauses employed in the derivation belong to B. The interested reader is referred to [Nienhuys-Cheng and de Wolf, 1997] for an excellent and extensive overview of these frameworks.

More formally, the *bottom clause* $\bot(c)$ with regard to a clause c and a background theory B is the most specific clause such that

$$B \cup \bot(c) \vdash_o c \tag{5.31}$$

The bottom clause is of interest because it bounds the search for a clause covering the example c as any single clause hypothesis h covering c with regard to B must be more general than $\bot(c)$.

Example 5.28. Let B consist of

$$\begin{array}{l}\mathsf{polygon(X)} \leftarrow \mathsf{rectangle(X)}\\ \mathsf{rectangle(X)} \leftarrow \mathsf{square(X)}\end{array}$$

and let c be as follows:

$$\mathsf{pos(X)} \leftarrow \mathsf{red(X), square(X)}$$

Then

$$\bot\mathsf{(c)} : \mathsf{pos(X)} \leftarrow \mathsf{red(X), square(X), rectangle(X), polygon(X)}$$

Any clause that is not more general than the bottom clause cannot cover the example c and can therefore be safely pruned away, for instance, $\mathsf{pos(X)} \leftarrow \mathsf{green(X)}$. Bottom clauses are thus similar to the S set in version spaces as the (positive) example c determines the set $S = \{\bot(c)\}$. Thus the bottom clause captures all information that is relevant to the example and the background theory. In the example, even though the predicates green and circle might appear in the background theory, they are irrelevant to the specific example, and hence do not appear in the bottom clause. Given the bottom clause with regard to a positive example, inductive logic programming systems only consider the elements in the corresponding version space. This can be realized by searching the space from general to specific starting from $\top$ using a refinement operator that considers only generalizations of the bottom clause $\bot(c)$. Although refinement operators working under relative generality could, in principle, be used, it is more convenient to employ the standard θ-subsumption operators. This is the strategy in the well-known inductive logic programming system PROGOL of Muggleton [1995].

So far we have not given any details as to how bottom clauses are computed. In general this will depend on the notion of relative generality considered. Below, we present the computation in its most general form, that is, for inverse entailment, the semantic notion of relative generality.

We are looking for the most specific clause $\bot(c)$ such that

$$B \cup \bot(c) \models c \tag{5.32}$$

This is logically equivalent to

$$B \cup \neg c \models \neg\bot(c) \tag{5.33}$$

If c is a clause containing variables, then $\neg c$ is a set of facts obtained by applying a skolemization substitution θ. For instance, for the above example, $\theta = \{X \leftarrow \mathsf{sk}\}$ and

$$\neg c\theta = \{\neg \mathsf{pos(sk)}, \mathsf{red(sk)}, \mathsf{square(sk)}\} \tag{5.34}$$

How do we then obtain $\bot(c)$? Well, we first compute $\neg c$, which consists of one false ground fact ($\neg head(c)\theta$) as c is a definite clause, and a set of positive ground facts (corresponding to the atoms in the skolemized $\neg body(c)\theta$). Then we compute the set of all ground facts entailed by $B \cup \neg c$, which is given by the least Herbrand model of B and the skolemized atoms of $\neg body(c)\theta$. These atoms, together with the fact $\neg head(c)\theta$, are the atoms of $\neg body(c\theta)$. The computation of the least Herbrand model can be realized using Algo. 2.2 of Chapter 2. In the example, this corresponds to

$$\neg\bot(c)\theta = \neg c\theta \cup \{\mathsf{rectangle(sk)}, \mathsf{polygon(sk)}\} \tag{5.35}$$

We then compute θ^{-1} (replacing each different skolimization constant by a different variable), apply it to $\neg\bot(c)$, and negate the result to obtain $\bot(c)$:

$$\bot(c) : \mathsf{pos(X)} \leftarrow \mathsf{red(X)}, \mathsf{square(X)}, \mathsf{rectangle(X)}, \mathsf{polygon(X)} \tag{5.36}$$

The different steps in computing the bottom clause are summarized in Algo. 5.4.

Algorithm 5.4 Computing the bottom clause $\bot(c)$

```
Find a skolemization substitution $\theta$ for $c$ (w.r.t. $B$ and $c$)
Compute the least Herbrand model $M$ of $B \cup \neg body(c)\theta$
Deskolemize the clause $head(c\theta) \leftarrow M$ and return the result.
```

Various theoretical properties of bottom clauses under various restrictions have been investigated. One finding is that the resulting bottom clause may be infinite when functors are used, a problem that can be dealt with by imposing a syntactic bias that restricts the form of atoms to be included in $\bot(c)$.

5.6.2 Relative Least General Generalization*

Another question that has received quite some attention concerns the existence and the computation of least general generalizations relative to a background theory. For convenience, we shall study this problem only when c_1 and c_2 have the same predicate in the head, and all clauses in the background theory B are definite clauses and range-restricted. For this case, Buntine has shown that that $rlgg_B(c_1, c_2)$ (with respect to generalized subsumption) can be computed using Algo. 5.5.

Algorithm 5.5 Computing the $rlgg_B(c_1, c_2)$

```
Compute $\bot(c_1)$ with regard to $B$
Compute $\bot(c_2)$ with regard to $B$
Compute and **return** $lgg(\bot(c_1), \bot(c_2))$
```

Let us illustrate this notion using two examples. The first concerns a special case of the $rlgg_B$, already studied by Plotkin, where $B \cup \{c_1, c_2\}$ are all ground facts. The $rlgg_B$ of two such examples is illustrated in the example below and forms the basis of the inductive logic programming system GOLEM [Muggleton and Feng, 1992].

Example 5.29. Let

$$\begin{array}{l} B = \{\mathsf{p(t,a), m(t), f(a), p(j,p), m(j), m(p)}\} \\ c_1 = \mathsf{fa(t,a)} \leftarrow \text{ and } c_2 = \mathsf{fa(j,p)} \leftarrow \end{array}$$

where p stands for parent, f for female, m for male, and fa for father. Then

$$rlgg_B(c_1, c_2) = lgg\begin{pmatrix} \mathsf{fa(t,a)} \leftarrow \mathsf{p(t,a), m(t), f(a), p(j,p), m(j), m(p)} \\ \mathsf{fa(j,p)} \leftarrow \mathsf{p(t,a), m(t), f(a), p(j,p), m(j), m(p)} \end{pmatrix}$$

$$= \left\{ \begin{array}{ll} \mathsf{fa(Vtj, Vap)} \leftarrow & \mathsf{p(t,a), m(t), f(a), p(j,p), m(j), m(p),} \\ & \mathsf{p(Vtj, Vap), m(Vtj), m(Vtp), p(Vjt, Vpa),} \\ & \mathsf{m(Vjt), m(Vjp), m(Vpt), m(Vpj)} \end{array} \right. \tag{5.37}$$

where the variables have been named with the constants they originate from. For instance, Vtj denotes the variable generalizing t in the first clause and j in the second one.

The resulting clause would not be used in this particular form. The reason is clear from the example. All the literals present in the background theory reappear in the *rlgg*. Since the *rlgg* is to be used in conjunction with the background theory these literals are logically redundant and are removed. This process can be called *reduction w.r.t. the background theory.* Afterward, the clause can be further simplified by reducing it under θ-subsumption; cf. Algo. 5.3. This yields

$$\mathsf{fa(Vtj, Vap)} \leftarrow \mathsf{p(Vtj, Vap), m(Vtj)}$$

Together with B, this clause entails the clauses c_1 and c_2.

As another example of the use of relative least generalization, consider the following example, in which the clauses and the background theory are non-factual.

Example 5.30. Let B consist of the following clauses

$$\begin{array}{l} \mathsf{polygon(X)} \leftarrow \mathsf{triangle(X)} \\ \mathsf{polygon(X)} \leftarrow \mathsf{square(X)} \end{array}$$

Then consider the two clauses

$$\begin{array}{l}\mathsf{pos} \leftarrow \mathsf{triangle(X), circle(Y), smaller(X,Y)}\\ \mathsf{pos} \leftarrow \mathsf{square(X), circle(Y), smaller(X,Y)}\end{array}$$

Computing the bottom clauses leads to

$$\begin{array}{l}\mathsf{pos} \leftarrow \mathsf{triangle(X), circle(Y), polygon(X), smaller(X,Y)}\\ \mathsf{pos} \leftarrow \mathsf{square(X), circle(Y), polygon(X), smaller(X,Y)}\end{array}$$

So, the relative least general generalization is

$$\mathsf{pos} \leftarrow \mathsf{circle(Y), polygon(X), smaller(X,Y)}$$

It should be clear that the *rlgg* operation inherits many of the properties of the *lgg* operation, in particular the complexity problems. Therefore, it is often used in conjunction with severe bias restrictions on the syntax and possibly semantics of the clauses to be induced; cf. Sect. 6.6.

5.6.3 Semantic Refinement*

Most inductive logic programming algorithms proceed from general to specific while applying a refinement operator. When working under θ-subsumption, care has to be taken that refinements of clauses are proper, that is, that the refinements are not equivalent to the original clause; cf. Sect. 5.4.3. This problem becomes even more apparent when refining clauses under a background theory or when the predicates are symmetric or transitive, as illustrated in the following example.

Example 5.31. Suppose that we are learning predicates about natural numbers, and that we are given the predicates equal/2 and lessThan/2 in the background theory, with the usual interpretation. Consider now refining the clause:

$$c_1 = \mathsf{positive(X,Y)} \leftarrow \mathsf{lessThan(X,5), equal(X,Y)}$$

Refinements under θ-subsumption of this clause include:

$$\begin{array}{l}c_2 = \mathsf{positive(X,Y)} \leftarrow \mathsf{lessThan(X,5), equal(X,Y), equal(Y,X)}\\ c_3 = \mathsf{positive(X,Y)} \leftarrow \mathsf{lessThan(X,5), equal(X,Y), lessThan(Y,5)}\\ c_4 = \mathsf{positive(X,Y)} \leftarrow \mathsf{lessThan(X,5), equal(X,Y), equal(X,5)}\end{array}$$

The first two refinements are equivalent to the original clause in the light of common knowledge about natural numbers, which could be formalized in a background theory B:

$$\begin{array}{l}\mathsf{equal(X,Y)} \leftarrow \mathsf{equal(Y,X)}\\ \mathsf{equal(X,Y)} \leftarrow \mathsf{equal(X,Z), equal(Z,Y)}\\ \mathsf{lessThan(X,Y)} \leftarrow \mathsf{equal(X,Z), lessThan(Z,Y)}\\ \mathsf{lessThan(X,Y)} \leftarrow \mathsf{lessThan(X,Z), lessThan(Z,Y)}\\ \mathsf{false} \leftarrow \mathsf{lessThan(X,Y), equal(X,Y)}\end{array}$$

The above-specified background theory is unusual in the sense that it merely lists properties of the involved predicates rather than defining the predicates themselves. The properties listed include the symmetry and transitivity of equal, as well as the transitivity and anti-symmetry of lessThan. Nevertheless, it should be clear that $B \wedge c_2 \models c_1$ and $B \wedge c_3 \models c_1$. To see this, the reader might want to resolve one of the clauses of B with c_2 (or c_3) to yield c_1. At the same time, because c_2 and c_3 are refinements of c_1, $B \wedge c_1 \models c_2$ and $B \wedge c_1 \models c_3$. Therefore, c_2 and c_3 are equivalent to c_1 with regard to the background theory B. Furthermore, clause c_4 cannot cover any instance, because the condition part of c_4 always fails due to the contradictory conditions $\mathsf{lessThan}(\mathsf{X}, 5)$ and $\mathsf{equal}(\mathsf{X}, 5)$ according to the last clause of the background theory.

At this point, the question arises as to whether it is possible to avoid such redundant or contradictory clauses in the search process. The answer to this question is positive, and we shall develop a general approach for realizing this. While doing so, we assume, as in the example, that a background theory B in the form of a set of range-restricted Horn clauses is given and a clause c is to be specialized.

Refinements c' of c that satisfy $B \wedge c' \models c$ are redundant, and therefore should not be considered during the search process. When applying a refinement operator under θ-subsumption, this amounts to requiring that c' not θ-subsume $\bot(c)$.

Example 5.32. The bottom clause $\bot(c_1)$ with regard to the background theory of the previous example is

$$\begin{aligned}&\mathsf{positive}(\mathsf{X}, \mathsf{Y}) \leftarrow \\ &\qquad \mathsf{lessThan}(\mathsf{X}, 5), \mathsf{equal}(\mathsf{X}, \mathsf{Y}), \mathsf{equal}(\mathsf{Y}, \mathsf{X}), \mathsf{lessThan}(\mathsf{Y}, 5)\end{aligned}$$

The clauses c_2 and c_3 θ-subsume $\bot(c_1)$, and hence are redundant.

Similarly, if for a refined clause c, $\bot(c)$ includes the predicate false, it is contradictory. For instance, the bottom clause $\bot(c_4)$ contains the literal false in its condition part, and hence c_4 is contradictory.

These examples suggest the following method of dealing with this problem. After computing refinements under θ-subsumption, check whether the refined clauses are contradictory or redundant, and if so, prune them away. For testing this, the concept of a bottom clause can be employed, though the bottom clauses need not be computed explicitly to carry out these tests as optimizations are possible. At the same time, it is advised to employ syntactic restrictions (such as requiring that all clauses in the background theory be range-restricted and that the bottom clause neither introduce new variables nor new terms in the clause). The above discussion also suggests that it may be convenient to employ a special separate background theory, such as that in Ex. 5.31, to specify properties about the predicates in the domain. This type of background theory is special in that it specifies properties or constraints on these predicates rather than defining them. It is typically also much simpler, shorter, and therefore more efficient to use than the typical background theories found in applications of inductive logic programming.[4]

5.7 Aggregation*

Several studies have shown that the ability to use aggregation inside logical and relational learning can be essential for the success of an application, cf. [Perlich and Provost, 2003, Vens et al., 2006, Knobbe et al., 2001, Krogel and Wrobel, 2001]. The question thus arises as to how aggregated literals can be used in the learning process. In this section, we focus on aggregated conditions of the type introduced in Sect. 4.13. Until recently, it was unclear how such aggregated literals could be could be refined. We shall employ the general framework of Vens et al. [2006], who study refinement of literals involving aggregation along three dimensions. We illustrate this using the clause

$$\mathsf{passes(Student)} \leftarrow \mathsf{AVG}\{\mathsf{Mark}|\mathsf{markFor(Student, Course, Mark)}\} \geqslant 10$$

which states that a student passes if the average mark she gets is at least 10. We employ the same notation as in Sect. 4.13. There are three elements in this condition:

1. the aggregation function (here: AVG),
2. the membership interval (here: $[10, \infty[$) used in the condition, and
3. the set used as the argument of the aggregation function (here: the set of Marks for the $\mathsf{Student}$); for convenience, we shall not explicitly distinguish the query from the set of substitutions for which it succeeds.

The observation made by Vens et al. [2006] is that one can refine along three dimensions, which correspond to the three elements indicated above.

Example 5.33. The following two operations will generalize the clause:

- generalize AVG to MAX, or
- generalize 10 to 8.

On the other hand, neither generalizing nor specializing the query

$$\mathsf{markFor(Student, Course, Mark)}$$

is guaranteed to yield generalizations or specializations. The reason, as we shall see soon, is that the aggregate function AVG is neither monotonic nor anti-monotonic. On the other hand, if we would have started from the clause

[4] On a technical note, this way of computing the refinements amounts to working with *semantically free* clauses, whereas the bottom clause is a kind of *semantically closed* clause; cf. [De Raedt and Ramon, 2004]. These notions are the relational equivalents of free and closed item-sets; cf. Exercise 3.8.

$$\mathsf{passes(Student)} \leftarrow \mathsf{MIN}\{\mathsf{Mark}|\mathsf{markFor(Student, Course, Mark)}\} \geqslant 10$$

we could have generalized the clause by refining the query to

$$\mathsf{markFor(Student, Course, Mark), mathematics(Course).}$$

The three dimensions together determine a cube along which one can generalize and specialize. The cube is defined by three partial orders.

Along the aggregate function dimension, we have:

$$\mathsf{MIN} \preceq_a \mathsf{AVG} \preceq_a \mathsf{MAX} \preceq_a \mathsf{SUM}$$

and

$$\mathsf{COUNT} - \mathsf{DISTINCT} \preceq_a \mathsf{COUNT}$$

where the relation $\mathcal{A} \preceq_a \mathcal{B}$ holds if and only if for all sets S: $\mathcal{A}(S) \leqslant \mathcal{B}(S)$. Note that $\mathsf{MAX} \preceq_a \mathsf{SUM}$ only holds for sets of positive numbers. Along the query dimension, we employ the usual θ-subsumption relation, that is, we write $Q_1 \preceq_\theta Q_2$ when Q_1 θ-subsumes Q_2. Finally, for the membership intervals, we employ the usual subset relation, that is, an interval I_1 subsumes an interval I_2 if and only if $I_2 \subseteq I_1$.

Whether a particular refinement is a generalization or a specialization is determined by whether the aggregate condition is monotonic or anti-monotonic. This is defined in a similar way as for quality criteria; cf. Eq. 3.10. More formally, we say that an aggregate condition of the form $\mathcal{A}(Q) \geqslant t$ with aggregate function $\mathcal{A}$, query Q and threshold t is *monotonic* if and only if

$$\forall \text{ queries } G \preceq_\theta S, \forall t : \mathcal{A}(G) \geqslant t \rightarrow \mathcal{A}(S) \geqslant t \tag{5.38}$$

The notion of anti-monotonicity is defined dually. For instance, the condition $\mathsf{MIN}(Q) \geqslant 5$ is monotonic, whereas $\mathsf{MAX}(Q) \geqslant 5$ is anti-monotonic, and $\mathsf{AVG}(Q) \geqslant 5$ is neither monotonic nor anti-monotonic no matter what query Q is used.

Conditions of the form $\mathcal{A}(Q) \geqslant t$ can now be generalized by:

- generalizing $\mathcal{A}$, that is, replacing $\mathcal{A}$ by a function $\mathcal{G}$ such that $\mathcal{A} \preceq_a \mathcal{G}$, yielding $\mathcal{G}(Q) \geqslant t$;
- generalizing the interval, that is, replacing the threshold t by a threshold g such that $g \leqslant t$, yielding $\mathcal{A}(Q) \geqslant g$.

These two operations will produce generalizations regardless of whether the condition is monotonic or anti-monotonic. However, if the condition $\mathcal{A}(Q) \geqslant t$ being refined is monotonic, one can also apply a third operation, which generalizes Q, that is, replaces Q by a query G for which $G \preceq_\theta Q$, yielding $\mathcal{A}(G) \geqslant t$. The monotonicity requirement is only necessary to guarantee that the result of applying the final operation is a generalization. If on the other hand the condition $\mathcal{A}(Q) \geqslant t$ being refined is anti-monotonic, this operation realizes a specialization.

Finally, let us remark that the following dualities hold.

- Applying the inverse of the first two operations realizes specialization for conditions of the form $\mathcal{A}(Q) \geqslant t$.
- Applying the first two operations on conditions of the form $\mathcal{A}(Q) < t$ realizes specialization.
- Applying the inverse of the third operation on a monotonic condition $\mathcal{A}(Q) \geqslant t$ realizes specialization.

Exercise 5.34. Show some generalizations and specializations of the clause

$$\mathsf{fails(Student)} \leftarrow \mathsf{MIN}\{\mathsf{Mark}|\mathsf{markFor(Student, logic, Mark)}\} \leqslant \mathsf{10}$$

along all possible dimensions.

5.8 Inverse Resolution

Perhaps the most ambitious and intuitively appealing framework for generalization is that of inverse resolution. The underlying idea is simple. It is based on the observation that John Alan Robinson's resolution principle [1965] underlies most deductive inference operators used in theorem-proving systems. This observation, together with the general principles concerning the notion of generality $g \models s$, led Stephen Muggleton [1987] to propose the inversion of the resolution principle.

The inverse resolution idea can be formalized as follows. Given are two sets of two clauses g and s. If s follows from g by resolution, notation $g \vdash_{res} s$, then g is more general than s. Resolution is then the operation by which s can be computed from g and inverse resolution should allow the computation of g starting from s.

Example 5.35. Consider the clauses c_1, c_2 and c_3:

c_1: $\mathsf{grandparent(GP, GC)} \leftarrow \mathsf{father(GP, C), parent(C, GC)}$
c_2: $\mathsf{father(F, C)} \leftarrow \mathsf{male(F), parent(F, C)}$
c_3: $\mathsf{grandparent(GP, GC)} \leftarrow \mathsf{male(GP), parent(GP, C), parent(C, GC)}$

For these clauses, $c_1 \wedge c_2 \vdash_{res} c_3$ and, of course, we can also write $c_1 \wedge c_2 \vdash_{res} c_1 \wedge c_3$ and $c_1 \wedge c_2 \vdash_{res} c_2 \wedge c_3$ if we replace one of the clauses by the resolvent. Therefore, $c_1 \wedge c_2$ is more general than $c_1 \wedge c_3$ and $c_2 \wedge c_3$. Whereas specializations such as c_3 can be obtained by resolution, the idea is now to define operators that allow us to infer $c_1 \wedge c_2$ from $c_1 \wedge c_3$ and from $c_2 \wedge c_3$.

Various types of inverse resolution operators have been distinguished. Let us start by considering the resolution principle for propositional clauses. In its general form, we can derive c from $c_1 \wedge c_2$, that is, $c_1 \wedge c_2 \vdash_{res}$ when the clauses are defined as in the following scheme:

$$\frac{c_1 = \ \{l, l_1, \ldots, l_n\} \text{ and } c_2 = \ \{\neg l, l'_1, \ldots, l'_m\}}{c = \ \{l_1, \ldots, l_n, l'_1, \ldots, l'_m\}} \tag{5.39}$$

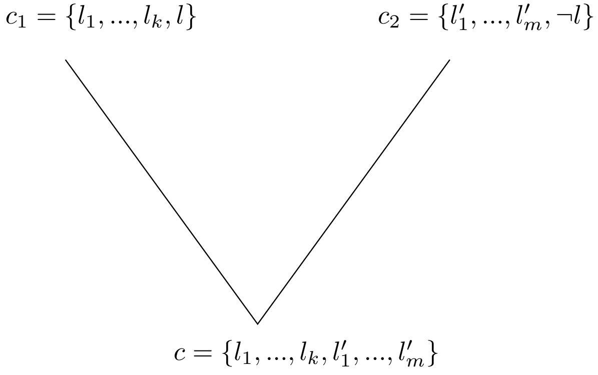

$c_1 = \{l_1, ..., l_k, l\}$
$c_2 = \{l'_1, ..., l'_m, \neg l\}$
$c = \{l_1, ..., l_k, l'_1, ..., l'_m\}$

Fig. 5.10. The V-operators

This application of resolution is graphically illustrated in Fig. 5.10 in the typical V form. The inverse resolution *V-operators* now induce c_1 from c_2 and c, or c_2 from c_1 and c, such that $c_1 \wedge c_2 \vdash_{res} c$. If the involved clauses are Horn clauses, it is convenient to distinguish two different types of V-operator.

The *absorption* operator proceeds according to the following scheme:

$$\frac{p \leftarrow q, k_1, \ldots, k_m \text{ and } q \leftarrow l_1, \ldots, l_n}{p \leftarrow l_1, \ldots, l_n, k_1, \ldots, k_m \text{ and } q \leftarrow l_1, \ldots, l_n} \tag{5.40}$$

In this scheme, the leftmost clause below the line is the resolvent of the two clauses above the line and the rightmost clause below is a copy of the rightmost clause above the line. Thus the specialization operator deductively infers the clauses below the line from those above the line, whereas the absorption operator inductively infers the clauses above the line from those below it.

Similarly, by changing the position of the copied clause, we obtain the *identification* operator:

$$\frac{p \leftarrow q, k_1, \ldots, k_m \text{ and } q \leftarrow l_1, \ldots, l_n}{p \leftarrow l_1, \ldots, l_n, k_1, \ldots, k_m \text{ and } p \leftarrow q, k_1, \ldots, k_m} \tag{5.41}$$

Example 5.36. An illustration of the use of inverse resolution is given in Fig. 5.11. It assumes that the clauses

$$\begin{array}{l}\mathsf{foursided} \leftarrow \mathsf{rectangle} \\ \mathsf{rectangle} \leftarrow \mathsf{square}\end{array}$$

are part of the background theory. Applying resolution, one proceeds from top to bottom to infer the example

$$\mathsf{pos} \leftarrow \mathsf{square}, \mathsf{red}$$

from the hypothesis

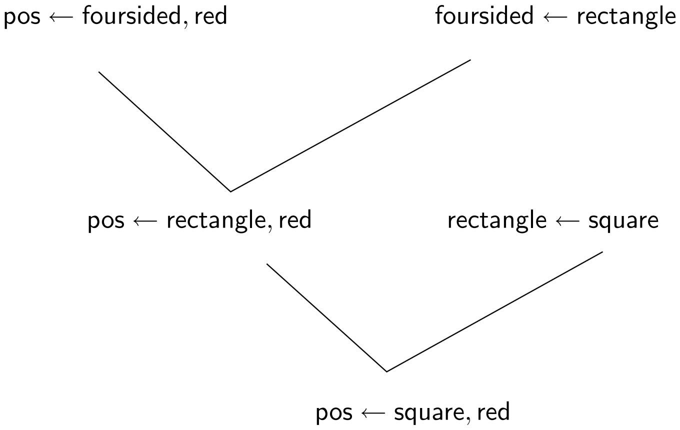


Fig. 5.11. An inverse resolution derivation

pos ← foursided, red

and the background theory. When applying inverse resolution one proceeds from bottom to top to inductively infer the hypothesis from the example and the background theory. This involves two applications of the absorption operator.

One of the difficulties with inverse resolution is that there are various ways to generalize. Consider, for example, the last clause in the previous example and assume that the clause symmetric ← square also belongs to the background theory. Then there are two alternative routes for generalization. In addition to the one shown in Fig. 5.11, one can infer pos ← symmetric, red. The difficulty is now that using the operators as they are defined above, it is impossible to obtain the clause pos ← foursided, red, symmetric given the background knowledge, which complicates systematically searching through the space of possible clauses.

So far, we discussed only propositional inverse resolution. What changes if we consider resolution for first-order logic? Well, the key difference is that the substitutions must be inverted as well. Recall that for first-order resolution two clauses, $c_1 = \{l, l_1, ..., l_n\}$ and $c_2 = \{\neg l', l'_1, ..., l'_m\}$, resolve to form $c = (l_1 \vee ... \vee l_n \vee l'_1 \vee ... \vee l'_m)\theta$ where θ is the most general unifier of l and l', that is, $l\theta = l'\theta$. If we want to compute c_1 from c and c_2 we have to invert θ as well. In general, the inverse substitution θ^{-1} need not be unique, as illustrated in Fig. 5.12 and 5.13.

Exercise 5.37. Can you specify a (non-deterministic) algorithm to compute the inverse resolvent in the absorption operator?

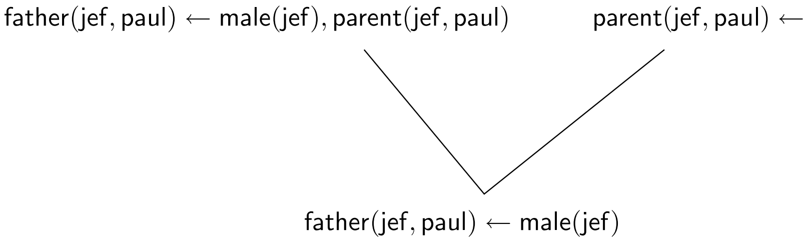

father(jef, paul) ← male(jef), parent(jef, paul)
parent(jef, paul) ←
father(jef, paul) ← male(jef)

Fig. 5.12. Inverse resolution example

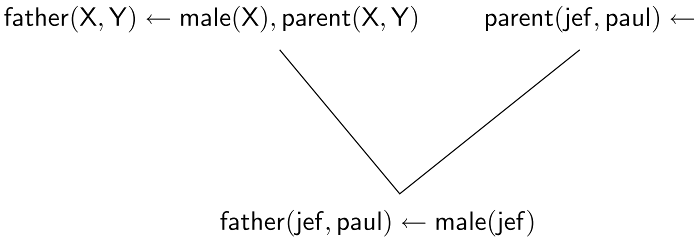

father(X, Y) ← male(X), parent(X, Y)
parent(jef, paul) ←
father(jef, paul) ← male(jef)

Fig. 5.13. Inverse resolution example

The different possible choices to invert the substitution introduce further choice points and non-determinism in the search, which complicates the search strategy further.

There are also two so-called W-operators, which invert two resolution steps in parallel. We define them in the context of Horn clauses. The general scheme for these operators is specified in Figs. 5.14 and 5.15.

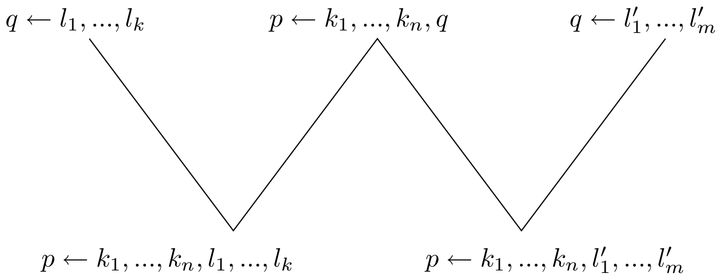

$q \leftarrow l_1, ..., l_k$
$p \leftarrow k_1, ..., k_n, q$
$q \leftarrow l'_1, ..., l'_m$
$p \leftarrow k_1, ..., k_n, l_1, ..., l_k$
$p \leftarrow k_1, ..., k_n, l'_1, ..., l'_m$

Fig. 5.14. The intra-construction operator

The W-operators link two V-operators in such a way that one of the clauses is shared. In Figs. 5.14 and 5.15, the two clauses at the bottom of the figure are derived by resolution; and vice versa, the three clauses at the top of the

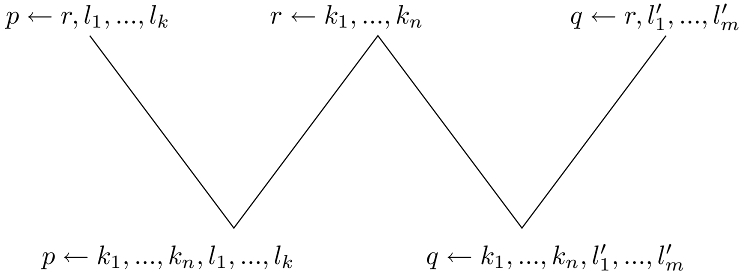

$p \leftarrow r, l_1, ..., l_k$
$r \leftarrow k_1, ..., k_n$
$q \leftarrow r, l'_1, ..., l'_m$
$p \leftarrow k_1, ..., k_n, l_1, ..., l_k$
$q \leftarrow k_1, ..., k_n, l'_1, ..., l'_m$

Fig. 5.15. The inter-construction operator

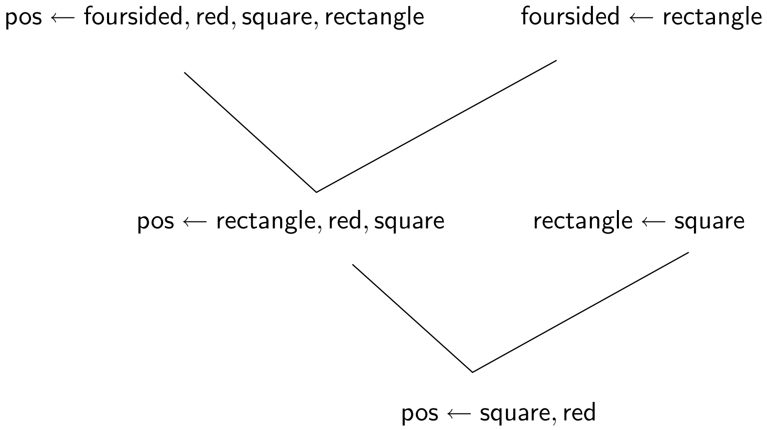

$\mathsf{pos} \leftarrow \mathsf{foursided}, \mathsf{red}, \mathsf{square}, \mathsf{rectangle}$
$\mathsf{foursided} \leftarrow \mathsf{rectangle}$
$\mathsf{pos} \leftarrow \mathsf{rectangle}, \mathsf{red}, \mathsf{square}$
$\mathsf{rectangle} \leftarrow \mathsf{square}$
$\mathsf{pos} \leftarrow \mathsf{square}, \mathsf{red}$

Fig. 5.16. Inverse resolution and bottom clauses

figure are derived by inverse resolution from the two clauses at the bottom. Using the notation for inference operators, we obtain the following schemes for *intra-construction* and *inter-construction*, respectively.

$$\frac{q \leftarrow l_1, \ldots, l_k \text{ and } p \leftarrow k_1, \ldots, k_n, q \text{ and } q \leftarrow l'_1, \ldots, l'_m}{p \leftarrow k_1, \ldots, k_n, l_1, \ldots, l_k \text{ and } p \leftarrow k_1, \ldots, k_n, l'_1, \ldots, l'_m} \tag{5.42}$$

$$\frac{p \leftarrow r, l_1, \ldots, l_k \text{ and } r \leftarrow k_1, \ldots, k_n \text{ and } q \leftarrow r, l'_1, \ldots, l'_m}{p \leftarrow k_1, \ldots, k_n, l_1, \ldots, l_k \text{ and } q \leftarrow k_1, \ldots, k_n, l'_1, \ldots, l'_m} \tag{5.43}$$

Example 5.38. Applying the intra construction operator to the clauses

$$\mathsf{grandparent(X, Y)} \leftarrow \mathsf{father(X, Z), father(Z, Y)}$$
$$\mathsf{grandparent(X, Y)} \leftarrow \mathsf{father(X, Z), mother(Z, Y)}$$

yields the clauses:

$$\begin{aligned}&\mathsf{grandparent(X,Y)} \leftarrow \mathsf{father(X,Z), newp(Z,Y)}\\&\mathsf{newp(Z,Y)} \leftarrow \mathsf{father(Z,Y)}\\&\mathsf{newp(Z,Y)} \leftarrow \mathsf{mother(Z,Y)}\end{aligned}$$

The W-operators are of special interest because they introduce new predicates into the language For the intra-construction operator of Eq. 5.42, the predicate q does not appear below the line, and hence when using this scheme inductively, a new predicate will be introduced. For the inter-construction operator of Eq. 5.43, the predicate r is new. The automatic introduction of new terms or predicates in the description language is called *predicate invention*. Inventing *relevant* new predicates is one of the hardest tasks in inductive logic programming, because there are so many possible ways to introduce such predicates and because it is hard to judge the quality of such predicates. In the few attempts to invent new predicates, compression measures have been employed as a quality criterion. The different inverse resolution operators and predicate invention will be illustrated in Sect. 7.4.2, where the propositional theory revision system DUCE [Muggleton, 1987] will be discussed.

The above-sketched complications with the search process explain why inverse resolution operators are not very popular in practice. Nevertheless, inverse resolution is extremely useful as a general framework for reasoning about generality. It can serve as a general framework in which the other frameworks can be reformulated. Various theoretical results exist in this respect. As an illustration, let us look at the generation of the bottom clause under inverse entailment.

Example 5.39. Reconsider the bottom clause example of Ex. 5.28. Observe that $B \cup \bot(c) \vdash_{res} c$. Therefore we should be able to obtain $\bot(c)$ by inverse resolution from c. This is indeed possible as shown in Fig. 5.12, where a literal is added to the condition part of a clause by applying a single inverse resolution step and using the empty inverse substitution. By repeating this process until no further conditions can be added, the bottom clause would be obtained.

5.9 A Note on Graphs, Trees, and Sequences

In Chapter 4, we provided an almost exhaustive list of different data structures and representations used within symbolic machine learning and data mining. The present chapter has provided an overview of techniques and frameworks for reasoning about the generality of *logical* descriptions only. One important question, that, so far, has not been answered yet, is how these logical frameworks relate to and can be used for simpler data structures such as graphs, trees, and sequences.

The answer to this question is interesting because it shows a key advantage of working with an expressive representation such as logic. Because graphs, trees and sequences can easily be represented using Horn logic, the frameworks for generality introduced in this chapter can be specialized to work with graphs, trees or sequences. This is an example of *downgrading* results for more expressive representations to those that are less expressive. To realize this one must encode the less expressive representation using the more expressive one, and then apply the results of the more general framework.

To illustrate this point, let us investigate how this works for graphs. Recall from Sect. 4.8 that a graph (V, E) consists of a set of nodes or vertices V and a set of edges $E \subseteq V \times V$. Let us also assume that the graphs are directed, that is, that the edges (x, y) and (y, x) are different. Such graphs can be encoded by introducing for each vertex v, an atom $\mathsf{node}(v)$, and for each edge (x, y) an atom $\mathsf{edge}(x, y)$. For instance, the graph (V, E) with $V = \{a, b, c\}$ and $E = \{(a, a), (a, b), (b, c), (c, c)\}$ can be represented by the following Horn clause:

$$\leftarrow \mathsf{node(A), node(B), node(C), edge(A, A), edge(A, B), edge(B, C), edge(C, C)}$$

where we have replaced the identifiers of the vertices with different variables in the logical notation. Notice that this type of clausal representation of graphs imposes some syntactic restrictions on the form of the resulting clause:

- there is an empty conclusion part
- every variable that occurs in the Horn clause must also occur in an atom for the predicate $\mathsf{node/1}$
- all terms are variables.

This is an example of a particular syntactic bias; cf. Sect. 6.6.

The reason for encoding graphs using clausal logic is that the standard frameworks for generality, that is, θ-subsumption and OI-subsumption, can now be used for a variety of purposes. First, they can be used as both coverage and generality relations. Selecting θ-subsumption with the above encoding of graphs corresponds to using subgraph homeomorphism for matching, whereas OI-subsumption results in subgraph isomorphism. Using the latter notion every node in the subgraph must be matched with a unique node in the more specific graph, whereas θ-subsumption allows for one node to match with multiple nodes in the more specific graph; cf. Sect. 9.4.6. Second, the operators for these subsumption relations can be used to specialize and generalize the graph-based representations, though they may need to be slightly adapted to respect the syntactic bias imposed by the encodings of graphs. For instance, applying a refinement operator that adds literals to the condition part of the clause, one obtains encodings of more specific graphs. For instance, adding the literal $\mathsf{edge(A, C)}$ or $\mathsf{node(D)}$ yields a graph that is extended with an edge or a node. These are the two typical elementary operations in graph refinement. Other operations that are relevant include the minimally or least general generalizations, which correspond to finding the maximally common subgraphs; cf. also Sect. 9.4.6.

Finally, let us mention that it is possible, and often easy, to modify this scheme for other types of graphs, trees or sequences. For labeled graphs, the $\mathsf{edge/2}$ predicate needs to be replaced by a $\mathsf{label/3}$ predicate for each of the possible labels. Representing undirected graphs is a bit more tricky because in logic the predicate arguments in an atom are ordered from left to right, and hence directed. To represent undirected graphs we need to employ a symmetric predicate. This is possible using the semantic refinement techniques of Sect. 5.6.3. One then needs to add the clause

$$\mathsf{edge(A, B)} \leftarrow \mathsf{edge(B, A)}$$

to the background theory to specify the symmetry of the edge/2 predicate and then consider generalization relative to this background theory. For trees and sequences, the same encoding as for graphs can be employed, although further syntactic constraints will have to be imposed on the resulting Horn clauses.

5.10 Conclusions

This chapter has discussed the generality relation within logic, and it has been shown that the generality relation coincides with logical entailment. This important property has allowed us to devise inductive inference operators by inverting deductive ones. By imposing different restrictions on the form of the hypotheses and on the allowed deductive inference operators, different frameworks for generality were obtained. The most important frameworks include θ- and OI-subsumption, and various forms of subsumption relative to a background theory and inverse resolution. For these frameworks, the most important operators, that is, inductive inference rules, were derived and their properties studied. Towards the end of the chapter, we discussed several advanced topics, such as semantic refinement, aggregation and the application of these logical frameworks for generality to graphs, trees and sequences.

5.11 Bibliographic Notes

The most complete and detailed account of generalization and specialization in the context of inductive logic programming is given in the book by Nienhuys-Cheng and de Wolf [1997]. It provides a formal account of the various frameworks for and properties of generalization (with the exception of OI-subsumption). Most influential in the theory of inductive logic programming has been Plotkin [1970, 1971], who introduced θ-subsumption, relative subsumption and the notions of least general generalization (with and without background theory). His work, now more than 30 years old, still underlies most of the existing inductive logic programming systems.

Shortly after Plotkin's influential work, Vere [1975] introduced inverse substitutions and Shapiro [1983] introduced refinement operators, whose properties were studied by van der Laag and Nienhuys-Cheng [1994]. Stephen Muggleton has also contributed many essential concepts to the theory of inductive logic programming. This includes the framework of inverse resolution [Muggleton, 1987, Muggleton and Buntine, 1988], inverse entailment [Muggleton, 1995] and inverse implication [Muggleton, 1992a] (see also [Idestam-Almquist, 1993]). The idea of induction as the inverted deduction goes back to [Michalski, 1983] and of inverse resolution to [Wirth, 1988] and [Sammut and Banerji, 1986]. The concept of bottom clauses goes back to Buntine [1987] and was also studied under the names of starting clauses [Adé et al., 1995] and saturation [Rouveirol, 1994].

Many current results were formalized by Akihiro Yamamoto [1999a,b]. OI-subsumption was introduced and studied more recently by Esposito et al. [1996]. Semantic refinement was studied by De Raedt and Ramon [2004], refinement at the theory level by Badea [2001], and refinement involving aggregation by Vens et al. [2006] and Knobbe et al. [2001, 2002].

6

The Upgrading Story

Whereas previous chapters have introduced the foundations of multi-relational data mining and inductive logic programming, this chapter discusses the development of logical and relational learning systems. Rather than providing a detailed and exhaustive overview of the many systems that exist, we present and illustrate a methodology *for developing such systems using three well-known case studies:* FOIL *[Quinlan, 1990],* TILDE *[Blockeel and De Raedt, 1998] and* WARMR *[Dehaspe and Toivonen, 2001]. The methodology is one of the key lessons learned from the development of numerous logical and relational learning systems over the past two decades. The methodology states that in order to develop a novel logical or relational learning system, it is advantageous to start from a well-known effective propositional learner and to upgrade it by extending its representations and operators. This is advantageous because it allows one to maximally profit from the research on propositional learners (and inherit their efficiency and effectiveness), and also because there will be a clear relationship between the new logical or relational learning system and its propositional counterpart. This should not only allow the new system to emulate its propositional counterpart but also make the new system easier to use and understand. From a pragmatic perspective, it is often necessary to impose certain (semantic or syntactic) restrictions on the induced hypotheses or to guide the search using certain types of preferences. Such restrictions and preferences are known as* biases *and are studied in the last section of this chapter.*

6.1 Motivation for a Methodology

When tackling a particular application, the data mining analyst has to decide 1) how to formulate the data mining task, and 2) which data mining algorithm to employ to solve the application. In this process, the need to develop a novel logical or relational learning system may arise because of the limitations of the available (propositional) tools. In the first phase, the analyst must typically identify the mining task from the many available ones, such as classification, regression, probabilistic modeling, clustering, and association rules discovery. At the same time, he must determine a suitable representation, and, for reasons outlined in Chapter 4, the chosen representation may well need to be a relational one. The task and nature of the representation then determine the choice of the data mining algorithm or system to be employed.

Unfortunately, when dealing with relational problems, there may not be a readily available system that is suitable for the problem at hand. In such situations, the analyst may want to consider devising a novel logical or relational learning system. Furthermore, if the mining task has already been identified and an effective propositional learner exists for tackling a propositional version of the problem, it is a good idea to *upgrade* the propositional learner, which means to adapt the existing learner so that it can cope with more expressive representations. This adaptation has to be performed by the machine learning or data mining expert, and typically involves the implementation of a new upgraded system. Upgrading the propositional learner typically consists of the following steps:

- changing the representation of the examples to deal with relations;
- changing the representation of hypotheses to deal with relations;
- adapting the algorithm as little as possible to deal with the upgraded representations; this step often involves the modification of the operators to traverse the search space;
- adding new features, for instance, to deal with background knowledge, where desired;
- implementing and evaluating the system.

Whereas the first few steps are more concerned with the design of the algorithm, the last few are concerned with its implementation. During the upgrading process, one should take care that the propositional system remains a *special case* of its upgrade so that the propositional system can still be *emulated.*

This *upgrading methodology* is an effective means for obtaining novel logical and relational learning systems. Evidence for this claim will be presented by showing in the next sections that three well-known systems, FOIL [Quinlan, 1990], TILDE [Blockeel and De Raedt, 1998] and WARMR [Dehaspe and Toivonen, 2001], were designed using this methodology. Even though not all developers of these systems may have explicitly followed this methodology, the discussion still provides insight and case studies into how to develop novel logical and relational learning tools. At the same time, three popular such systems will be encountered.

There are many reasons why following the methodology is advantageous. First, as argued above, the need to develop an upgrade of a particular propositional learner may occur naturally when tackling a specific application. Second, by upgrading a learner that is already effective for propositional representations, one can benefit from the experiences and results obtained in the propositional setting. In many cases, such as, for instance, decision trees, this implies that one can rely on well-established methods and findings, which are the outcomes of several decades of machine learning research. It will be hard to do better starting from scratch. Third, upgrading an existing learner is also easier than starting from scratch as many of the components (such as heuristics, search strategy, etc.) can be recycled. It is therefore also economical in terms of person power. Fourth, the upgraded system will be able to simulate the propositional one, which provides guarantees that the output hypotheses will perform well on propositional problems even though it is likely that the upgraded system will be computationally more expensive to use than its propositional counterpart. Finally, it may be possible to incorporate new features in the learner by following the methodology. One feature, which is often absent from propositional learners and may be easy to incorporate, is the use of a background theory. Finally, the author wishes to stress that the methodology – like any other methodology – has limitations and does not always apply. Indeed, an obvious limitation occurs when there does not exist a propositional learner for a given task.

6.2 Methodological Issues

The upgrading methodology, described above, mentions a number of different steps, which we will now investigate in more detail.

6.2.1 Representing the Examples

The first step to take is to upgrade the representation of the examples in order to deal with relations. An important decision to be made in this context is concerned with the choice of the learning setting. Should one employ learning from *interpretations* or from *entailment*? Often, the answer to this question is already given in the propositional learner that one starts from. If it involves examples in the form of propositional clauses, it will be convenient to employ learning from entailment; if, on the other hand, examples correspond to boolean variable assignments or interpretations, it will be better to employ interpretations as examples. In some other cases, both choices will be possible and the choice made will be a matter of personal preference.

To illustrate this point, consider the task of basket analysis, where one wants to compute all frequent item-sets in transactional data. This task was already illustrated in Ex. 3.2 and 3.5. Because the standard formulation of frequent item-set mining already employs the learning from interpretations setting, it is appropriate to employ this setting when upgrading frequent item-set mining (cf. also Sect. 6.5) towards relational representations. On the other hand, when considering a rule learning setting, as in the playtennis example of Ex. 4.5, it is more natural to employ clauses or facts to describe the examples.

Another point to stress is that the application under consideration may not require a full upgrade towards the logical or relational representations introduced in Chapter 4. As stated there, there is an important trade-off between expressive power and efficiency. Therefore, depending on the priorities and requirements of a particular application, it may be wise to use one of the intermediate representations discussed in Chapter 4. Nevertheless, for such intermediate representations, a similar choice between learning from entailment and from interpretations may also be necessary.

6.2.2 Representing the Hypotheses

A similar question arises for representing the hypotheses or patterns. How should one upgrade the representation of the hypotheses? If the propositional representation of hypotheses closely corresponds to a concept in logic (such as clause, term, or query) it will be most convenient to employ the corresponding logical representation. Otherwise, a novel relational or logical representation may have to be designed.

To illustrate this point, consider upgrading the representations employed by typical rule learning systems. These systems learn rules of the form

$$\textsf{IF } a_1 = v_1 \textsf{ and...and } a_n = v_n \textsf{ THEN } \mathit{class} = c$$

where the a_i are different attributes and the v_i are values the a_i can take, and the attribute *class* is the class attribute (with value c). These rules directly correspond to clauses of the form

$$\mathit{class} = c \ \leftarrow a_1 = v_1, \ldots, a_n = v_n$$

If only rules for one class are learned, they can directly be employed as the definition of the "predicate" $\mathit{class} = c$. However, most rule learners compute rules for different classes. These rules are then combined in order to make predictions. There are various ways for realizing this. First, the rule learner may derive a so-called *decision list*, which is nothing else than an ordered set of rules. An example is then classified as belonging to the class of the first rule that fires (that is, covers) the example. This can be realized in Prolog by adding the built-in predicate cut ("!") to the end of each rule. The desired effect – that only the first rule whose conditions are satisfied, will fire – is then realized; cf. [Mooney and Califf, 1995] and also Sect. 6.4, where we provide a detailed example. Second, the rules may be unordered but weighted. To classify an example, the weights of the rules of a particular class that cover the example are aggregated (for instance, summed) to compute scores for all of the classes. The example is then classified into the class with the best score. This effect can be realized by incorporating the weights in the clauses and then implementing the scoring method (for example, using a meta-predicate in Prolog).

The illustration points out one of the most important lessons of the upgrading methodology: there is nothing sacred about the standard logical concepts and there may be good reasons for introducing new relational representations that upgrade some procedural propositional hypothesis language. Of course, this should be exercised with care and only when the traditional logical concepts cannot easily capture the propositional representation.

6.2.3 Adapting the Algorithm

A propositional learner that searches the space of possible hypotheses can often be upgraded by only modifying the operators to traverse the search space to cope with the upgraded hypotheses space; and of course, the coverage test will need to be upgraded too. As the vast majority of propositional operators are a special case of θ-subsumption or OI-subsumption, it will often suffice to consider these two frameworks for generalization. The particular choice will then depend on the properties of the operators that the propositional learning system relies on.

For instance, when the search space is searched *completely* and an optimal refinement operator (cf. Chapter 3) is needed, one should consider OI-subsumption. On the other hand, if the propositional system relies on the existence of a least general generalization, θ-subsumption is to be preferred.

Apart from upgrading the operators and the covers test, one should make as few changes to the original algorithm as possible (especially w.r.t. heuristics and search strategy). This will allow one to maximally profit from the knowledge the propositional learner is based on and at the same time avoid having to go through the same tuning that the designers of the original system had to go through. It also guarantees that the upgraded system can emulate the original one.

6.2.4 Adding Features

Finally, it may be desirable to add some new and typical relational learning features. One feature to incorporate that immediately comes to mind and that is worth considering is background knowledge in the problem setting. Indeed, it is often useful to allow the user to specify new predicates and relations in the background theory, which can then be employed to complete the example descriptions, as extensively discussed in Chapter 4. This extension is often easy to realize as one, in principle, only needs to change the coverage test.

Now that we have introduced the methodology, we discuss three case studies in the next few sections.

6.3 Case Study 1: Rule Learning and Foil

As a first case study, we consider the system Foil [Quinlan, 1990], one of the earliest and still most popular relational learning systems. It upgrades propositional rule learners to the relational setting. We now argue that Foil, as a prototypical relational rule learner, could have been developed by following the above sketched methodology.

6.3.1 Foil's Problem Setting

So, we now consider the problem of developing a relational rule learning system. Traditional rule learning systems typically start from an attribute-value representation as in the playtennis example in Sect. 4.2 and aim at finding rules that accurately predict the class of the examples. In Sect. 4.2, we have already discussed several possibilities for representing such attribute-value learning tasks in a relational setting. Because the target is to derive IF-THEN rules that predict the class, it is natural to employ the learning from entailment setting. Furthermore, let us assume that we work with two classes of examples: positive and negative. Then, the representation employed in Ex. 4.6 motivates the following choices:

- examples are ground facts; positive examples are true, and negative ones are false;
- a hypothesis corresponds to a set of definite clauses, and a definite clause represents a rule.

In addition, given the limited form of examples employed, it is useful to employ a background theory that contains further predicates and information about the examples. This background theory can again be represented as a set of definite clauses.

This leads to the following problem specification, which forms the basis of the inductive logic programming system Foil [Quinlan, 1990] and many of its variants (such as mFoil [Lavrač and Džeroski, 1994]):

Given

- a set of true ground facts P, the set of positive examples,
- a set of false ground facts N, the set of negative examples,
- a background theory B, consisting of a set of definite clauses

Find: a set of definite clauses H such that $\forall p \in P : B \cup H \models p$ and $\forall n \in N : B \cup H \not\models n$.

One illustration of this setting was given in Ex. 4.6 where the background theory B was empty. Nevertheless, we provide another illustration.

Example 6.1. Assume that the background theory consists of the relations subscription, course, company and person in the summerschool database of Sect. 4.4 (cf. Figs. 4.1 and 4.2). If the positive examples are

$$\mathsf{attendsParty(blake)} \leftarrow$$
$$\mathsf{attendsParty(miller)} \leftarrow$$

and the negative one is

$$\mathsf{attendsParty(adams)} \leftarrow$$

then the single clause

$$\mathsf{attendsParty(P)} \leftarrow \mathsf{person(P, J, C), company(C, commercial)}$$

is a solution as it covers the two positive examples and does not cover the negative example.

It should be mentioned that the setting originally introduced by Quinlan in FOIL is more restricted than the one specified above:

- it restricts the background knowledge to contain ground unit clauses (that is, ground facts) only, yielding an *extensional background theory*, and
- it does not allow for functors

At the same time, rather using an *intensional* coverage test, which would merely test whether a hypothesis h covers an example e by checking whether $B \wedge h \models e$, it employs an *extensional* coverage test. Deciding whether a hypothesis h *extensionally* covers an example e with regard to an extensional background theory B involves identifying a clause $c \leftarrow b_1, ..., b_n$ that belongs to the hypothesis h such that there exists a substitution θ for which $c\theta = e$ and the $b_i\theta \subseteq B$.

Example 6.2. Applying the extensional coverage test to attendsParty(blake) and the clause mentioned in the previous illustration amounts to verifying whether there exists a substitution θ such that

$$\{\mathsf{person(blake, J, C), company(blake, commercial)}\}\theta \subseteq B$$

The effect of the choices made in FOIL is that a simple setting (avoiding many complications such as functors) is obtained, which allows for a very fast implementation of some essential components of the learning algorithm. FOIL is still one of the most efficient relational learners that exists today, despite the facts that it was developed around 1990. On the other hand, the use of an extensional background theory and coverage testing is more restricted than the full learning from entailment setting.

Because these restrictions are not essential to the problem setting itself, and are not imposed by some of FOIL's variants, we will largely ignore them throughout the remainder of the section. A more detailed discussion of extensional coverage testing and its efficiency is contained in Sect. 10.1.

Exercise 6.3. Discuss why the extensional coverage test can be implemented efficiently and also sketch situations in which the extensional coverage test causes problems (that is, where extensional and intensional coverage testing may produce different results).

6.3.2 FOIL's Algorithm

Now that the problem has been specified, it is time to design an algorithm for solving it. Due to the closeness of the propositional problem setting to rule learners such as CN2 [Clark and Niblett, 1989, Clark and Boswell, 1991] and AQ [Michalski, 1983], it is convenient to employ a *separate-and-conquer* algorithm [Fürnkranz, 1999]. At the heart of CN2 and AQ is the famous *covering* algorithm sketched in Algo. 6.1. This algorithm repeatedly finds a single rule that is considered *best* (that is, maximizes the number of positive examples covered while keeping the number of negative examples covered as low as possible). The *best* rule is then added to the hypothesis H and all examples of P that are covered by the rule are removed from P. This process terminates when it is impossible to find a good rule or when all positive examples have been covered. Good rules compress the examples according to the minimum description length principle, which states that the encoding of the examples using the rules should be shorter than the original encoding of the examples. This criterion is used to avoid over-fitting; cf. below.

Algorithm 6.1 The covering algorithm

```
$H := \emptyset$
repeat
  $b :=$ call bestrule($P$,$N$)
  if $b$ is a rule then
    remove from $P$ all examples covered by $b$
    add $b$ to $H$
  end if
until $b =$ "not found" or $P = \emptyset$
output $H$
```

To find the best rule, systems such as FOIL, CN2 and AQ perform a heuristic general-to-specific beam search procedure along the lines of Algo. 3.4 sketched in Chapter 3. One instantiation (using beam search with beam-size equal to 1, that is, hill climbing) describing FOIL is shown in Algo. 6.2.

To find the best rule, FOIL searches through the space of rules. The structure of this search space is given by the θ-subsumption lattice, and a specialization operator ρ is employed. Due to the non-existence of an ideal specialization operator under θ-subsumption, FOIL (like most other relational learners) employs a pragmatic operator, which systematically adds literals to a given clause. The search process starts at the most general clause $\top$, which for any given predicate p/n corresponds to the fact $p(X_1, ..., X_n)$, where the X_i are different variables. The hill climbing Algo. 6.2 then repeatedly replaces the current clause by the one that scores best among its refinements. This process continues until either no negative example is covered, or the current clause no longer compresses the examples.

Algorithm 6.2 The function **bestrule(P,N)**

```
$c := \top$;
while $c$ covers examples from $N$ and $c$ compresses the examples do
  $c$ := the best refinement in $\rho(c)$
end while
if $c$ compresses the examples then
  return $c$
else
  return "not found"
end if
```

To decide which refinement c' of a clause c scores best, FOIL uses a variant of the *weighted information gain heuristic*:

$$wig(c,c') = \frac{n(c',P)}{n(c,P)} \times \left(log_2\frac{n(c',P)}{n(c',P\cup N)} - log_2\frac{n(c,P)}{n(c,P\cup N)}\right) \qquad (6.1)$$

where $n(c,E)$ denotes the number of examples in E that c covers.

The term $log_2\frac{n(c,P)}{n(c,P\cup N)}$ is an estimate of the amount of information needed to specify that an example covered by the clause is positive. The difference between the two terms is the information gain. The information gain is then weighted by the fraction of positive examples that remain covered after specialization. The weighted information gain balances information gain with coverage.

Weighted information gain is a variant of the heuristic employed by the earlier CN2 system, which FOIL generalizes. It is also a simplification of the one actually employed in FOIL as the FOIL heuristic would rather count the number of *answer substitutions* than the number of covered examples.

A substitution θ is *an answer substitution* for an example e by a clause $h \leftarrow b_1,...,b_m$ if and only if θ grounds the clause, $h\theta = e$ and the query $\leftarrow b_1\theta,...,b_m\theta$ succeeds in the database.

Usually, it is not a good idea to count substitutions instead of examples because in some applications the number of substitutions for different examples can vary a lot, leading to potentially improper balancing of the different examples. For instance, in the musk application [Dietterich et al., 1997], a multi-instance learning problem, there is one example having more than 6,000 instances, whereas most examples only have a couple of instances. When substitutions are counted instead of examples, this particular example becomes almost as important as all other examples taken together!

Example 6.4. Reconsider Ex. 6.1 and consider the following sequence of refinements:

$$\mathsf{attendsparty(P)} \leftarrow$$
$$\mathsf{attendsparty(P)} \leftarrow \mathsf{person(P,J,C)}$$
$$\mathsf{attendsparty(P)} \leftarrow \mathsf{person(P,J,C),company(C,commercial)}$$

The weighted information gain from the first clause to the second one is

$$\frac{2}{3} \times \left(log_2 \frac{2}{3} - log_2 \frac{2}{3} \right) = 0 \text{ bits}$$

and from the second one to the third is:

$$\frac{2}{2} \times \left(log_2 \frac{2}{2} - log_2 \frac{2}{3} \right) = 0.58 \text{ bits}$$

Exercise 6.5. Provide a concrete illustration that shows that counting substitutions may yield different results than counting examples.

FOIL also employs a criterion, based on the minimal description length principle, to decide when to stop refining a rule. According to this criterion, FOIL will stop refining the current hypothesis, when the encoding length of the clause will become longer than the encoding length of the covered examples.

It should be stressed that other heuristics than those presented here could be (and actually have been) employed in relational learners such as FOIL. As in FOIL, two types of heuristic can be distinguished: one that guides the search towards the more promising clauses, for instance, information gain, and one that is meant to cope with noisy data and to avoid over-fitting, for instance, minimum description length. More details on the particular heuristics employed in FOIL can be found in [Quinlan, 1990] and an overview of alternatives can be found in [Lavrač and Džeroski, 1994].

Rather than discussing the use of different heuristics, which are quite similar to those in the propositional setting, we now focus on some new problems that arise in a relational context.

First, as extensively discussed in Chapter 5, there are some serious problems with specialization operators under θ-subsumption, in particular, the non-existence of ideal operators, which implies that one has to resort to the use of a pragmatic operator. Such pragmatic operators proceed by adding literals to clauses. As a consequence, care must be taken that the generated clauses are proper specializations and do not belong to the same equivalence class. For example, the sequence of clauses

$$\begin{array}{l} \mathsf{p(X)} \leftarrow \mathsf{m(X, Y_1)} \\ \mathsf{p(X)} \leftarrow \mathsf{m(X, Y_1), m(X, Y_2)} \\ \mathsf{p(X)} \leftarrow \mathsf{m(X, Y_1), m(X, Y_2), m(X, Y_3)} \\ \ldots \end{array}$$

can be generated by a pragmatic operator, even though all clauses in this sequence are equivalent under θ-subsumption. This problem can be alleviated by resorting to another type of subsumption (such as OI-subsumption); see Chapter 5 for more details.

Secondly, and somewhat related, there is the problem of *determinate* literals. These are literals that when added to a clause, change neither the covered examples nor the covered substitutions. More formally, a literal l is *determinate* with regard to a clause $h \leftarrow b_1, ..., b_n$ and background theory B if and only if for all substitutions θ that ground the clause such that the query $\leftarrow b_1\theta, ..., b_n\theta$ succeeds in B there is exactly *one* substitution σ such that $(h \leftarrow b_1, ..., b_n, l)\theta\sigma$ is ground and the query $\leftarrow b_1\theta\sigma, ..., b_n\theta\sigma, l\theta\sigma$ succeeds in B.

Example 6.6. Continuing the attendsParty illustration, the literal person(P, C, J) is determinate with regard to the clauses

$$\begin{array}{l}\mathsf{attendsParty(P)} \leftarrow \\ \mathsf{attendsParty(P)} \leftarrow \mathsf{subscribes(P, erm)}\end{array}$$

but the literal person(P, president, J) is not.

Determinate literals cause problems for heuristic methods because they do not result in any improvement of the score; cf. Ex. 6.4. Yet, they are often essential because they introduce new variables in the clauses on which further conditions can be specified (such as the job type and the company name when learning attendsParty). Therefore, determinate literals often receive a special treatment. In the literature on inductive logic programming, the following solutions have been considered:

- systematically adding all determinate literals to the current clause, and pruning them away when the clause has been completed (cf. [Quinlan, 1993a]); and
- considering macro-operators, or looking ahead during the search; these solutions consider adding multiple literals during one refinement step, and are especially useful if the first literal added is determinate [Blockeel and De Raedt, 1997]. Related to the use of macro-operators is the use of a resolution step that replaces one literal by a set of literals [Bergadano and Giordana, 1990].

The complex structure on the search space imposed by the subsumption lattice (or other generality relations) is the cause of many problems in relational learning and inductive logic programming. These problems occur not only in a rule learning setting but are quite typical for relational learning in general. One approach to simplifying the search space employs the notion of a *language* or *syntactic bias.* The language bias restricts the search space by requiring that all clauses (or hypotheses) satisfy certain syntactic or semantic restrictions. Because language bias is an important concept for relational learning in general, we devote Sect. 6.6 to it.

Many extensions and variants of the basic Foil algorithm and systems have been developed and are described in the literature. An overview of some of the most important developments is provided at the end of this chapter.

Exercise 6.7. Discuss how to upgrade the cautious specific-to-general Algo. 3.5 to work with examples in the form of definite clauses for a particular predicate. (The solution to this exercise is the basis for the inductive logic programming system Golem [Muggleton and Feng, 1992].)

6.4 Case Study 2: Decision Tree Learning and Tilde

As a second case, we study one of the best known family of learning algorithms, known under the name *top-down induction of decision trees*, and argue that systems such as Tilde [Blockeel and De Raedt, 1998], which induces logical decision trees from examples, have been derived according to the methodology.

6.4.1 The Problem Setting

An algorithm for top-down induction of a decision tree addresses either a classification task, where the target attribute to be predicted is discrete, or, a regression task, where the target attribute is continuous. Decision trees owe their popularity to their simplicity as well as the efficiency with which they can be induced. The efficiency is due to the *divide-and-conquer* algorithm that efficiently partitions the space of examples into coherent regions with regard to the target attribute.

One example decision tree, for predicting the class attribute party in the relation participant (in Table 6.1 below), is shown in Fig. 6.1. The internal nodes of a decision tree contain boolean tests and the leaves contain the value for the class attribute that is predicted (for the party attribute in the example).[1] To classify an example, one starts at the leaf and performs the test on the example. If the outcome of the test is positive or true, the left branch is followed; if it is false, the right one is taken. The resulting node either is a leaf node containing the predicted class value or an internal node, in which case the procedure is recursively applied. For instance, the example adams is classified in the rightmost leaf yielding the class Party = no. Observe also that there is a close correspondence between a decision tree and a set of rules. Indeed, the tree in Fig. 6.1 can be represented using the following set of rules:

IF Seniority = senior THEN Party = yes
IF Seniority ≠ senior and Company = jvt THEN Party = yes
IF Seniority ≠ senior and Company ≠ jvt THEN Party = no

Alternatively, when employing boolean tests, it may be more convenient to employ a decision list:

IF Seniority = senior THEN Party = yes
ELSIF Company = jvt THEN Party = yes
ELSE Party = no

Given the popularity and effectiveness of this class of algorithms, one may wonder how to upgrade to a relational learning setting. To focus our attention on a relational learning problem, let us reconsider the Bongard problems, one of which is illustrated in Fig. 6.2.

[1] Traditional decision trees also allow for non-boolean tests with multiple outcomes. For simplicity, we will not deal with such tests explicitly in this book, though this is, of course, possible.

Table 6.1. The participant relation

Name	Job	Seniority	Company	Party
adams	researcher	junior	scuf	no
blake	president	junior	jvt	yes
king	manager	junior	pharmadm	no
miller	manager	senior	jvt	yes
scott	researcher	senior	scuf	yes
turner	researcher	junior	pharmadm	no

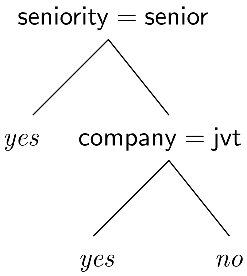


Fig. 6.1. A decision tree for the party attribute (adapted from [De Raedt et al., 2001])

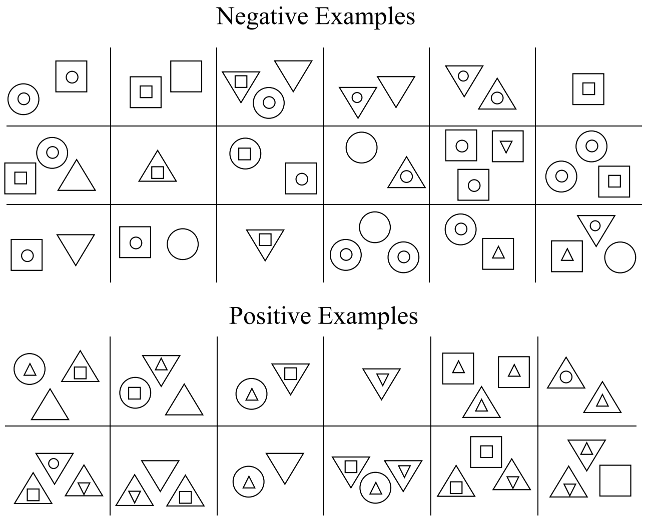


Fig. 6.2. A larger Bongard problem. Reprinted with permission from [De Raedt et al., 2001]

Two immediate questions arise.

- How should the examples be represented?
- What is a relational decision tree?

There are several possible answers to the first question. On the one hand, for the Bongard problems, it is quite natural to employ *interpretations* that completely describe a particular scene. The interpretations are then also labeled with the class value. For instance, the rightmost top example can be represented as

$$\{\mathsf{square(s1)}, \mathsf{square(s2)}, \mathsf{in(s1, s2)}\}$$

Another possibility is to employ one relational database to model the examples as sketched in Ex. 4.27. In both cases, a background theory can be used to complete the interpretations (as discussed in Chapter 4).

The next design decision to make concerns the definition of a relational decision tree. To design a proper definition in relational logic, it is helpful to carefully analyze the propositional decision tree, and to draw some correspondences. Formulated in logic, a test corresponds to an atom or a query. So, what happens if we replace the nodes by queries? To classify an example, one runs the query on the example and takes the corresponding branch in the tree. This seems like a reasonable choice although there is one complication related to the use of variables. If multiple nodes refer to the same logical variables, should they be considered as the same variable or not? For instance, in the example relational decision tree shown in Fig. 6.3, when querying in(T1, T2) should T1 be restricted to triangles because the root query requires T1 to be a triangle?

Because classifying an example in a relational setting often requires one to employ long chains of literals connected through variables, it seems best to define the semantics of relational decision trees so that multiple occurrences of a variable, along a succeeding branch of the decision tree, denote the same variable. Using this view, an equivalent representation of the relational decision tree in Fig. 6.3 reads as follows:

IF triangle(T1), in(T1, T2), triangle(T2) THEN Class = yes
ELSIF triangle(T1), in(T1, T2) THEN Class = no
ELSIF triangle(T1) THEN Class = no
ELSIF circle(C) THEN Class = no
ELSE Class = yes

Thus, the decision tree corresponds to a decision list, where the leaves of the decision tree are traversed from left to right and where each leaf contributes one rule to the list. Furthermore, whereas the variable bindings and atoms are propagated along the succeeding branches of the decision tree, this is not done for the failing branches. For example, the literals (and bindings) for the leftmost leaf produces the condition triangle(T1), in(T1, T2), triangle(T2) but for the left leaf under circle(C), the parent literal triangle(T1) does not occur in the condition of the rule for this leaf because it is already known to have failed (if triangle(T1) would have succeeded, one of the first three rules must have succeeded as well).

Let us also mention that this particular use of an if-then-else construct would be modeled using Prolog's cut ("!"). So, the above decision list or logical decision tree can be represented by the following Prolog program. The effect of "!" in this program is that the first rule for class that succeeds determines the class.

```
class(yes) ← triangle(T1), in(T1, T2), triangle(T2), !.
class(no) ← triangle(T1), in(T1, T2), !.
class(no) ← triangle(T1), !.
class(no) ← circle(C), !.
class(yes) ←
```

Using this representation, it is easy to check how an example e is classified by a logical decision tree t. One merely needs to compute the Prolog program pt corresponding to the tree t, assert both pt and e in a Prolog database and pose the query ← class(C). If a background theory B is employed, B should be asserted in the database as well before posing the query.

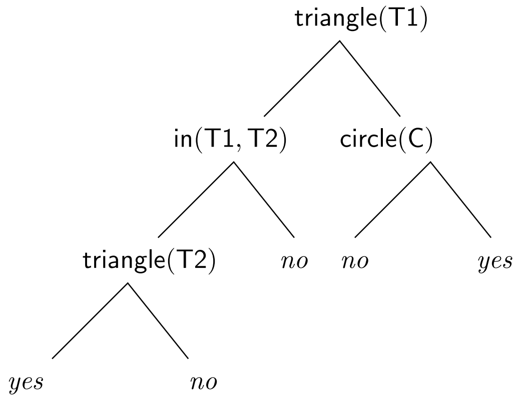


Fig. 6.3. A logical decision tree for a Bongard problem

By now, we are able to formally define the problem setting tackled by a logical decision tree induction system such as Tilde [Blockeel and De Raedt, 1998]:

Given

- a set of labeled examples E in the form of interpretations,
- (possibly) a background theory B in the form of a definite clause theory, which can be used to complete the examples,

Find: a logical decision tree that correctly classifies the examples in E.

6.4.2 Inducing Logical Decision Trees

The logical decision tree learning algorithm is summarized in Algo. 6.3. It starts with the tree containing a single node, all examples and the empty query, and then recursively completes the nodes in the tree. To complete a node, the algorithm first tests whether the example set in the node is sufficiently homogeneous. If it is, it is turned into a leaf; if it is not, all possible tests for the node are computed and scored using a heuristic. The best test is then selected and incorporated into the node and two new nodes are added to the tree: the left one contains those examples for which the test succeeds, the right one those for which the test fails. The procedure is then called recursively for the two sub-nodes.

The only point where the algorithm differs from traditional decision tree learners is in the generation of the tests to be incorporated in the nodes. To this aim, the algorithm employs a refinement operator ρ that works under θ-subsumption. For convenience, we have assumed that the refinement operator specializes a query Q (a set of literals) by adding literals l to the query yielding Q, l. For example, for the query $\leftarrow \mathsf{in(X,Y)}$, the refinement operator may compute refinements such as

$$\leftarrow \mathsf{in(X,Y), triangle(X)}$$
$$\leftarrow \mathsf{in(X,Y), circle(X)}$$

Related to the use of the refinement operator is the need to propagate the query along the succeeding branches of the decision tree. This propagation is necessary to realize the variable bindings among the different tests. Let us illustrate this using the decision tree in Fig. 6.3. If one does not propagate the query $\leftarrow \mathsf{triangle(T1)}$ along the leftmost branch, the node for $\leftarrow \mathsf{triangle(T2)}$ would contain the examples for which the query $\leftarrow \mathsf{triangle(T1), in(T,T2)}$ succeeds rather than the intended $\leftarrow \mathsf{triangle(T1), in(T1,T2)}$.

The only point that we have not addressed yet concerns the heuristic functions used to determine the best tests and to decide when to turn nodes into leaves. At this point, it is important to realize that these problems are essentially the same as for traditional decision tree learners, and that, therefore, the same solutions apply. The first-order decision tree learner TILDE [Blockeel and De Raedt, 1998] employs exactly the same heuristics as the famous decision tree learner C4.5 [Quinlan, 1993a]. A popular heuristic is based on *information gain*, which tries to measure the amount of information that is gained by performing a particular test. The entropy (or information) $I(P, N)$ needed to classify an example in one of two classes P and N (with $E = P \cup N$) is defined as

$$I(P, N) = -\frac{n(P)}{n(E)} \times log_2 \frac{n(P)}{n(E)} - \frac{n(N)}{n(E)} \times log_2 \frac{n(N)}{n(E)} \tag{6.2}$$

Algorithm 6.3 Inducing decision trees **dt**(T: tree, E: examples, Q: query)

```
if $E$ is sufficiently homogeneous then
  $T$:= leaf($C$) where $C$ is the majority class in $E$
else
  for all queries $(Q, l) \in \rho(Q)$ do
    compute $score(Q, l, E)$
  end for
  let $(Q, l)$ be the best refinement with regard to score
  $T.test := l$
  $E_l := \{e \mid (Q, l)$ succeeds in $B \cup e\}$
  $E_r := E - E_l$
  call dt($T.left$,$E_l$,$(Q, l)$)
  call dt($T.right$,$E_r$,$Q$)
end if
```

where P and N are the sets of positive and negative examples, respectively, and $n(X)$ denotes the number of examples in the set X. Furthermore, if we split the sets of examples $E = P \cup N$ into the sets $E_l = P_l \cup N_l$ and $E_r = P_r \cup N_r$ with regard to the test t then the information gain can be expressed as

$$IG(E, E_l, E_r) = I(P, N) - \frac{n(E_l)}{n(E)} \times I(P_l, N_l) - \frac{n(E_r)}{n(E)} \times I(P_r, N_r) \quad (6.3)$$

This expression measures how much information is gained by performing the test t. Decision tree learning algorithms then select the test that results in the maximal information gain.

Example 6.8. Assume that E contains nine positive and five negative examples. Then

$$I(P, N) = -\frac{9}{14} \times log_2 \frac{9}{14} - \frac{5}{14} log_2 \frac{5}{14} = .940 \ bits$$

Assume furthermore that there is a test t that splits the examples into E_l with three positive and four negative examples, and into E_r with six positive and one negative examples, then

$$IG(E, E_l, E_r) = .940 - \frac{7}{14} \times .985 - \frac{7}{14} \times .592 = .151 \ bits$$

These calculations inform us that if we know the outcome of the test t then we have gained .151 bits of information.

Typical decision tree learners also include heuristics to decide when to stop refining a certain node. A simple heuristic that can still be very effective is to stop expanding nodes when the number of examples in the nodes falls below a certain (user-defined) threshold.

Further heuristics and optimizations may be targeted at avoiding overfitting the data and dealing with noise. This includes algorithms for post-pruning decision trees and turning them into rules. There exist also decision tree learners that predict real valued attributes rather than discrete classes. They are known under the name of regression trees. A full discussion of these techniques is outside the scope of this book but can be found in [Breiman et al., 1984, Quinlan, 1993a, 1986, Mitchell, 1997].

6.5 Case Study 3: Frequent Item-Set Mining and Warmr

Since the seminal paper by Agrawal et al. [1993] on discovering association rules, the data mining community has devoted a lot of attention to the local pattern mining paradigm, and a vast number of different approaches and techniques have been developed to efficiently discover such patterns in a variety of different data sets. The large majority of the early approaches were, however, initially limited to flat representations. A natural question that arose in this context was whether the local pattern mining framework could be upgraded for use in relational databases. Below we show how this has been realized using the methodology sketched earlier for logical and relational learning. At this point, the reader may want to revisit the problem of frequent item-set mining and association rule mining in boolean or transactional data introduced in Ex. 3.3. Recall that the goal was to find all item-sets that frequently occur in the transactional data set.

6.5.1 Relational Association Rules and Local Patterns

According to the upgrading methodology, the first two questions to address are concerned with the logical representations of the examples and the patterns. Applied to frequent item-set mining, we are looking for the relational equivalent of item-sets used as transactions or patterns. As transactions in the form of item-sets are boolean interpretations, a natural choice is to use (Herbrand) interpretations as transactions. At the same time, patterns in the form of item-sets can be considered conjunctive queries, and therefore we can use logical queries as local patterns.

Example 6.9. Reconsider the basket analysis problem of Ex. 3.3 and the transaction $\{\mathsf{s}, \mathsf{m}, \mathsf{b}, \mathsf{c}\}$. A local pattern containing the item-set $\{\mathsf{m}, \mathsf{b}\}$ could be represented by the query $\leftarrow \mathsf{m}, \mathsf{b}$. This query covers the transaction, because the query succeeds in the database containing this single transaction.

Whereas this choice of representation is theoretically appealing, is elegant and has been used for local pattern mining, it is not so practical. The reason is that relational databases are usually not partitioned into interpretations and partitioning large relational databases may well be computationally expensive.

Therefore, there is also an alternative formulation of relational frequent pattern mining that circumvents the partitioning process. It is this formulation that will be used throughout this section.

To focus our attention, we shall use the summerschool database of Fig. 4.1 as a typical relational database. When mining for local patterns in the summerschool database, one must first determine the entity of interest. Are we looking for patterns about participants, job types, companies, courses, or a combination of these entities? The entity of interest determines what is being counted, and will determine the type of pattern searched for. As patterns can be represented as definite clauses, the entity of interest also determines the type of clause searched for.

Example 6.10. The following clauses serve as patterns about different entities of interest.

$$\begin{array}{l}
\mathsf{key(P)} \leftarrow \mathsf{participant(P,J,C,Pa),company(C,commercial)}\\
\mathsf{key(J)} \leftarrow \mathsf{participant(P,J,C,Pa),company(C,commercial)}\\
\mathsf{key(C)} \leftarrow \mathsf{participant(P,J,C,Pa),company(C,commercial)}\\
\mathsf{key(Co)} \leftarrow \mathsf{course(Co,L,T),subscription(adams,Co)}\\
\mathsf{key(C,J)} \leftarrow \mathsf{participant(P,J,C,Pa),company(C,commercial)}
\end{array}$$

The first three patterns make statements about participants, job types, and companies, respectively. In the fourth query the entity of interest is the course, and in the last query, it is the relationship between companies and job types. The first clause is a statement about participants. It covers all participants who work for a commercial company, that is, blake and miller. Thus the frequency of the first clause is 2.

We can now formalize the frequent query mining task in relational databases as follows:

Given

- a relational database D
- the entity of interest determining the *key*
- a frequency threshold t
- a language $\mathcal{L}$ of logical clauses of the form $key \leftarrow b_1, ..., b_n$ defining *key*.

Find: all clauses $c \in \mathcal{L}$ for which $freq(c, D) \geqslant t$, where

$$freq(c, D) = |\{\theta \mid D \cup c \models key\theta\}| \tag{6.4}$$

Notice that the substitutions θ only substitute variables that appear in the conclusion part of the clause. For instance, the frequency of the pattern about companies is 1 as jvt is the only commercial company.

In addition to frequent item-sets, the data mining community also employs association rules. Simple association rules in the boolean case are rules of the form

$$\mathsf{IF}\ itemset\ \mathsf{THEN}\ item$$

An example rule is IF m and b THEN s. It denotes that transactions involving m(ustard) and b(eer) typically also contain s(ausage). Traditionally, two measures are associated with these rules:

$$support(\mathsf{IF}\ set\ \mathsf{THEN}\ i,\ D) = \frac{freq(set \cup \{i\}, D)}{freq(\{\}, D)} \tag{6.5}$$

$$confidence(\mathsf{IF}\ set\ \mathsf{THEN}\ i,\ D) = \frac{freq(set \cup \{i\}, D)}{freq(set, D)} \tag{6.6}$$

Association rule mining is then concerned with finding all association rules that have a minimum support and a minimum confidence in a given database. As we will see soon, frequent pattern mining is often an intermediate step in finding all association rules of interest.

Thus a further question about local pattern mining is what a relational association rule is. Because local patterns are now clauses of the form $key \leftarrow query$, it is tempting to define association rules as

$$\mathsf{IF}\ query\ \mathsf{THEN}\ literal$$

However, it is unclear how to interpret this expression as it is not a clause.

Example 6.11. Consider the relational rule

$$\mathsf{IF\ participant(K, J, C, P)\ THEN\ subscription(K, C)}$$

This rule denotes that participants typically subscribe to *some* course because when the clause

$$\mathsf{key(K)} \leftarrow \mathsf{participant(K, J, C, P)}$$

covers some participant θ, the clause

$$\mathsf{key(K)} \leftarrow \mathsf{participant(K, J, C, P), subscription(K, C)}$$

typically also covers θ. Thus the meaning is different from the clause

$$\mathsf{subscription(K, C)} \leftarrow \mathsf{participant(K, J, P, C)}$$

which denotes that participants take *all* courses as all variables are universally quantified. So, new variables introduced in the conclusion part of a relational association rule are *existentially* quantified, which explains also why we stick to the IF – THEN notation. Also, the support and confidence of this rule are both 100%.

By now it is easy to define the problem of relational association rule mining:

Given

- a relational database D,

- the entity of interest determining the *key*,
- a support threshold s,
- a confidence threshold c,
- a language $\mathcal{L}$ of clauses of the form $key \leftarrow b_1, ..., b_n$.

Find: all relational association rules r of the form IF q THEN l, where $(key \leftarrow q)$ and $(key \leftarrow q, l) \in \mathcal{L}$, such that $support(r) \geqslant s$ and $confidence(r) \geqslant c$.

It is well-known in the field of data mining that when all frequent item-sets are known, it is easy to derive the corresponding association rules. This idea can also be applied to the relational case. Indeed, for every frequent pattern $key \leftarrow l_1, ..., l_n$ (with relative frequency r) consider the association rules

$$\textsf{IF } l_1, ..., l_{i-1}, l_{i+1}, ..., l_n \textsf{ THEN } l_i$$

This association rule will have a support of r. The confidence can be derived from r and the support of rules involving $key \leftarrow l_1, ..., l_{i-1}, l_{i+1}, ..., l_n$. Therefore, most association rule miners start by first computing all frequent item-sets and then using them to compute the corresponding association rules. As this second step does not involve accessing the database, it is less time consuming than the first one, which also explains why most research in local pattern mining is concerned with finding the frequent patterns. This is not different in the relational case.

6.5.2 Computing Frequent Queries

Algorithms for computing the set of frequent patterns are based on the anti-monotonicity of the frequency constraint, as discussed in Sect. 3.7. As expected, the frequency constraint is also anti-monotonic in the relational case. The reader may want to verify that for all clauses c_1, c_2 for which c_1 θ-subsumes (or OI-subsumes) c_2, for all databases D, and thresholds t

$$freq(c_2, D) \geqslant t \rightarrow freq(c_1, D) \geqslant t \qquad (6.7)$$

Therefore, one can directly apply Algo. 3.3 which finds all hypotheses that satisfy an anti-monotonic constraint using general-to-specific search. Most instantiations of this algorithm for frequent pattern mining, however, perform some optimizations. An important optimization, introduced in APRIORI by Agrawal and Srikant [1994], is shown in Algo. 6.4. Because the database may be very large, it is desirable to minimize the number of passes through the database. Therefore, Algo. 6.4 computes the frequent patterns *level-wise*, that is, at each level i, it first computes the candidate clauses Can_i and then computes the frequencies of all clauses in Can_i by scanning the database once. Thus Algo. 6.4 is an instantiation of Algo. 3.3, where the two for loops have been inverted and where the search proceeds in a level-wise fashion.

This algorithm applies directly to the relational case though there are a number of subtleties. First, the level-wise scanning of the database works only

Algorithm 6.4 Computing frequent clauses

```
$Can_0 := \{(\top, 0)\}$
$i := 0$
**while** not $Can_i = \emptyset$ **do**
  **for** all $d \in D$ **do**
    **for** all $(h, f) \in Can_i$ **do**
      **if** $h$ covers $d$ **then**
        increment $f$
      **end if**
    **end for**
  **end for**
  $Th_i := \{h \mid (h, f) \in Can_i$ and $f \geqslant t\}$
  $Can_{i+1} := \{\rho_o(h) \mid h \in Can_i \cap Th_i\}$
**end while**
**return** $\cup_i Th_i$
```

if one can load the examples one by one. Although this is easy when working with interpretations, it is harder using the full database approach that we have adopted. One might of course load all possible entities of interest (say all participants) as atoms for the *key*, but using standard database techniques there may be more efficient ways to compute the number of covered entities of interest. Second, Algo. 6.4 employs an optimal specialization operator. When working under θ-subsumption such operators do not exist. Therefore, it is better to work under OI-subsumption. However, when working under OI-subsumption, the implementation of an optimal refinement operator is also both complex and computationally expensive; cf. [Nijssen and Kok, 2003].

Early versions of relational pattern mining systems, such as Dehaspe's WARMR system [1997, 2001], introduced the problem of relational association rule discovery, and employed θ-subsumption and non-optimal refinement operators generating many duplicate queries that had to be filtered using expensive θ-subsumption tests, which explains why these early versions were rather inefficient. One improvement studied in the item-set mining case concerns the use of the APRIORI *join* to generate the candidate set at the next level. This *join* operation basically computes

$$Can_{i+1} \leftarrow \{l \cup k \mid l, k \in Can_i \text{ and } \mid l \cap k \mid = i - 1\} \tag{6.8}$$

This join operation is responsible for much of the efficiency of APRIORI. In principle, it can be upgraded using the *glb* operation under θ-subsumption or OI-subsumption though this has, as far as the author is aware, not yet been introduced in relational pattern miners. Other possible improvements are concerned with the clever use of data structures and database techniques to store the elements in the search space and to compute the frequencies.

Let us conclude this section by providing an example run of the frequent clause mining algorithm.

Example 6.12. Reconsider the Bongard problem illustrated in Fig. 4.4. Assume the entities of interest are the different scenes, and that the available predicates are in(K, O1, O2), stating that in scene K, object O1 is inside O2, circle(K, C) and triangle(K, T), and assume that the language $\mathcal{L}$ specifies that no constants be used. Assuming the frequency threshold on the positive scenes is 6, the algorithm may start as follows (under OI-subsumption):

- Th_0 contains key(K) ←
- Can_1 contains
  - key(K) ← triangle(K, T) which is frequent
  - key(K) ← circle(K, C) which is frequent
  - key(K) ← in(K, O1, O2) which is frequent
- Can_2 contains
  - key(K) ← triangle(K, T1), triangle(K, T2) which is frequent
  - key(K) ← circle(K, C), triangle(K, T) which is frequent
  - key(K) ← circle(K, C1), circle(K, C2) which is not frequent as not all scenes contain two circles
  - key(K) ← in(K, O1, O2), circle(K, O1) which is not frequent (as there is no circle inside another object)
  - key(K) ← in(K, O1, O2), circle(K, O2) which is frequent
  - key(K) ← in(K, O1, O2), triangle(K, O1) which is frequent
  - key(K) ← in(K, O1, O2), triangle(K, O2) which is infrequent as there is no triangle containing another object
- ...

Exercise 6.13. * Define the problem of frequent tree or graph mining and outline an algorithm for solving it. Discuss the key challenges from an algorithmic point of view

6.6 Language Bias

Logical and relational learning systems often employ a so-called declarative bias to constrain the search through the vast space of hypotheses. Bias is typically defined as anything other than the training instances that influences the results of the learner. Bias should ideally be *declarative*, that is, explicit, transparent and understandable to the user so that the user is given the opportunity to specify and tune the bias towards her own needs. Various forms of bias are typically distinguished: *language bias*, which imposes syntactic or semantic restrictions on the hypotheses to be induced, *preference bias*, which specifies the conditions under which to prefer one hypothesis over another one, and *search bias*, which is concerned with the heuristics and search strategy employed by the learner.

In this section, we will focus on language bias. Search bias has, to some extent, already been discussed in the previous sections. Search bias is also a special case of preference bias, and other forms of preference bias have not yet received much attention in the literature. At a general level, two forms of declarative language bias can be distinguished: *syntactic* and *semantic* bias. Syntactic bias, as the name indicates, merely restricts the syntax of the allowed hypotheses. Semantic bias, on the other hand, restricts the behavior of the hypotheses with regard to the examples and possibly the background theory. Both forms of bias restrict the hypotheses spaces $\mathcal{L}_h$.

6.6.1 Syntactic Bias

Formally speaking, syntactic bias is concerned with defining the syntax of the hypotheses in $\mathcal{L}_h$. From a general computer science perspective, one can regard syntactic bias as defining the well-formed elements in the formal language $\mathcal{L}_h$, a task for which one typically employs some type of grammar. A great variety of mechanisms and formalisms to specify such grammars has been developed in the inductive logic programming literature; see Nédellec et al. [1996] for an overview. Even though these formalisms often reflect the personal preferences or needs of their developers, there exist a few principles that underlie virtually all syntactic biases employed. These include the use of *predicate*, *type* and *mode* declarations.

Types and Modes

The predicate declarations specify the predicates to be used, and the type declarations the corresponding types of the predicates. Such declarations are often written as $type(pred(type_1, ..., type_n))$, where *pred* denotes the name of the predicate and the $type_i$ denote the names of the types. In addition, there can (but need not) be type definitions, which contain a specification of the domain of the different types. For instance, if the type corresponds to an attribute, one might specify the domain of the attribute using a declaration of the form $type = [v_1, ..., v_n]$, where the v_i are the different values the attribute or type can take, or, if it is continuous, one might specify the range. Furthermore, if the type allows for structured terms, the allowed function symbols can be specified. For example, to specify a type allowing for the set of terms $\mathsf{F} = \{\mathsf{f(g(f(g(...f(0)...))))}\}$ the following declarations

$$\mathsf{nul} = 0;\ \mathsf{G} = \mathsf{g(F)};\ \mathsf{F} = \mathsf{f(G)}\ \text{or}\ \mathsf{f(nul)}$$

recursively define the set of terms F. Type declarations are important because they allow the refinement operators to generate only hypotheses that are *type-conform.* A hypothesis is type-conform if it satisfies the type restrictions.

Example 6.14. For instance, given the type declarations

$$\begin{array}{l}\mathsf{type(p(F))}\\ \mathsf{type(lives(person, location))}\\ \mathsf{type(in(location, city))}\end{array}$$

the clauses

$$\begin{array}{l}\mathsf{p(f(f(f(0))))}\\ \leftarrow \mathsf{lives(X, Y), in(X, C)}\end{array}$$

are not type-conform, but the clauses

$$\begin{array}{l}\mathsf{p(f(g(f(0))))}\\ \leftarrow \mathsf{lives(X, Y), in(Y, C)}\end{array}$$

are.

In addition to types, there are also *modes*, which specify restrictions on the order of literals in clauses. The way that literals are ordered in a clauses may determine whether the clause is potentially relevant (and may contribute to distinguishing positive from negative examples), the efficiency of testing the clause, and, when working with functors (as in program synthesis), whether calls to the corresponding predicate terminate. A *mode declaration* is an expression of the form $mode(pred(m_1, ..., m_n))$, where the m_i are different modes. Typically, three modes are distinguished: input (denoted by “+”), output (denoted by “-”) and ground (denoted by “#”). The input mode specifies that at the time of calling the predicate the corresponding argument must be instantiated, the output mode specifies that the argument will be instantiated after a successful call to the predicate, and the constant mode specifies that the argument must be ground (and possibly belong to a specified type). A clause $h \leftarrow b_1, ..., b_n$ is now *mode-conform* if and only if

1. any input variable in a literal b_i appears as an output variable in a literal b_j (with $j < i$) or as an input variable in the literal h,
2. any output variable in h appears as an output variable in some b_i,
3. any arguments of predicates required to be ground are ground.

Example 6.15. Consider the following declarations:

$$\begin{array}{l}\mathsf{mode(molecule(-)).\ mode(atom(+, -, \#, \#)).\ mode(bond(+, +, -, \#)).}\\ \mathsf{type(molecule(m)).\ type(atom(m, a, at, r)).\ type(bond(m, a, a, bt)).}\end{array}$$

Then the clauses

$$\begin{array}{l}\mathsf{molecule(M)} \leftarrow \mathsf{atom(B, A, c, 3)}\\ \mathsf{molecule(M)} \leftarrow \mathsf{atom(M, A, D, 3), bond(M, B, A, C)}\end{array}$$

are not mode-conform because the variable B does not satisfy the input mode and C is not ground. The following clauses, however, satisfy the modes:

$$\begin{array}{l}\mathsf{molecule(M)} \leftarrow \mathsf{atom(M, A, c, 3)}\\ \mathsf{molecule(M)} \leftarrow \mathsf{atom(M, A, c, 3), bond(M, A, B, double)}\end{array}$$

Very often, inductive logic programming systems integrate the notation for modes and types, summarizing the above declarations as

$$\begin{array}{l}\mathsf{mode(molecule(-m)).}\\ \mathsf{mode(atom(+m - a, \#at, \#r)).}\\ \mathsf{mode(bond(+m, +a, -a, \#bt)).}\end{array}$$

Other Syntactic Restrictions

Many possibilities exist to enforce other syntactic restrictions on the language of hypotheses $\mathcal{L}_h$. First, there exist *parametrized* biases. These are syntactic biases in which the language $\mathcal{L}_h$ is defined as a function of a set of parameters. For instance, one can restrict the number of literals, variables or simply the size of clauses in $\mathcal{L}_h$. Second, there exist *clause-schemata*. Schemata are essentially second-order clauses, where the predicate names are replaced by predicate variables, for example, the clause

$$\mathsf{P(X)} \leftarrow \mathsf{Q(X,Y), R(X)}$$

with predicate variables P, Q, and R. A schemata can be instantiated by replacing the predicate variables by predicate names to yield a proper clause. For instance, the above schema could be instantiated to the clause

$$\mathsf{mother(X)} \leftarrow \mathsf{parent(X,Y), female(X)}$$

A set of schemata then represents the language consisting of all clauses that are an instance of a schema.

Third, and perhaps most elegantly, William Cohen [1994a] has proposed to use a variant of definite clause grammars to directly represent the set of clauses allowed in the hypotheses (cf. Sect. 4.6 for a discussion of grammars).

Example 6.16. * Consider the following grammar (inspired on Cohen's Grendel system, but simplified for readability)

$$\begin{array}{l}
\mathsf{S} \rightarrow [\mathsf{illegal(A,B,C,D,E,F)} \leftarrow], \mathsf{Body} \\
\mathsf{Body} \rightarrow [] \\
\mathsf{Body} \rightarrow \mathsf{literal,\ Body} \\
\mathsf{literal} \rightarrow [\mathsf{A}]\mathsf{pred}[\mathsf{C}] \\
\mathsf{literal} \rightarrow [\mathsf{A}]\mathsf{pred}[\mathsf{E}] \\
\mathsf{literal} \rightarrow [\mathsf{E}]\mathsf{pred}[\mathsf{C}] \\
\mathsf{literal} \rightarrow [\mathsf{B}]\mathsf{pred}[\mathsf{D}] \\
\mathsf{literal} \rightarrow [\mathsf{D}]\mathsf{pred}[\mathsf{F}] \\
\mathsf{literal} \rightarrow [\mathsf{B}]\mathsf{pred}[\mathsf{F}] \\
\mathsf{pred} \rightarrow [=] \\
\mathsf{pred} \rightarrow [\neq] \\
\mathsf{pred} \rightarrow [<] \\
\mathsf{pred} \rightarrow [>]
\end{array}$$

This grammar defines various possibilities for the $\mathsf{illegal}$ predicate, which succeeds when the chessboard with the white king at $\mathsf{A,B}$, the black king at $\mathsf{C,D}$ and the black rook at $\mathsf{E,F}$ is an illegal chess position. The terminal symbols are written between square brackets. As the reader may want to verify, the clause

$$\mathsf{illegal(A,B,C,D,E,F)} \leftarrow \mathsf{A = C, B < D}$$

is within the language of the definite clause grammar, but the clause

$$\mathsf{illegal(A, B, C, D, E, F)} \leftarrow \mathsf{A = B}$$

is not.

Regardless of the formalism employed, the designer of the logical or relational learning system must ensure that the bias mechanism can be incorporated in the search strategy. Often this is realized by defining a special-purpose refinement operator that only generates clauses within the language of hypotheses.

Exercise 6.17. Define a generality relationship for rule schemata and discuss its properties. How would you use it for structuring the search space? (The solution to this exercise forms the basis for the system MOBAL [Morik et al., 1993].)

Exercise 6.18. Outline a specialization operator employing the definite clause grammar bias. (Hint: a sequence of derivation steps in the grammar corresponds to a specialization under θ-subsumption.)

6.6.2 Semantic Bias

In addition to syntactic restrictions, it can be desirable to constrain the behavior of the target predicate by imposing semantic restrictions on the clauses in the hypotheses.

At least two such restrictions have been employed in the literature. The first restriction is that to *determinate* clauses, which we encountered in Sect. 4.12.1 when talking about propositionalization. A clause $h \leftarrow b_1, ..., b_n$ is *determinate* if and only if for all substitutions θ that ground h, and all i, there exists at most one substitution σ such that

$$b_1\theta\sigma, ..., b_n\theta\sigma \text{ succeeds in } B \tag{6.9}$$

Example 6.19. Under the usual semantics of family relationships, the clause

$$\mathsf{isafather(F)} \leftarrow \mathsf{parent(F, C), male(F)}$$

is not determinate, because F may have multiple children C. However, the clause

$$\mathsf{father(F, C)} \leftarrow \mathsf{parent(F, C), male(F)}$$

is determinate.

Another form of semantic bias is concerned with learning *functional* predicates. A predicate p/n is *functional* if and only if if satisfies the constraint

$$X = Y \leftarrow p(X_1, ..., X_n, X), p(X_1, ..., X_n, Y) \tag{6.10}$$

where we have, for convenience, written the functional argument as the last one.

Example 6.20. Consider the predicates multiply(X, Y, P) and father(F, C) under the usual definitions. multiply is functional because for any given pairs of numbers X, Y there is exactly one product. father is functional because any child C has exactly one father.

Functional predicates arise in many situations and there exist specialized inductive logic programming systems for inducing them, such as FFoil [Quinlan, 1996]. One can enforce that a functional predicate is induced by checking that the generated hypotheses satisfy the constraint. If they do not, the predicate definition for p/n must be overly general, and hence it must be specialized. Indeed, assume that the constraint for the father predicate is violated. Then there must be somebody with two fathers, and one of the corresponding facts is covered but must be false. Therefore the definition of father is overly general. A more general technique for dealing with constraints is presented in Sect. 7.2.2.

Exercise 6.21. Discuss the differences and similarities between the use of constraints to specify a semantic bias and the use of constraints for semantic refinement; cf. Sect. 5.6.3.

6.7 Conclusions

This chapter introduced a methodology for developing logical and relational learning algorithms and systems. The methodology starts by identifying a propositional learning or mining system of interest, and then first upgrades the problem setting by changing the representation language of hypotheses and examples and the covers relation, and then upgrades the propositional learning algorithm in a systematic manner by modifying its operators. The methodology is effective and it was shown at work on three case studies from the literature of inductive logic programming: the Foil system for rule-learning, the Tilde system for logical decision tree induction, and Warmr for relational association rule induction. The resulting systems also have the desirable properties that the original propositional system is a special case of the upgraded one. Finally, we also discussed various ways to restrict the search space of logical and relational learning systems by means of a declarative language bias. This is often necessary to make the search more tractable even though it imposes an additional burden on the user.

6.8 Bibliographic Notes

This chapter borrows many ideas and principles of Van Laer and De Raedt [2001], De Raedt et al. [2001].

Many inductive logic programming systems can be regarded as an extension of the Foil system of Quinlan [1990]. Some of the most important developments include: mFoil [Lavrač and Džeroski, 1994], which aims at handling noise and whose Prolog source code is available in the public domain, FFoil [Quinlan, 1996], which learns a functional predicate $p(t_1, ..., t_n)$ implementing the function $pf(t_1, ..., t_{n-1}) = t_n$, Focl [Pazzani and Kibler, 1992], which employs resolution steps to obtain additional refinements, and Foidl [Mooney and Califf, 1995], which learns decision lists rather than unordered rule sets. Other systems, such as Muggleton's prominent Progol system [Muggleton, 1995] and Srinivasan's Aleph system [Srinivasan, 2007], possess some similarities with Foil, even though Aleph and Progol were developed from a richer logical perspective. Aleph and Progol apply inverse entailment (cf. Sect. 5.6.1) on a positive example and then heuristically search the θ-subsumption lattice more generally than the resulting bottom clause. Furthermore, rather than employing a beam search, Progol is based on an A*-like search mechanism. Another early inductive logic programming system is Golem [Muggleton and Feng, 1992]. It combines the cautious specific-to-general Algo. 3.5 with the covering algorithm. It provides an answer to Ex. 6.7.

Several relational or logical decision tree learners have been developed. These include the early work of Watanabe and Rendell [1991], the regression tree learner S-Cart [Kramer, 1996], the system Tilde by [Blockeel and De Raedt, 1998], who also studied the expressive power of logical decision trees as compared to relational learners that induce rules or decision lists, and the more recent relational probability trees by Neville et al. [2003], which contain probability estimates in the leaves of the tree. A related approach by Boström and Idestam-Almquist [1999] discusses a divide-and-conquer approach based on performing resolution steps (that is, unfolding) on an overly general logic program. A more recent approach computes aggregates of relational decision trees in the form of relational random forests [Van Assche et al., 2006].

Frequent queries and relational association rules were introduced by Dehaspe and De Raedt [1997], Weber [1997], but became popular in data mining with an application in predictive toxicology [Dehaspe et al., 1998, Dehaspe and Toivonen, 2001]. Since then they have received quite some attention; especially their efficient computation was studied by Nijssen and Kok [2001, 2003]. Recently, condensed representations (such as closed queries; cf. Ex. 3.8) for relational queries have been developed [De Raedt and Ramon, 2004, Garriga et al., 2007]. Furthermore, the work on relational frequent query mining and its applications has inspired much of the work on frequent graph mining, which has been very popular over the past few years [Inokuchi et al., 2003, Kuramochi and Karypis, 2001, Yan and Han, 2002, Nijssen and Kok, 2004].

Many different forms of declarative bias have been studied in the literature. Modes and types were already introduced in the Model Inference System of Shapiro [1983] and they have become standard since their incorporation in the Progol system by Muggleton [1995]. Parametrized language restrictions were employed by the Clint system of De Raedt [1992], who also studied ways to shift the bias when the language was too restricted. Clause schemata were introduced in Blip [Emde et al., 1983] and used in Mobal [Morik et al., 1993]. The definite clause grammar bias was first used in Grendel [Cohen, 1994a]. Determinacy restrictions were employed in Linus [Lavrač and Džeroski, 1994] and Golem [Muggleton and Feng, 1992]. An overview of declarative bias is contained in [Nédellec et al., 1996].

7

Inducing Theories

Whereas the previous chapter focussed on learning the definition of a single predicate in a classification and discovering properties of the entities of interest in the form of frequent patterns, the present chapter investigates the complications that arise when learning multiple, possibly interrelated predicates. This chapter is thus concerned with the induction and revision of logical theories from examples. In its most general form, logical theories are pure Prolog programs (that is, sets of definite clauses) and examples provide information about the input and output behavior of the intended target program and theory induction corresponds to program synthesis from examples, an old dream of any programmer. In a more modest and realistic scenario, the logical theory represents a knowledge base in an expert system, and the user provides examples of correct and incorrect inferences. In both cases, the purpose is to learn a theory that is correct with regard to the evidence.

Because the problem of theory induction is a very general one, different settings have been studied, including incremental learning, active learning and the use of an oracle, and revision of an existing theory rather than learning one from scratch. At the same time, a wide variety of different techniques are useful in a theory induction setting, which explains why the chapter is divided into five different sections. Sect. 7.1 introduces the problems of theory revision and induction. Section 7.2 focuses on abductive reasoning, a form of reasoning that is studied in the field of abductive logic programming and that aims at inferring missing facts for specific observations. Abduction is complementary to induction, which focuses on inferring general rules explaining different observations. Abduction plays an important role in theory revision, as it can be used to relate observations for different predicates to one another. It will also be shown how integrity constraints can be used to constrain the search space in abductive reasoning and theory revision.

Section 7.3 addresses the theory revision problem, where we especially focus on the influential MODEL INFERENCE SYSTEM *of Shapiro [1983], which will be used as a showcase system throughout this chapter. The* MODEL INFERENCE SYSTEM *can not only induce theories in the form of knowledge bases* *but can also synthesize programs from examples. Because theory revision in first-order logic as incorporated in the* Model Inference System *is quite involved, it is worthwhile to study also some simpler settings that illustrate alternative views on theory revision. Therefore, Sect. 7.4 introduces two famous propositional theory revision systems, the* Horn *algorithm by Angluin et al. [1992] and the* Duce *system by Muggleton [1987]. Finally, we end this chapter by investigating in Sect. 7.5 how a set of integrity constraints can be induced from a theory. This complements the use sketched above of integrity constraints to constrain the search space, and it is quite useful in a database setting.*

7.1 Introduction to Theory Revision

In this section, we introduce the problem of theory induction and revision, relate it to the important notion of model inference, and present a simple though general algorithm for tackling theory revision problems.

7.1.1 Theories and Model Inference

In this book, the term *theory* refers to a pure Prolog (or definite clause) program. The basic *theory induction* setting is as follows:

Given

- a set of positive and negative examples E
- a language of clauses $\mathcal{L}$
- a covers relation c
- possibly a background theory B (a set of definite clauses)

Find a set of clauses $T \subseteq \mathcal{L}$ such that T (possibly together with B) covers all positive and no negative examples.

Typically, when learning theories, the covers relation corresponds to entailment though it is also possible to learn theories from entailment. Furthermore, as the semantics of a definite clause program P is given by its least Herbrand model $M(P)$ (cf. Chapter 2), it is convenient to view a theory as a model. It is this perspective that Ehud Shapiro [1983] adopted in his influential Model Inference System and that is now standard within inductive logic programming. The problem of learning a theory is then that of inferring the right model (represented using a logical theory). In reasoning about inferring theories, it is often assumed that there is an unknown target model, the so-called *intended interpretation*, that captures the semantics of the unknown target theory. Positive examples then take the form of ground facts that belong to the intended interpretation, and negatives are ground facts not belonging to the intended interpretation. This view is equivalent to learning from entailment when the theory contains definite clauses (without negation).

Example 7.1. Consider the following target theory

```
rank(7) ←                                   rank(8) ←
rank(9) ←                                   rank(10) ←
rank(a) ←                                   rank(k) ←
rank(q) ←                                   rank(b) ←
red(h) ←                                    red(d) ←
black(s) ←                                  black(c) ←
pair(c(X, Y1), c(X, Y2)) ←
suit(X) ← black(X)
suit(X) ← red(X)
card(c(X, Y)) ← rank(X), suit(Y)

triple(C1, C2, C3) ← pair(C1, C2), pair(C2, C3)
fullhouse(C1, C2, C3, C4, C5) ← pair(C1, C2), triple(C3, C4, C5)
```

Starting from the definitions of card, pair (and the predicates rank, suit, red and black), and some positive examples such as triple(c(7, h), c(7, d), c(7, s)) and fullhouse(c(a, s), c(a, d), c(7, h), c(7, d), c(7, s)), the task could be to induce the definitions triple and fullhouse.

Example 7.2. * Consider the program

```
append([], X, X) ←
append([X|XR], Y, [X|ZR]) ← append(XR, Y, ZR)
```

This program can be learned from examples such as append([], [a, b], [a, b]) and append([a, b], [c, d], [a, b, c, d]).

The first example illustrates a *multiple predicate learning* problem, because one learns the definitions of multiple predicates; the second example illustrates a *program synthesis* setting. In both types of problems, the clauses in the target theory are interdependent. In the cards example, the definition of learn fullhouse depends on that of pair and triple. Changing the definition of pair also affects the coverage of fullhouse. Similarly, in the program synthesis illustration, the semantics of the recursive clause depends on the presence of the base case. In order to develop algorithms for inferring theories, we need to cope with these interdependencies. Interdependencies also introduce ordering effects that need to be taken into account. In the cards example, it is easier to learn fullhouse when pair and triple are known; in the program synthesis example, one cannot learn the recursive case without taking into account the base case.

Exercise 7.3. Consider learning the predicates aunt and sister that represent family relationships. Assume the predicates male, female and parent are given. Discuss the interdependencies and ordering effects when learning aunt and sister.

7.1.2 Theory Revision

Very often it will be easier to revise a given theory than to learn one from scratch. This is the problem of *theory revision* that can be specified as follows:

Given

- a set of positive and negative examples E
- a language of clauses $\mathcal{L}$
- an initial theory T
- a covers relation c
- possibly a background theory B (a set of definite clauses)

Find a set of clauses $T' \subseteq \mathcal{L}$ such that T' (possibly together with B) covers all positive and no negative examples and is as close to T as possible.

Of course, there are various measures of closeness between theories that can be applied. Most approaches to theory revision, however, define these notions only implicitly, for instance, by trying to minimize the number of steps through the search space.

In this chapter, we will focus on theory revision instead of theory induction. Furthermore, we will study theory revision in an incremental and possibly interactive setting. *Incremental* learners process their examples one by one and *interactive* learners are allowed to pose queries to the user. Interactive learning is sometimes called *active* learning or learning from *queries*, and the user is sometimes referred to as the *oracle*. Various types of queries are possible; cf. also Sect. 7.3.1. The reader should, however, keep in mind that most of the principles underlying such learning systems also apply to theory induction and non-incremental learners; the key differences typically lie in the heuristics and search strategy employed.

Algorithm 7.1 A theory revision algorithm

```
while $T$ is not (complete and consistent) do
  if $T$ covers negative examples then
    choose a specialization in $\gamma_s(T)$
  end if
  if $T$ does not cover all positive examples then
    choose a generalization in $\gamma_g(T)$
  end if
end while
```

A general way of dealing with theory revision problems is illustrated in Algo. 7.1. It starts from an initial theory T and refines it whenever it is inconsistent with a given example. As in Chapter 3, it specializes the given theory whenever the theory covers a negative example, and generalizes it whenever it does not cover a positive example. The key difference with the techniques seen so far is that a refinement operator γ at the theory level is now employed.

Operators at the theory level start from theories, that is, *sets* of clauses, and have to specialize or generalize these. In Chapter 5, we have already seen operators that work on sets of clauses when dealing with (inverse) resolution. However, one can also refine a theory by refining one of its clauses. These theory refinement operators γ typically employ an underlying clausal refinement operator ρ. Two typical theory refinement operators work as follows.

The *theory generalization operator* $\gamma_{\rho,g}$ employing the clausal generalization refinement operator ρ_g is defined as

$$\gamma_{\rho,g}(T) = \{T - \{c\} \cup \{c'\} \mid c' \in \rho_g(c) \text{ with } c \in T\} \cup \{T \cup \{c\} \mid c \in \mathcal{L}_h\} \quad (7.1)$$

The *theory specialization operator* $\gamma_{\rho,s}$ employing the clausal specialization refinement operator ρ_s is defined as follows:

$$\gamma_{\rho,s}(T) = \{T - \{c\} \cup \{c'\} \mid c' \in \rho_s(c) \text{ with } c \in T\} \cup \{T - \{c\} \mid c \in T\} \quad (7.2)$$

The generalization operator $\gamma_{\rho,g}$ either generalizes one of the clauses in the theory or adds a new clause. The specialization operator $\gamma_{\rho,s}$ either specializes one of the clauses or deletes an entire clause.

Example 7.4. Consider the theory consisting of the following clauses:

flies(X) ← bird(X)
bird(X) ← ostrich(X)
normal(X) ← blackbird(X)
ostrich(oliver) ←
bird(X) ← blackbird(X)

and the negative example flies(oliver) that is covered by this theory. Various refinements correctly account for this example, for instance:

1. retract flies(X) ← bird(X)
2. retract bird(X) ← ostrich(X)
3. retract ostrich(oliver) ←
4. specialize flies(X) ← bird(X) into flies(X) ← bird(X), normal(X)

Assume that we select the fourth option, and then encounter an uncovered positive example flies(tweety) ←. To deal with this example, there are again various options, such as:

1. assert flies(tweety) ←
2. assert bird(tweety) ← and normal(tweety) ←
3. assert blackbird(tweety) ←

The last operation appears to be the most desirable one from a user perspective.

The example illustrates various important issues when revising theories. First, there are a vast number of possible refinements that can be performed. Whereas in the example, single-step refinements typically suffice, in general multiple-step refinements may be necessary. The key problem in theory revision will thus be to control the combinatorial explosion. Second, the example illustrates the use of *abduction* to effectively deal with the uncovered positive example. Whereas inductive reasoning aims at generalizing specific facts into general rules, abductive reasoning infers missing facts in order to explain specific observations. Abductive reasoning is dealt with in the next section. Third, the example also illustrates the order effects and dependencies in theory revision. In particular, the predicate flies/1 is changed by modifying the underlying predicates bird/1 and normal/1. Furthermore, the order of processing the examples matters.

Exercise 7.5. Describe what changes when processing the examples in Ex. 7.4 in the reverse order.

7.1.3 Overview of the Rest of This Chapter

The remainder of this chapter discusses different aspects of theory revision in four different sections. Section 7.2 introduces abductive logic programming, which combines abductive reasoning with logic programming. Abductive logic programming, where the operators are only allowed to add or delete facts, can be considered a special case of theory revision. To introduce abductive logic programming, Sect. 7.2.1 will discuss abductive operators and reasoning in more detail, Sect. 7.2.2 will show how powerful constraints that specify restrictions on the intended interpretations can be employed, and then Sect. 7.2.3 will study abduction under constraints, the usual setting for abductive logic programming. Once the principles of abduction have been explained, we shall turn our attention to the original problem of theory revision. In Sect. 7.3, we focus on the seminal Model Inference System by Shapiro [1983], which is, after so many years, still one of the most elegant and powerful theory revision and program synthesis systems. As the Model Inference System is interactive, the influence of the use of queries that can be posed by the learner will be studied in Sect. 7.3.1 before our presenting the actual Model Inference System in Sect. 7.3.2. While the Model Inference System works with very expressive representations — that of definite clause logic as in Prolog — other theory revision systems have studied simpler settings using different principles.

Two famous propositional theory revision systems will be the topic of Sect. 7.4, in which we shall first introduce the Horn algorithm by Angluin et al. [1992] in Sect. 7.4.1 and then the Duce system by Muggleton [1987] in Sect. 7.4.2. Horn is an algorithm that induces propositional Horn theories in polynomial time using particular types of queries to an oracle, whereas Duce is a heuristic propositional theory revision system based on inverse resolution.

Finally, in Sect. 7.5, we shall show how integrity constraints in the form of full clausal theories can be inferred from data. This is, as we shall argue, a kind of inverse theory revision problem.

7.2 Towards Abductive Logic Programming

This section introduces the principles of abductive logic programming, the study of abductive reasoning within the representations offered by logic programming. The first subsection explains what abductive reasoning is and defines the abductive inference operator. The second subsection then shows how integrity constraints can be used to constrain the solutions returned in theory revision and abductive logic programming. Finally, in the last subsection we present a simple abductive logic programming algorithm.

7.2.1 Abduction

Abduction, as it is commonly used in computational logic, is a procedure to infer missing facts from a theory. The basic *abductive inference operator* can be derived by inverting the following deductive inference rule:

$$\frac{p \leftarrow q_1, ..., q_n \text{ and } q_1\theta \wedge ... \wedge q_n\theta}{p \leftarrow q_1, ..., q_n \text{ and } p\theta} \tag{7.3}$$

That is, abduction starts from a clause $p \leftarrow q_1 \wedge ... \wedge q_n$ and a fact $p\theta$ and infers that $q_1\theta \wedge ... \wedge q_n\theta$ must be true. This also implies that, abduction like induction, can be regarded as a form of inverted deduction; cf. Sect. 5.1.

Example 7.6. Consider the clause $\mathsf{flies(X)} \leftarrow \mathsf{bird(X), normal(X)}$ and the fact $\mathsf{flies(tweety)}$. Abduction infers that $\mathsf{bird(tweety)}$ and $\mathsf{normal(tweety)}$ are both true.

When the clause used by the abductive operator has variables in its body that do not appear in its head, the abduced facts may not be ground. In such situations, one typically applies a skolem substitution (which replaces the variables in $body(c)\theta$ with new constants) or consults the user.

The abductive operator can be used to address the dependency problem in theory revision. Indeed, examples for one predicate can be reduced to those for other predicates. By revising the theory to correctly handle these other examples, the original theory revision problem can be solved. The following scheme based on the abductive operator defined in Eq. 7.3 can be used for this purpose.

- If $p\theta$ is a positive example and the clause $p \leftarrow q_1, ..., q_n$ belongs to the theory, then infer that the $q_1\theta$ and ... and $q_n\theta$ are positive examples.
- If $p\theta$ is a negative example and the clause $p \leftarrow q_1, ..., q_n$ belongs to the theory, then infer that $q_1\theta$ or ... or $q_n\theta$ must be a negative example.

The first case is a direct application of the abductive inference operator. The second case corresponds to its contra-position. It is especially useful when the clause is used in a proof of $p\theta$ from the theory. Indeed, the proof will no longer hold if one of the $q_i\theta$ no longer follows from the theory.

Example 7.7. Reconsider Ex. 7.4. If flies(tweety) is a positive example, it will be covered if the two examples bird(tweety) and normal(tweety) are covered. If on the other hand it is a negative example that is covered and the clause for flies/1 is needed to prove normal(tweety), then flies(tweety) will no longer be covered when either bird(tweety) or normal(tweety) is no longer covered.

One way of incorporating the abductive operators into a theory revision algorithm, is to work with a procedure $\mathbf{tr}(E)$ that takes the current set of examples E as a parameter. When the abductive operator is applied on a positive example, the procedure is called recursively using $\mathbf{tr}(E \cup \{q_1\theta, ..., q_n\theta\})$; when it is applied on a negative example, one can use the following calls: $\mathbf{tr}(E \cup \{q_1\theta\})$, ..., $\mathbf{tr}(E \cup \{q_n\theta\})$. If one of these calls succeeds, then return success.

7.2.2 Integrity Constraints

The use of integrity constraints in theory revision has been motivated by its use in abductive logic programming and belief revision, where integrity constraints provide a powerful means to constrain the abduced theories in a declarative manner. Integrity constraints play an important role as in traditional databases, where they impose restrictions on the state (or extension) the database can be in. In a theory revision context, they constrain the theories that can be generated. In this chapter, we shall view an *integrity constraint* as a (general) clause that is true in the intended interpretation. Furthermore, a theory *satisfies* an integrity constraint if and only if the constraint is true in the least Herbrand model of the theory. If a theory does not satisfy a constraint, it *violates* the constraint.

The theory revision problem can now be modified to allow the user to specify integrity constraints and to require that the generated theory satisfy all constraints.

Example 7.8. Consider the theory shown in Ex. 7.4. The constraint

$$\mathsf{vampire(X); normal(X)} \leftarrow \mathsf{flies(X)},$$

which states that only normal animals and vampires fly, is violated for $\mathsf{X} = \mathsf{oliver}$.

Observe that facts that are true or false in the intended interpretation, that is, positive and negative examples, can be represented as integrity constraints. Indeed, a positive example p corresponds to a constraint of the form

$p \leftarrow$; a negative example n corresponds to a constraint $\leftarrow n$. Thus, integrity constraints are more expressive than examples.

The reason for employing integrity constraints in theory revision and abductive logic programming is that they provide the user with a very powerful means to specify necessary properties of the target theory. Furthermore, they are not restricted to definite clauses like the target theory. Also, even for constraints in the form of definite clauses, adding these constraints directly to the theory is not an option as this leads to complicated theories with many redundancies; this in turn complicates and slows down the theorem-proving process. Finally, certain constraints should never be used as inference rules. Think, for example, about the rule

$$\mathsf{Y} < \mathsf{150} \leftarrow \mathsf{human}(\mathsf{X}), \mathsf{age}(\mathsf{X}, \mathsf{Y}),$$

which should not be used as part of a definition of the predicate $<$. This reveals that integrity constraints allow the user to make the theory revision process very knowledge intensive.

The general principle for using constraints for theory revision is in a way similar to abductive reasoning. When a constraint $q_1; ...; q_m \leftarrow p_1, ..., p_k$ is violated by theory T for a substitution θ we have that:

- either the definition of one of the predicates p_i in the body of the constraint is too general (that is, T covers $p_i\theta$ but should not), or
- the definition of one of the predicates q_j in the head of the constraint is too specific (that is, T does not cover $q_j\theta$ but should).

This observation reveals that violated constraints can be used to generate examples that are not satisfied by the theory. When there is only one literal in a constraint, it is clear that the violated instance of the literal itself is an example that is not satisfied by the theory. However, if there is more than one literal in a violated constraint, the responsible examples have to be identified by performing *credit assignment* on a violated instance of the constraint.

If the system has access to a user that is willing to answer questions, that is, an *oracle* (cf. Sect. 7.3.1), the credit assignment procedure may query it for the truth-value in the intended interpretation of the instances of the literals in the violated constraint. Continuing our earlier example, the system could query the oracle for the truth-values of flies(oliver), normal(oliver) and possibly vampire(oliver) to detect the reason for the violation. This process would result in an example that is handled differently by the current theory and the intended interpretation. This example could then be passed on to the theory revision system for further processing.

When the theory revision system does not have access to an oracle, one can hypothesize the truth-value in the intended interpretation of specific examples in the same way as we handled the abduced examples in Sect. 7.2.1.

When reducing violated integrity constraints to examples, the theory revision system must, however, carefully interleave the processing of the examples and the constraints. When an example for a predicate p is incorrect in the current theory, one cannot expect the constraints mentioning the predicate p to be satisfied. Also, the predicates (and the constraints) that depend on the predicate p may be affected. A predicate p *depends* on a predicate q with regard to a given theory if and only if there is a clause in the definition of p that mentions q, or there is a clause in the definition of p that mentions a predicate r that depends on q. Therefore, one should postpone checking constraints that depend on predicates for which there are still incorrectly handled examples.

Example 7.9. Reconsider the constraint

$$\mathsf{vampire(X); normal(X)} \leftarrow \mathsf{flies(X)}.$$

If there are still incorrectly handled examples for vampire — say vampire(dracula) is a positive but uncovered example — one cannot expect the constraint to be satisfied and should therefore postpone the verification of the constraint until the positive example is satisfied.

Exercise 7.10. Can you relax the conditions for postponing the examples in situations where a predicate has either only uncovered positive examples or only covered negative examples?

7.2.3 Abductive Logic Programming

The special case of theory revision that employs integrity constraints but only allows us to add or delete missing facts is known as abductive logic programming. It has received a lot of attention in the computational logic community and can be defined as follows:

Given

- a definite clause theory T[1],
- a ground fact f,
- a set of abducible predicates A,
- a set of integrity constraints I,

Find a set of facts F for predicates in A such that $T \cup F \models f$ and $T \cup F$ is consistent with the integrity constraints I.

Abductive logic programming distinguishes predicates that are *abducible* from those that are not. For abducible predicates there can be missing facts that need to be abduced. All other predicates are assumed to be correct and complete; so for these predicates no missing facts need to be inferred.

In many cases, it is desirable to consider only *minimal* solutions, that is, minimal sets of facts that form a solution to the abductive logic programming

[1] The computational logic community has also devoted a lot of attention to dealing in a more sound manner with normal programs, which employ negation as failure [Kakas et al., 1992, Denecker and De Schreye, 1992, Flach, 1994].

problem. Throughout this section, it will be assumed, for reasons of simplicity, that the integrity constraints are denials, that is, clauses of the form $\leftarrow q_1, ..., q_n$. This simplifies the credit assignment problem in that theories that violate a denial must be overly general.

Example 7.11. Consider the theory

$$\begin{array}{ll} \mathsf{bird(X)} \leftarrow \mathsf{ostrich(X)} & \mathsf{flies(X)} \leftarrow \mathsf{bird(X), normal(X)} \\ \mathsf{bird(X)} \leftarrow \mathsf{blackbird(X)} & \end{array}$$

the integrity constraint

$$\leftarrow \mathsf{ostrich(X), normal(X)}$$

and the fact flies(tweety). Assume furthermore that ostrich, blackbird and normal are the only abducibles. Then one (minimal) solution to the abductive logic programming problem is $F = \{\mathsf{blackbird(tweety), normal(tweety)}\}$.

The example shows that abduction is a process that infers missing facts needed to explain a single observation (such as flies(tweety)). This contrasts with inductive inference, which aims at inferring general laws from several observations. Even though the technical differences between induction and abduction in computational logic can be explained in this way, there is an ongoing debate on the differences between abduction and induction in the artificial intelligence and philosophy of science literature; cf. [Flach and Kakas, 2000].

A simplified abductive procedure is shown in Algo. 7.2. The procedure is called using **abduce(**$\leftarrow p$**;** $\emptyset$**)**, where p is the fact to be derived using abduction. It is a simple extension of the traditional SLD-resolution procedure presented in Chapter 2. Upon entry, it checks whether the goal is empty; if it is, the current set of abduced facts is output. Otherwise, if the atom q_1 contains an abducible predicate, it is skolemized to $q_1\sigma$. It is then tested whether adding this atom to F still satisfies the integrity constraints. If it does, the abductive procedure is called recursively on the remaining part of the goal. If the atom q_1, on the other hand, does not contain an abducible predicate, the usual SLD-resolution operator is applied to yield the next goal.

Example 7.12. Reconsider the theory of Ex. 7.11, the goal flies(tweety) and the constraint $\leftarrow$ normal(X), ostrich(X). The resulting SLD-tree is shown in Fig. 7.1. In the leftmost branch, the fact ostrich(tweety) $\leftarrow$ is first abduced; afterwards, one also tries to abduce normal(tweety) $\leftarrow$. This fails, however, because the resulting set of facts violates the integrity constraint. The search therefore proceeds in the rightmost branch, where the facts blackbird(tweety) $\leftarrow$ and normal(tweety) $\leftarrow$ are abduced and returned by the procedure.

At this point, the reader may observe that Algo. 7.2 only works well when the integrity constraints are denials. If a denial is violated, it is valid to conclude that the theory is too general, and needs to be specialized. As the only

Algorithm 7.2 The abductive function **abduce(**$\leftarrow q_1, ..., q_n$**;** F**)**

```
if $n = 0$ then
  return $F$
else if the predicate in $q_1$ is abducible then
  compute a skolem substitution $\sigma$ such that $q_1\sigma$ is ground
  if $F \cup T \cup \{q_1\sigma\}$ satisfies $I$ then
    call abduce($\leftarrow q_2\sigma, ..., q_n\sigma$; $I$; $F \cup \{q_1\sigma\}$)
  else
    fail
  end if
else if possible then
  select the next clause $q \leftarrow r_1, ..., r_m$ in $T$ for which $mgu(q, q_1) = \theta$
  call abduce($\leftarrow r_1\theta, ..., r_m\theta, q_2\theta, ..., q_n\theta$; $F$)
else
  fail
end if
```

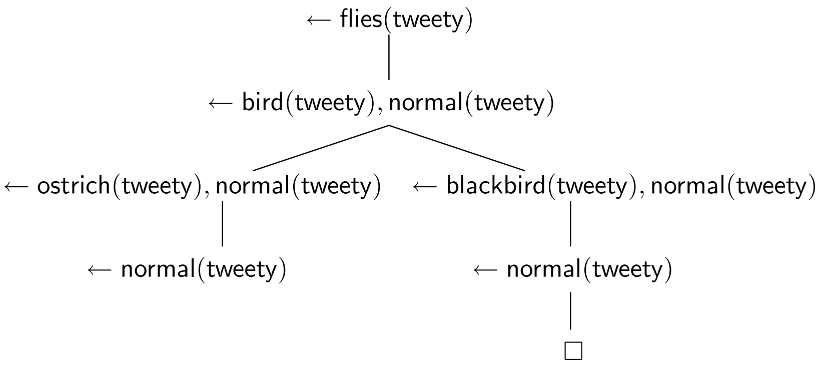

← flies(tweety)
← bird(tweety), normal(tweety)
← ostrich(tweety), normal(tweety)
← blackbird(tweety), normal(tweety)
← normal(tweety)
← normal(tweety)
□

Fig. 7.1. Illustrating abduction

available operation to the abductive procedure is to add a fact, it is correct to prune away sets of facts F that violate a denial. For constraints that are general clauses, pruning may not be justified, as illustrated below.

Example 7.13. Reconsider the previous example but assume also that the additional constraint normal(X) ← blackbird(X) is available. Then the procedure would proceed as before except that directly after abducing blackbird(tweety) ← the novel constraint would be violated, and therefore the procedure would fail.

A partial solution is to test the integrity theory only at the end, that is, just before returning the solution set.

Exercise 7.14. Why is this a partial solution only? (Hint: reconsider the previous example but replace the clause flies(X) ← bird(X), normal(X) with flies(X) ← bird(X).)

Algo. 7.2 can be adapted to find a *minimal* set of facts by using a breadth-first strategy (where the depth would be defined in terms of the size of F) instead of a depth-first one. In Chapter 8, we shall extend the algorithm to work in a probabilistic context, which allows one to associate probabilities with the different solutions, and to prefer the one with the highest probability.

Abductive logic programming is a rich research field that has successfully addressed such problems under many different settings (including the use of negation). A detailed overview of these is outside the scope of this book, but can be found in [Kakas et al., 1992].

Abductive logic programming also has interesting applications, including:

- diagnosis, where the abduced components correspond to errors in devices;
- planning and scheduling (in the event calculus), where the abduced facts denote actions, and together form the plan or schedule;
- view updating in deductive or relational databases, where the abducible predicates correspond to the extensional facts, and the non-abducible ones to the intensional or view predicates, and where the user can specify desired updates at the level of view predicates, and the abductive solver then proposes possible realizations at the level of the extensional predicates. This setting has actually been illustrated in the examples, though one could also imagine retractions of facts at the level of view predicates.

Finally, let us still stress that the abductive logic programming setting dealt with in this section forms a special type of theory revision problem. Furthermore, Algo. 7.2 is closely related to the MODEL INFERENCE SYSTEM, in which the only generalization operator is the one that adds specific facts, and in which the oracle always answers positively (and the MODEL INFERENCE SYSTEM's eager strategy is used); cf. Sect. 7.3 below.

7.3 Shapiro's Theory Revision System

Having introduced abduction, we now focus on the MODEL INFERENCE SYSTEM for theory revision and program synthesis. We first discuss how learners can interact with the user (the so-called oracle), and then provide a more detailed presentation of the MODEL INFERENCE SYSTEM.

7.3.1 Interaction

Abductive reasoning provides us with a very useful tool to deal with the dependencies among the different predicates when revising theories. However, given the vast number of possible applications of the operators, we need a way to guide the search. In principle, one could perform a heuristic search through the space of possibilities but it has turned out to be quite hard to come up with powerful heuristics to search through this enormous space. An easier approach, taken by *interactive* systems, is to guide the search by means of an *oracle*. The oracle is nothing else than the user, who is supposed to answer queries about the truth and falsity of specific logical formulae in the intended interpretation. More formally, one distinguishes:

- *membership* queries, which query the oracle for the truth-value of a specific ground fact in the intended interpretation,
- *existential* queries, which query the oracle for all substitutions θ that make an atom a ground and $a\theta$ true in the intended interpretation; e.g. when presented with the query $\mathsf{father}(\mathsf{X}, \mathsf{soetkin})$, the oracle may answer $\mathsf{X} = \mathsf{luc}$;
- *subset* queries, which query the oracle about whether a certain clause is true in the intended interpretation;
- *equivalence* queries, which ask the oracle whether a particular hypothesis is equivalent to the target theory; the oracle answers yes if that is the case, and presents a counter-example otherwise.

The Model Inference System relies on an oracle for two different tasks. First, its key specialization operation is to remove an incorrect clause from the current theory. This is accomplished by the backtracing algorithm sketched in the next subsection. Second, it uses an oracle to decide whether a candidate clause to be added to the theory covers a positive example.

The Backtracing Algorithm

The first way that the Model Inference System employs an oracle is by identifying incorrect clauses in the current theory, where a clause is *incorrect* if and only if it is false in the intended interpretation.

In Shapiro's Model Inference System, this property is used in the *backtracing* algorithm, Algo. 7.3, which starts from a covered negative example n and identifies an incorrect clause in the present theory. The algorithm first computes the SLD-tree and then analyzes the successful branches of the tree. Each such branch is then analyzed in a bottom-up manner. The algorithm starts by considering the deepest literal that was resolved and consults the oracle to decide whether the literal is true in the interpretation or not. If it is, the search continues with the parent literals. Otherwise, the clause used to resolve q is incorrect and output.

Example 7.15. Consider the theory

$$\begin{array}{ll} \mathsf{normal}(\mathsf{X}) \leftarrow \mathsf{ostrich}(\mathsf{X}) & \mathsf{flies}(\mathsf{X}) \leftarrow \mathsf{bird}(\mathsf{X}), \mathsf{normal}(\mathsf{X}) \\ \mathsf{bird}(\mathsf{X}) \leftarrow \mathsf{ostrich}(\mathsf{X}) & \mathsf{ostrich}(\mathsf{oliver}) \leftarrow \end{array}$$

and the false negative $\mathsf{flies}(\mathsf{oliver})$. The SLD-tree is shown in Fig. 7.2. Assuming that the first clause is incorrect, the backtracing algorithm would ask the oracle for the truth-values of $\mathsf{ostrich}(\mathsf{oliver})$ (true) and $\mathsf{normal}(\mathsf{oliver})$ (false) and return the incorrect clause $\mathsf{normal}(\mathsf{X}) \leftarrow \mathsf{ostrich}(\mathsf{X})$ which was used to resolve $\mathsf{normal}(\mathsf{oliver})$.

Algorithm 7.3 The backtracing algorithm

```
compute the $SLD$-tree for $n$
for all successful branches of the tree do
  initialize $q$ to the last literal resolved upon
  repeat
    if $q$ is true in the intended interpretation then
      $q$ := parent literal of $q$ in branch
    else
      output clause used to resolve $q$
    end if
  until found
end for
```

Exercise 7.16. The backtracing algorithm will in the worst case query the oracle for the truth-value of *all* literals employed in a successful branch. This corresponds to a linear search through the proof. Can you sketch the outline of Shapiro's divide-and-conquer approach? Rather than selecting the next atom on the path from the bottom to the top of the proof, the divide-and-conquer approach focuses on the atom in the middle of the tree.

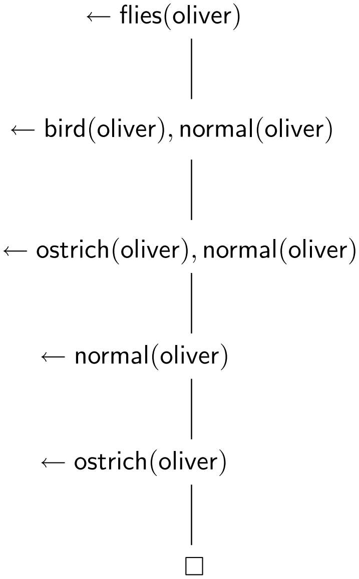

← flies(oliver)
← bird(oliver), normal(oliver)
← ostrich(oliver), normal(oliver)
← normal(oliver)
← ostrich(oliver)
□

Fig. 7.2. Illustrating backtracing.

The backtracing algorithm, to some extent, employs a form of abductive reasoning. Indeed, in the above example, from the negative example flies(oliver) the system abduces that some of the atoms used in the proof may be negative as well, and therefore queries the oracle for their truth-values. This process is continued until enough evidence is accumulated to identify an incorrect clause.

Discovering Missing Clauses

A second reason for employing an oracle in a theory revision system is to guide the search when processing an uncovered positive example. When processing such an example, one must decide whether there are missing clauses for the predicate in the example or whether perhaps some of the underlying predicates are incomplete. To this end, Shapiro's MODEL INFERENCE SYSTEM employs three different strategies, one of which heavily relies on the use of an oracle. To explain the differences, we need to define when a clause *covers* an example *with regard to* a model.

More formally, a clause $p \leftarrow q_1, ..., q_n$ *covers* an example q in the *model* M if and only if

$$mgu(p, q) = \theta \text{ and } \exists \sigma : (q_1\theta, ..., q_n\theta)\sigma \text{ is true in } M \tag{7.4}$$

While learning, the system cannot directly access the intended interpretation, but can consult it indirectly by posing queries to the oracle and using the examples it already knows. This is why covers testing is performed not with regard to the intended intepretation but rather through the use of a different model M. Shapiro distinguishes three different possibilities:

- In the *lazy* strategy, the model M contains only the example set E, and an atom is considered true if it is a known positive example, false if it is a known negative example, and unknown otherwise.
- In the *eager* strategy, the model M corresponds to the intended interpretation. To decide on the truth-value of an atom, the eager strategy first verifies whether it already knows the truth-value of the atom in the example set E, and if it does not, it queries the oracle for the truth-value.
- In the *adaptive* strategy, the model M corresponds to the least Herbrand model of $P \cup \{c\} \cup T$, where P is the set of positive examples, T is the current theory and c is the clause whose coverage is evaluated.

At this point, one can also imagine a fourth possibility, a variant of the *adaptive* strategy, which employs the least Herbrand model of $P \cup T$.

Example 7.17. Suppose one has to decide whether the clause

$$\mathsf{flies(X)} \leftarrow \mathsf{normal(X), bird(X)}$$

covers the example flies(tweety). Using the *lazy* strategy, the example is not covered by the clause unless the facts normal(tweety) and bird(tweety) have already been provided as positive examples. Using the *eager* strategy, the oracle will be queried for the truth-values of $\mathsf{normal(tweety)}$ and $\mathsf{bird(tweety)}$ if they are still unknown. When the *adaptive* strategy is applied, these queries are answered by relying on a theorem prover (such as Prolog) that contains the current theory, the positive examples and the above-listed clause.

Notice the close correspondence between abduction and these forms of coverage testing. Also, for non-recursive theories, the third and fourth strategies always produce the same answers. The use of the clause itself is meant for recursive theories.

Example 7.18. Consider the example $\mathsf{anc(jef,john)}$ in the light of the clause

$$\mathsf{anc(X,Y)} \leftarrow \mathsf{anc(X,Z)}, \mathsf{anc(Z,Y)}$$

and the positive examples $\mathsf{anc(jef,an)}$, $\mathsf{anc(an,mary)}$ and $\mathsf{anc(mary,john)}$. The example is not covered under the fourth strategy, but it is covered under the adaptive strategy.

These different settings are also used by modern inductive logic programming systems. Indeed, the use of an *extensional* background theory by FOIL corresponds to the *lazy* strategy whereas an *intensional* setting corresponds to the *adaptive* strategy or its variant.

7.3.2 The Model Inference System

Now we are able to sketch Shapiro's seminal MODEL INFERENCE SYSTEM. It combines the backtracing algorithm with general-to-specific search for clauses covering the examples, as indicated in Algo. 7.4. The backtracing algorithm is used to locate an incorrect clause c whenever a negative example is covered by the current theory. Incorrect clauses are retracted from the theory and also marked, so that the MODEL INFERENCE SYSTEM will not generate them again.[2] This implies that the theory specialization operator only retracts incorrect clauses. When a positive example is not covered, the MODEL INFERENCE SYSTEM performs a complete general-to-specific search for a clause covering the positive example. During this search, the system never considers marked clauses again. Furthermore, during this search process the system needs to test whether a candidate clause covers the example in the model M, which is defined by the employed strategy (as discussed in the previous section). Observe that, given the complete search, an optimal refinement operator at the level of clauses should be used. Note also that the theory generalization operator merely adds clauses to the current theory.

Example 7.19. We illustrate the MODEL INFERENCE SYSTEM using a simple session to induce the definition of the $\mathsf{member/2}$ predicate (reproduced

[2] At this point, one can easily imagine more efficient ways of realizing this than marking all clauses.

Algorithm 7.4 The Model Inference System

```
$T := \emptyset$ (the theory)
repeat
   read in the next example $e$
   repeat
      while $T$ covers a negative example $n$ do
         identify an incorrect clause $c \in T$
                  (using the contradiction backtracing algorithm)
         retract $c$ from $T$
         mark $c$
      end while
      while $T$ does not cover a positive example $p$ do
         find an unmarked clause $c$ that covers $p$ in $M$
         assert $c$ in $T$
      end while
   until $T$ is complete and consistent with all known examples
until satisfied
```

and adapted from [Shapiro, 1983], excerpt from pages 104–109, ©1983 Massachusetts Institute of Technology, by permission of The MIT Press);[3] the eager strategy is employed and the language bias specifies that the first argument of member is an element and the second is a list.

?-mis.

Next fact? member(a, [b, a]) *is true.*

Checking facts ... member(a, [b, a]) *is not entailed.*

MIS discovers that the reason for this is that there is no clause covering the example (according to Eq. 7.4) and decides to search for one.

Searching for a clause that covers member(a, [b, a]) ←*...*

Checking member(X, [Y|Z]) ←

Found clause member(X, [Y|Z]) ←

Checking facts ... no error found

Next fact? member(a, [b, c]) *is false.*

Fact member(a, [b, c]) *is entailed ... locating an incorrect clause*

Clause member(X, [Y|Z]) ← *is incorrect and retracted*

Checking facts ... member(a, [b, a]) *is not entailed.*

Searching for a clause that covers member(a, [b, a]) ←*...*

Checking member(X, [Y|Z]) ← *... refuted*

Checking member(X, [Y, Z|U]) ← *... refuted*

Checking member(X, [Y|Z]) ← member(Y, Z)

Query: is member(b, [a]) *true? no.*

If the answer were yes, the original example would be covered by the clause.

Checking member(X, [Y|Z]) ← member(X, Z)

[3] For understandability, the order of the queries and candidate clauses has been slightly modified.

Query: is member(a, [a]) *true? yes.*
Found clause member(X, [Y|Z]) ← member(X, Z)
Checking facts ... member(a, [b, a]) *is not entailed.*
Query: is member(a, []) *true? no.*
This query is asked to verify whether member(a, [a]) is covered by the clause already found.
Searching for a clause covering member(a, [a])*...*
Checking member(X, [X|Z]) ← ...
Found clause member(X, [X|Z]) ←
Current definition of member/2*:*
    member(X, [X|Z]) ←
    member(X, [Y|Z]) ← member(X, Z)
Checking facts ... no error found

The next, much more involved session of synthesizing the definition of insertion sort (isort/2, reproduced and adapted from [Shapiro, 1983], excerpt from pages 166–173, ©1983 Massachusetts Institute of Technology, by permission of The MIT Press), using the Model Inference System is included in this book because it very well illustrates the difficulties and combinatorics encountered when synthesizing programs from examples. Even though, according to the typical Prolog programmer, programming insertion sort is extremely easy (not to say trivial), many clauses are considered and many queries have to be posed to the oracle before converging on a solution. Again, the *eager* search strategy is employed. The example can be skipped without loss of continuity.

Example 7.20. * In the intended interpretation, isort/2 succeeds when the second argument is the sorted list corresponding to the first argument. The target program is:

```
isort([], []) ←
isort([X|Y], Z) ← isort(Y, V), insert(X, V, Z)

insert(X, [], [X]) ←
insert(X, [Y|Z], [X, Y|Z]) ← X ⩽ Y
insert(X, [Y|Z], [Y|V]) ← X > Y, insert(X, Z, V)
```

The predicate insert/3 has three arguments: the first is an element, and the other two are (sorted) lists. The predicate succeeds when the third argument is the second argument expanded by inserting the first argument in the appropriate position.

The Model Inference System employs a strong bias. We will assume that it knows the types of the predicates, the modes (the first argument of isort/2 is the input argument, the second is the output argument; the first two arguments of insert/3 are input arguments, the third is an output argument), that isort/2 and insert/3 are functional (that is, the output arguments are a function of the input arguments, which implies that for each input argument, there is exactly one output argument for which the predicate succeeds; cf. also Sect. 6.6.2), and that isort/2 employs the (unknown) predicate insert/3, which in turn employs $\leqslant$.

?-mis.

Next fact? isort([2, 3, 1], [1, 2, 3]) *is true.*

Checking facts ... isort([2, 3, 1], [1, 2, 3]) *is not entailed.*

MIS discovers that the reason for this is that there is no clause covering the example (according to Eq. 7.4) and decides to search for one.

Searching for a clause that covers isort([2, 3, 1], [1, 2, 3]) ←*...*

Query: is isort([3, 1], [1, 2, 3]) *true? no.*

Query: is insert(2, [3, 1], [1, 2, 3]) *true? no.*

Query: is isort([1], [1, 2, 3]) *true? no.*

Query: is insert(2, [1], [1, 2, 3]) *true? no.*

Query: is insert(3, [1], [1, 2, 3]) *true? no.*

Query: for which values of X *is* isort([3, 1], X) *true? X = [1,3].*

Query: is insert(2, [1, 3], [1, 2, 3]) *true? yes.*

Found clause isort([X|Y], Z) ← isort(Y, V), insert(X, V, Z)

Checking facts ... isort([2, 3, 1], [1, 2, 3]) *is not entailed.*

Query: for which values of X *is* isort([1], X) *true? X =[1].*

Query: for which values of X *is* isort([], X) *true? X =[].*

Searching for a clause that covers isort([], [])*...*

Found clause isort([], []) ←*...*

Current hypothesis for isort/2*:*

isort([], []) ←
isort([X|Y], Z) ← isort(Y, V), insert(X, V, Z)

Checking facts ... isort([2, 3, 1], [1, 2, 3]) *is not entailed.*

Query: is insert(1, [], [1]) *true? yes.*

Searching for a clause that covers insert(1, [], [1]) *...*

Found clause insert(X, Y, [X|Y]) ← (an incorrect clause consistent with the current facts)

Checking facts ... isort([2, 3, 1], [1, 2, 3]) *is not entailed.*

Query: is insert(3, [1], [1, 3]) *true? yes.*

Searching for a clause that covers insert(3, [1], [1, 3])*...*

Query: is insert(3, [], [1, 3]) *true? no.*

Query: is insert(1, [], [1, 3]) *true? no.*

Found clause insert(X, [Y|Z], [Y, X|Z]) ←

Current hypothesis for insert/3*:*

insert(X, Y, [X|Y]) ←
insert(X, [Y|Z], [Y, X|Z]) ←

Checking facts ... no error found

At this point, all examples known to MIS are handled correctly. The user, however, discovers a mistake and enters it.

Next fact? isort([2, 3, 1], [2, 3, 1]) *is false.*

```
Fact isort([2, 3, 1], [2, 3, 1]) is entailed ... locating an incorrect clause
Query: is insert(3, [1], [3, 1]) true? no.
Clause insert(X, Y, [X|Y]) ← is incorrect and retracted
Current hypothesis for insert/3:
        insert(X, [Y|Z], [Y, X|Z]) ←
Checking facts ... isort([2, 3, 1], [1, 2, 3]) is not entailed.
Searching for a clause that covers insert(1, [], [1])...
Found clause insert(X, [], [X]) ←.
Checking facts ... no error found
Next fact? isort([2, 1, 3], [3, 2, 1]) is false.
Fact isort([2, 1, 3], [3, 2, 1]) is entailed ... locating an incorrect clause
Query: is isort([3], [3]) true? yes.
Query: is isort([1, 3], [3, 1]) true? no.
Query: is insert(1, [3], [3, 1]) true? no.
Clause insert(X, [Y|Z], [Y, X|Z]) ← is incorrect and retracted
Current hypothesis for insert/3:
        insert(X, [], [X]) ←.
Checking facts ... isort([2, 3, 1], [1, 2, 3]) is not entailed
Searching for a clause covering insert(3, [1], [1, 3])...
Query: is insert(3, [], [3]) true? yes.
Query: is insert(1, [], [3]) true? no.
Found clause insert(X, [Y|Z], [Y, X|Z]) ← Y ⩽ X
Current hypothesis for insert/3:
        insert(X, [], [X]) ←.
        insert(X, [Y|Z], [Y, X|Z]) ← Y ⩽ X
Checking facts ... no error found.
Next fact? isort([2, 1, 3], [1, 2, 3]) is true.
Checking facts ... isort([2, 1, 3], [3, 2, 1]) is not entailed.
Query: for which values of X is isort([1, 3], X) true ? X =[1,3].
Query: for which values of X is isort([3], X) true ? X =[3].
Query: is insert(1, [3], [1, 3]) true ? yes.
Searching for a clause covering insert(1, [3], [1, 3])
Found clause insert(X, [Y|Z], [X, Y|Z]) ← X ⩽ Y
Current hypothesis for insert/3:
        insert(X, [], [X]) ←
        insert(X, [Y|Z], [Y, X|Z]) ← Y ⩽ X
        insert(X, [Y|Z], [X, Y|Z]) ← X ⩽ Y
Checking facts ... no error found
Next fact? isort([3, 2, 1], [1, 3, 2]) is false.
Fact isort([3, 2, 1], [1, 3, 2]) is entailed ... locating an incorrect clause
Query: is isort([2, 1], [1, 2]) true? yes.
Query: is insert(3, [1, 2], [1, 3, 2]) true? no.
Clause insert(X, [Y|Z], [Y, X|Z]) ← Y ⩽ X is incorrect and retracted
Current hypothesis for insert/3:
        insert(X, [], [X]) ←
```

$\mathsf{insert(X, [Y|Z], [X, Y|Z])} \leftarrow \mathsf{X} \leqslant \mathsf{Y}$

Checking facts ... $\mathsf{isort([2, 3, 1], [1, 2, 3])}$ *is not entailed.*
Searching for a clause that covers $\mathsf{insert(3, [1], [1, 3])}$*...*
Query: is $\mathsf{insert(3, [], [])}$ *true? no.*
Query: is $\mathsf{insert(1, [], [])}$ *true? no.*
Found clause $\mathsf{insert(X, [Y|Z], [Y|V])} \leftarrow \mathsf{insert(X, Z, V), Y} \leqslant \mathsf{X}$
Current hypothesis for $\mathsf{insert/3}$*:*

$\mathsf{insert(X, [], [X])} \leftarrow$
$\mathsf{insert(X, [Y|Z], [X, Y|Z])} \leftarrow \mathsf{X} \leqslant \mathsf{Y}$
$\mathsf{insert(X, [Y|Z], [Y|V])} \leftarrow \mathsf{insert(X, Z, V), Y} \leqslant \mathsf{X}$

At this point the program is correct in the intended interpretation and the session terminates.

Shapiro has proven that, under certain conditions (for certain strategies), the Model Inference System *identifies* the target interpretation in the *limit*. Systems that identify an interpretation I in the limit are presented with a potentially infinite sequence of examples $e_1, ..., e_n, ...$ (with regard to I) and have to output a sequence $T_1, ..., T_n, ...$ of theories such that T_i is consistent with the first i examples $e_1, ..., e_i$. A system identifies interpretations in the limit if and only if for all possible interpretations I and all possible sequences of examples (with regard to I, in which each example eventually occurs), there is a number i such that $M(T_i) = I$ and $\forall j > i : T_j = T_i$. So, systems that identify interpretations in the limit converge in a finite number of steps upon a theory that is correct. Identification in the limit forms one framework for studying the convergence properties of logical learning systems. It is further discussed in Chapter 10 on computational learning theory.

7.4 Two Propositional Theory Revision Systems*

The previous section introduced the seminal Model Inference System that is able to infer arbitrary logic programs from examples of their input-output behavior and queries to the user. Generality and expressiveness, however, come at a computational price. As the example sessions illustrated, many questions had to be answered by the user before the system would converge upon the correct theory. Therefore, the present section introduces two simpler alternative systems for revising propositional theories from examples.

7.4.1 Learning a Propositional Horn Theory Efficiently

We first introduce the Horn algorithm by Angluin et al. [1992], which is efficient in that it runs in polynomial time, and hence poses only a polynomial number of questions to the user.[4] At the same time, it learns from *inter-*

[4] More precisely, let m be the number of clauses in the target theory, and n be the number of predicates. Then Algo. 7.5 learns a Horn theory that is logically equiva-

pretations rather than from *entailment* and uses a specific-to-general search strategy rather than a general-to-specific one.

The algorithm poses two types of queries. Equivalence queries ask the oracle whether a given Horn theory is correct or not. The oracle answers yes if it is, and provides a counter-example in the form of an interpretation if it is not. Membership queries simply ask whether a particular interpretation is a positive example or a negative one.

Algorithm 7.5 The HORN algorithm of Angluin et al. [1992]

```
$S := \emptyset$ (a sequence of negative examples)
$H := \emptyset$
while query equivalent($H$) does not return yes do
  Let $x$ be the counter-example returned by the equivalence query
  if $x$ violates at least a clause $c \in H$ then
    ($x$ is a positive example)
    remove all such clauses from $H$
  else
    ($x$ is a negative example)
    for each $s_i \in S$ such that $body(s_i \cap x) \neq body(s_i)$ do
      query member($s_i \cap x$)
    end for
    if any of these queries is answered negatively then
      $i := min\{j|\mathbf{member}(s_j \cap x) = no\}$
      $s_i := s_i \cap x$
    else
      add $x$ as the last element in the sequence $S$
    end if
    $H := \cup_{s \in S} clauses(s)$ where
          $clauses(s) = \{t \leftarrow s | t$ is a predicate or false and $t \notin s\}$
    remove from $H$ all clauses that violate a positive example.
  end if
end while
```

The algorithm is summarized in Algo. 7.5. One observation that is exploited by HORN is that a negative example x must violate one of the clauses in $clauses(s)$.

Example 7.21. Assume that the theory involves the propositional predicates flies, bird and normal, and that {bird, normal} is a negative example. Then the example violates at least one of the $clauses$({bird, normal}), that is,

$$\text{flies} \leftarrow \text{bird}, \text{normal}$$
$$\text{false} \leftarrow \text{bird}, \text{normal}$$

lent to the unknown target theory in polynomial time using $O(m^2n^2)$ equivalence queries and $O(m^2n)$ membership queries.

or their generalizations.

Given that HORN searches from specific to general, it starts to search this space of possibilities at the most specific of these, that is, at $clauses(s)$. During the learning process, HORN keeps track of an ordered sequence of negative examples S. Each new negative example x is used to refine the theory. There are two ways of realizing this. First, HORN attempts to generalize a negative example using one of the examples s already in S. For each such generalization $x \cap s$, HORN queries the user to find out whether the generalized example is negative or not. If it is negative, the example can safely be generalized (and replaced in S). If it is positive, generalization is not allowed as otherwise a positive example $(x \cap s)$ would become violated. Second, if the previous attempts to generalize x fail, the new example is added to the end of S. Finally, after processing a negative example, the clauses on H must be recomputed from S. Positive examples are only used to elmininate clauses from H that are incorrect. These are clauses that violate one of the positive examples already encountered.

Observe that the idea of generalization by intersection corresponds to computing a kind of least general generalization of two negative examples. This idea in combination with the removal of clauses (and candidates) that violate the dual examples (the positives) resembles the inductive logic programming system GOLEM [Muggleton and Feng, 1992]. However, whereas GOLEM computes the least general generalization of positives and removes the clauses covering negatives, HORN takes the converse approach.

Exercise 7.22. Explain why GOLEM and HORN perform dual operations.

A session with HORN is shown in the next example.

Example 7.23. Let the target theory be

$$\begin{array}{l}\mathsf{flies} \leftarrow \mathsf{bird}, \mathsf{normal}\\ \mathsf{normal} \leftarrow \mathsf{bird}, \mathsf{strongwings}\end{array}$$

?-Horn.

Query: is $\emptyset$ *equivalent to the target theory?*

No. Counter-example: $\{\mathsf{bird}, \mathsf{strongwings}, \mathsf{normal}\}$ *(negative).*

This is a negative example, covered by the current theory $H = \emptyset$, but violated by the first clause of the target theory.

At this point becomes $S = [\{\mathsf{bird}, \mathsf{strongwings}, \mathsf{normal}\}]$

Query: is

$$\begin{array}{l}\mathsf{false} \leftarrow \mathsf{bird}, \mathsf{strongwings}, \mathsf{normal}\\ \mathsf{flies} \leftarrow \mathsf{bird}, \mathsf{strongwings}, \mathsf{normal}\end{array}$$

equivalent to the target theory?

No. Counter-example: $\{\mathsf{bird}, \mathsf{strongwings}, \mathsf{normal}, \mathsf{flies}\}$ *(positive).*

Retracting $\mathsf{false} \leftarrow \mathsf{bird}, \mathsf{strongwings}, \mathsf{normal}$

Query: is

flies ← bird, strongwings, normal

equivalent to the target theory?

No. Counter-example: {bird, strongwings, flies} *(negative).*

Query: is {bird, strongwings} *positive? No.*

At this point $S = [\{$bird, strongwings$\}]$

Query: is

flies ← bird, strongwings

normal ← bird, strongwings

equivalent to the target theory?

(The clause false ← bird, strongwings is eliminated because it violates the positive example.)

No. Counter-example: {bird, normal} *(negative).*

Query: is {bird} *positive? Yes.*

At this point $S = [\{$bird, strongwings$\}, \{$bird, normal$\}]$

Query: is

flies ← bird, strongwings

normal ← bird, strongwings

flies ← bird, normal

strongwings ← bird, normal

equivalent to the target theory?

No. Counter-example: {bird, normal, flies} *(positive).*

Retracting strongwings ← bird, normal

Query: is

flies ← bird, strongwings

normal ← bird, strongwings

flies ← bird, normal

equivalent to the target theory? Yes.

Exercise 7.24. * Show that HORN asks at most $O(m^2n^2)$ equivalence queries and $O(m^2n)$ membership queries.

Exercise 7.25. **(hard) Angluin et al. [1992] also describe a more efficient variant of their algorithm. Instead of keeping track of S and H, they keep track of so-called meta-clauses. For instance, in the above example, instead of having two clauses for the negative example {bird, strongwings} they employ the notation flies, normal ← bird, strongwings and perform operations on these meta-clauses directly. Can you define this variant of this algorithm?

Angluin et al. [1992] have also shown that one needs equivalence as well as membership queries to obtain a polynomial-time algorithm, and in later work, Frazier and Pitt [1993] have developed a variant of HORN that learns from entailment.

Exercise 7.26. **(very hard) Can you derive a variant of HORN that learns from entailment?

7.4.2 Heuristic Search in Theory Revision

So far, this chapter has focused on theory revision algorithms, such as the MODEL INFERENCE SYSTEM, that perform a complete search through the space of possible solutions. In most realistic situations, this will be infeasible. Therefore, many theory revision systems resort to heuristics for guiding the search and also employ incomplete operators.

Rather than providing a detailed survey of such methods, let us illustrate this type of theory revision system using the DUCE algorithm of Muggleton [1987].[5] Even though DUCE is a propositional system, it illustrates many important issues such as inverse resolution, predicate invention and compression, another reason for presenting it.

DUCE starts from a theory T in the form of propositional definite clauses, and revises it in a theory T' such that T' *compresses* T as much as possible. During the search it has access to an oracle that can answer subset queries and name newly invented predicates.

The operators employed by DUCE are essentially the propositional versions of the inverse resolution operators introduced in Sect. 5.8, that is, absorption, identification, and intra- and inter-construction. In addition, a propositional variant of the *lgg* operator called *truncation* (as well as an operator to deal with negation) are employed. This results in a vast number of possible operations on any given theory. These operators are scored using the resulting compression in the size of the theory. The *size* of a propositional theory is simply the number of atoms that occur in it. So, the search process carried out by DUCE attempts to determine the operation that results in the largest reduction in size of the theory.

For each of the operators employed by DUCE, the result in compression can be computed using a formula in R and I, where $R \subset T$ is the set of clauses to which the operator is applied and I is the common subset of the atoms in the bodies of the rules in R. For instance, for the absorption operator, the resulting reduction in size is $(\mid R \mid -1) \times (\mid I \mid -1)$.

Example 7.27. Consider the theory (equal to R)

$$\begin{array}{l} \mathsf{primate} \leftarrow \mathsf{twoLegs}, \mathsf{noWings} \\ \mathsf{man} \leftarrow \mathsf{twoLegs}, \mathsf{noWings}, \mathsf{notHairy}, \mathsf{noTail} \\ \mathsf{gorilla} \leftarrow \mathsf{twoLegs}, \mathsf{noWings}, \mathsf{hairy}, \mathsf{noTail}, \mathsf{black} \end{array}$$

of size 14. Applying absorption yields

$$\begin{array}{l} \mathsf{primate} \leftarrow \mathsf{twoLegs}, \mathsf{noWings} \\ \mathsf{man} \leftarrow \mathsf{primate}, \mathsf{notHairy}, \mathsf{noTail} \\ \mathsf{gorilla} \leftarrow \mathsf{primate}, \mathsf{hairy}, \mathsf{noTail}, \mathsf{black} \end{array}$$

As the reader may want to verify, this results in a reduction of 2.

[5] A first-order upgrade of DUCE is the system CIGOL [Muggleton and Buntine, 1988].

Exercise 7.28. Provide the formulae for the other operators.

Duce employs a best-first search through the space of possible operator applications guided by compression. To reduce the search space, Duce only applies an operator on a rule set R and I when the operator yields a local maximum. Furthermore, the new clauses that result from the application of an operator are presented to the oracle before the operator is applied. A session with Duce (adapted from [Muggleton, 1987]) is shown in Ex. 7.29.

Example 7.29. Assume that Duce starts from the theory

blackbird ← beak, black, twoLegs, tail, wings
chimp ← brown, hairy, twoLegs, tail, noWings
eagle ← beak, golden, twoLegs, tail, wings
elephant ← grey, fourLegs, big, tail, trunk, noWings
elephant ← grey, fourLegs, small, tail, trunk, noWings
falcon ← beak, brown, twoLegs, big, tail, wings
gorilla ← black, hairy, twoLegs, noTail, noWings
lemur ← grey, twoLegs, tail, noWings
man ← brown, notHairy, twoLegs, big, noTail, noWings
man ← pink, notHairy, twoLegs, small, noTail, noWings
sparrow ← beak, brown, twoLegs, small, tail, wings

Duce thens proceed as follows:

!-induce
Is elephant ← fourLegs *valid (truncation [12])? no.*
Is elephant ← fourLegs, noWings *valid (truncation [11])? no.*
Is elephant ← fourLegs, trunk *valid (truncation [11))? yes.*
Is man ← notHairy, twoLegs, noTail, noWings *valid (truncation [9])? yes.*
What shall I call p *where* p ← twoLegs, noWings *(inter-con [1])?* primate.
What shall I call q *where* q ← beak, twoLegs, tail, wings *(inter-con [7])?* bird.
No further transformations possible.

Exercise 7.30. What is the theory resulting from the operations in the previous example?

7.5 Inducing Constraints

Constraints are not only useful to constrain the search for theories. They are also interesting statements about the domain of discourse that may be useful for other purposes. For instance, when designing databases, integrity constraints may be specified to guarantee the integrity of the database. The database management system should then take care that the database never enter a state that violates the integrity constraints. A natural question that arises in this context is whether one can also induce such integrity constraints? This question has received quite some attention in the data mining literature, and various algorithms have been developed for inducing a rich variety of integrity constraints from databases. In this section, we show how integrity constraints in the form of general clauses can be induced from a database (in the form of an interpretation). Furthermore, it will be shown how this general framework can be instantiated to infer other forms of integrity constraints such as functional, inclusion and multi-valued dependencies.

7.5.1 Problem Specification

The problem of inducing integrity constraints can be specified as follows.

Given:

- an interpretation D (representing the database)
- a finite language of range-restricted clauses $\mathcal{L}$

Find: a *complete* set of clauses $I \subseteq \mathcal{L}$ that cover the interpretation D.

Recall that a clause is *range-restricted* whenever all variables appearing in its head also appear in its body. A set of clauses is *complete* with regard to $\mathcal{L}$ and D if and only if there is no clause $c \in \mathcal{L}$ such that

$$covers(c, D) \text{ and } I \not\models c \tag{7.5}$$

Thus integrity constraints are clauses, and the database satisfies a clause if the interpretation (representing the database) is a model for the clause; otherwise, the databases violates the clause. Furthermore, one is looking for a *complete* set of clauses, that is, all clauses that cover the database must be either in I or entailed by I.

Example 7.31. Let the language $\mathcal{L}$ contain all clauses containing one variable and no constants over the predicates $\mathsf{human/1}$, $\mathsf{male1/1}$ and $\mathsf{female/1}$, and let the database D be the interpretation

$$\{\mathsf{male(luc)}, \mathsf{female(lieve)}, \mathsf{human(lieve)}, \mathsf{human(luc)}\}.$$

Then the following set of clauses constitutes a complete solution to the integrity constraint induction problem:

$$\begin{array}{ll} \leftarrow \mathsf{female(X)}, \mathsf{male(X)} & \mathsf{human(X)} \leftarrow \mathsf{male(X)} \\ \mathsf{female(X)}; \mathsf{male(X)} \leftarrow \mathsf{human(X)} & \mathsf{human(X)} \leftarrow \mathsf{female(X)} \end{array}$$

Even though other clauses in $\mathcal{L}$ are satisfied by the database, the theory is complete because all other clauses (such as $\leftarrow \mathsf{female(X)}, \mathsf{male(X)}, \mathsf{human(X)}$) are entailed by the theory.

The integrity theory induced in the above example is *minimal*. An integrity theory I is *minimal* if and only if

$$\forall c \in I : I - \{c\} \not\models c \tag{7.6}$$

Minimal theories are often desired because they do not contain redundant clauses. Observe also that it is important that $\mathcal{L}$ be finite. Otherwise, the induced theory might be infinite as well.

7.5.2 An Algorithm for Inducing Integrity Constraints

The key insight that leads to an algorithm is that when a clause $p_1; ...; p_m \leftarrow q_1, ..., q_n$ is violated by a database there must be a substitution θ for which $q_1\theta, ..., q_n\theta$ holds in D but $p_1\theta; ...; p_m\theta$ does not hold. Such a substitution can be computed by answering the query $\leftarrow q_1, ..., q_n, not(p_1; ...; p_m)$ using Prolog on the database D. (This procedure is only sound for range-restricted clauses.)

Example 7.32. Consider the database

$$\{\mathsf{human(luc)},\ \mathsf{human(lieve)},\ \mathsf{bat(dracula)},$$
$$\mathsf{male(luc)},\ \mathsf{male(dracula)},\ \mathsf{female(lieve)}\}.$$

Then the clause

$$\mathsf{human(X)} \leftarrow \mathsf{male(X)}$$

is violated for the substitution $\theta = \{\mathsf{X/dracula}\}$.

If a clause c violates a database for a substitution θ, one can refine it by either modifying $body(c)$ so that it no longer succeeds for θ or by modifying $head(c)$ so that it no longer fails for θ. Thus the clause can be refined by either specializing the body or generalizing the head, for instance, by adding literals to the head or the body. Instead of realizing this using two different operations, it is convenient to merely apply a refinement operator under θ-subsumption to the clause c.

Example 7.33. Some of the refinements of the violated clause

$$\leftarrow \mathsf{male(X)}$$

include

$$\begin{array}{ll}
\mathsf{human(X)} \leftarrow \mathsf{male(X)} & \mathsf{bat(X)} \leftarrow \mathsf{male(X)} \\
\mathsf{female(X)} \leftarrow \mathsf{male(X)} & \leftarrow \mathsf{male(X)}, \mathsf{female(X)} \\
\leftarrow \mathsf{male(X)}, \mathsf{bat(X)} & \leftarrow \mathsf{male(X)}, \mathsf{human(X)}.
\end{array}$$

So, from a θ-subsumption perspective, a specialization operator is applied on the clause. However, recall that the view of θ-subsumption discussed in Chapter 5 was based on learning from entailment. Since we are working with interpretations here, the specialization operator of θ-subsumption effectively generalizes the clause. This is also what is needed because the clause is overly specific as it does not cover the positive interpretation and, hence, must be generalized.

Given that our aim is to find a complete theory, we can easily adapt the generic algorithm (Algo. 3.10) presented earlier. This algorithm closely corresponds to the clausal discovery engine presented by De Raedt and Bruynooghe [1993].

Algorithm 7.6 Discovering integrity constraints

```
Queue := {$\square$};
$I$ := $\emptyset$;
while Queue $\neq \emptyset$ do
    Delete $c$ from Queue;
    if $c$ covers $D$ then
        add $c$ to $I$
    else
        Queue := Queue $\cup \rho_o(c)$
    end if
end while
return $I$
```

The algorithm starts from the most specific element (the empty clause $\square$ when learning from interpretations) and repeatedly generalizes it using an *optimal* refinement operator under θ-subsumption for the language $\mathcal{L}$. Whenever it encounters a clause that is satisfied by the database, it is added to the integrity theory I. Those clauses that are not satisfied are further refined.

Observe also that various search strategies are possible in the clausal discovery algorithm presented. Two natural ones include breadth-first and iterative deepening [De Raedt and Bruynooghe, 1993].

It is easy to see that the clausal discovery algorithm finds a complete integrity theory. However, it is not necessarily a minimal one. Some redundant clauses can be eliminated by verifying whether $I \models c$ before adding the clause c to the integrity theory. This requires however the use of a first-order theorem prover (such as SATCHMO [Manthey and Bry, 1988] or PTTP [Stickel, 1988]). Furthermore, this by itself does not suffice to guarantee that a minimal integrity theory is found.

Exercise 7.34. Can you provide an example that shows that non-minimal theories may be found even when using the above procedure to eliminate redundant clauses ? (Hint: this depends on the order in which the clauses are considered by the algorithm).

To guarantee a minimal theory, one must include a post-processing phase in which the clauses have to be processed in the order in which they were found. For each such clause, one can then check whether it is redundant with regard to the discovered theory. If it is, it can be deleted before going to the next clause.

A further optimization is possible when the language $\mathcal{L}$ is *anti-monotonic* [De Raedt and Bruynooghe, 1993]. A language is *anti-monotonic* if and only if

$$\forall c \in \mathcal{L} : (s \ \theta\text{-subsumes } c) \rightarrow s \in \mathcal{L} \tag{7.7}$$

Anti-monotonicity in this context requires that all generalizations of all clauses in $\mathcal{L}$ also belong to $\mathcal{L}$. For anti-monotonic languages, one can safely prune away refinements that are not reduced w.r.t. the data while still retaining a complete solution.

A refinement $c \cup \{l\}$ of a clause c is not *reduced* with regard to the data D if and only if $vars(l) \subseteq vars(c)$ and

$$\forall \theta : c\theta \text{ is violated if and only if } c \cup \{l\} \text{ is violated in } D \tag{7.8}$$

Example 7.35. Consider the database of Ex. 7.31. The refinement

$$\leftarrow \mathsf{male(X), human(X)}$$

of

$$\leftarrow \mathsf{male(X)}$$

is not reduced because the substitutions ($\mathsf{X = luc}$) that violate the first query also violate the second clause. Therefore, the two clauses are equivalent w.r.t. the database D. Formulated otherwise, the two clauses are equivalent because

$$\mathsf{human(X) \leftarrow male(X)}$$

covers the database. Because the language is anti-monotonic, it is safe to prune away the clause

$$\leftarrow \mathsf{male(X), human(X)}$$

because for all refinements of this clause, for instance,

$$\mathsf{tall(X) \leftarrow male(X), human(X)},$$

there will be an equivalent clause in $\mathcal{L}$ that does not contain the redundant literal $\mathsf{human(X)}$ (in our example: $\mathsf{tall(X) \leftarrow male(X)}$). Therefore, when the language is anti-monotonic it is safe to prune non-reduced refinements.

The algorithm equipped with a breadth-first search strategy and the pruning for redundant clauses is illustrated in the next example and Fig. 7.3.

Example 7.36. Let us now illustrate the clausal discovery algorithm of Algo. 7.5.2 at work on Ex. 7.31. We assume that the algorithm performs a breadth-first search. The algorithm starts at the clause $\leftarrow$, discovers that it does not cover the database and refines it by applying an optimal refinement operator. This leads to the clauses

$$\leftarrow \mathsf{m} \qquad \leftarrow \mathsf{f} \qquad \leftarrow \mathsf{h}$$

(we abbreviate atoms such as $\mathsf{male(X)}$ to m), which in turn cover the example and are therefore refined as well. Of the resulting clauses, the clauses

$$\leftarrow \mathsf{m}, \mathsf{f} \qquad \mathsf{h} \leftarrow \mathsf{m} \qquad \mathsf{h} \leftarrow \mathsf{f}$$

cover the example and are output; the clauses

$$\leftarrow \mathsf{m}, \mathsf{h} \qquad \leftarrow \mathsf{f}, \mathsf{h} \qquad \mathsf{m} \leftarrow \mathsf{f} \qquad \mathsf{f} \leftarrow \mathsf{m}$$

are not reduced. The only clauses that remain for refinement are

$$\mathsf{f} \leftarrow \mathsf{h} \qquad \mathsf{m} \leftarrow \mathsf{h}.$$

Given that we work with an optimal refinement operator, the only remaining refinement is

$$\mathsf{m}; \mathsf{f} \leftarrow \mathsf{h}$$

which forms a solution as well.

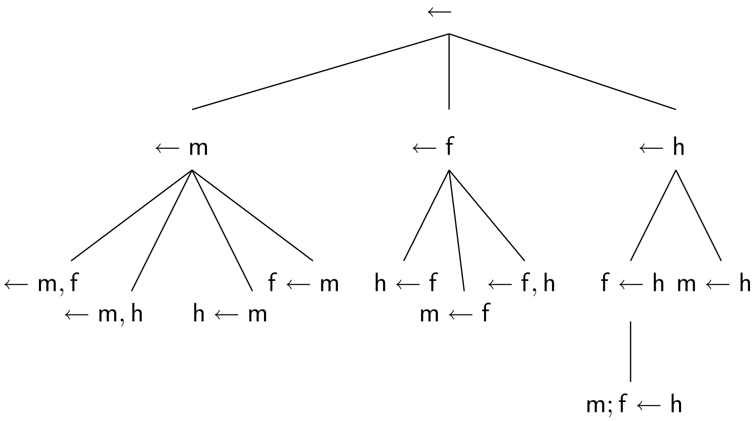


Fig. 7.3. Illustrating clausal discovery

The algorithm can easily be adapted to obtain some popular forms of integrity constraints in relational databases such as inclusion, functional and multi-valued dependencies. They play an important role in database design and normalization [Elmasri and Navathe, 1989].

Let us start by showing how functional dependencies can be induced on an example due to Flach [1993].

Functional dependencies are clauses of the form

$$\overline{Y} = \overline{Z} \leftarrow r(\overline{X}, \overline{Y}, \overline{W_1}), r(\overline{X}, \overline{Z}, \overline{W_2}) \tag{7.9}$$

where r denotes a relation and $\overline{X}, \overline{Y}$ and $\overline{Z}$ denote vectors of variables.

Example 7.37. Consider the database for the relation $\mathsf{train}(\mathsf{From},\mathsf{Hour},\mathsf{Min},\mathsf{To})$ that denotes that there is a train from From to To at time $\mathsf{Hour},\mathsf{Min}$:

$\mathsf{train}(\mathsf{utrecht},8,8,\mathsf{denbosch}) \leftarrow$
$\mathsf{train}(\mathsf{maastricht},8,10,\mathsf{weert}) \leftarrow$
$\mathsf{train}(\mathsf{utrecht},9,8,\mathsf{denbosch}) \leftarrow$
$\mathsf{train}(\mathsf{maastricht},9,10,\mathsf{weert}) \leftarrow$
$\mathsf{train}(\mathsf{utrecht},8,13,\mathsf{eindhoven}) \leftarrow$
$\mathsf{train}(\mathsf{utrecht},8,43,\mathsf{eindhoven}) \leftarrow$
$\mathsf{train}(\mathsf{utrecht},9,13,\mathsf{eindhoven}) \leftarrow$
$\mathsf{train}(\mathsf{utrecht},9,43,\mathsf{eindhoven}) \leftarrow$
$\mathsf{train}(\mathsf{tilburg},8,10,\mathsf{tilburg}) \leftarrow$
$\mathsf{train}(\mathsf{utrecht},8,25,\mathsf{denbosch}) \leftarrow$
$\mathsf{train}(\mathsf{tilburg},9,10,\mathsf{tilburg}) \leftarrow$
$\mathsf{train}(\mathsf{utrecht},9,25,\mathsf{denbosch}) \leftarrow$
$\mathsf{train}(\mathsf{tilburg},8,17,\mathsf{eindhoven}) \leftarrow$
$\mathsf{train}(\mathsf{tilburg},8,47,\mathsf{eindhoven}) \leftarrow$
$\mathsf{train}(\mathsf{tilburg},9,17,\mathsf{eindhoven}) \leftarrow$
$\mathsf{train}(\mathsf{tilburg},9,47,\mathsf{eindhoven}) \leftarrow$

Then the clause

$$\mathsf{From1} = \mathsf{From2} \leftarrow \mathsf{train}(\mathsf{From1},\mathsf{Hour1},\mathsf{Min},\mathsf{To}),\mathsf{train}(\mathsf{From2},\mathsf{Hour2},\mathsf{Min},\mathsf{To})$$

is a satisfied functional dependency. Using the vector notation, $\overline{X}$ represents $[\mathsf{To},\mathsf{Min}]$, $\overline{Y}$ represents $\mathsf{From1}$, $\overline{Z}$ represents $\mathsf{From2}$, $\overline{W_1}$ represents $\mathsf{Hour1}$, and $\overline{W_2}$ represents $\mathsf{Hour2}$. A further clause that is satisfied by this database is:

$$\mathsf{To1} = \mathsf{To2} \leftarrow \mathsf{train}(\mathsf{From},\mathsf{Hour1},\mathsf{Min},\mathsf{To1}),\mathsf{train}(\mathsf{From},\mathsf{Hour2},\mathsf{Min},\mathsf{To2})$$

Exercise 7.38. Are there any other constraints satisfied by the train database?

Functional dependencies can be discovered using the clausal discovery algorithm outlined above. One only needs to adapt the language $\mathcal{L}$ and the refinement operator to generate clauses that correspond to functional dependencies. However, instead of using the rather complex clausal notation to denote functional dependencies, it is much more convenient to work on the traditional database notation. In this notation, the first dependency would be represented as: $\mathsf{Min},\mathsf{To} \rightarrow \mathsf{From}$. Also, when inducing functional dependencies, complete and minimal integrity theories are preferred.

For completeness, we also define *multi-valued* and *inclusion* dependencies. They can be discovered in a similar fashion. *Multi-valued dependencies* are clauses of the form

$$r(\overline{X},\overline{Y_1},\overline{Z_2}) \leftarrow r(\overline{X},\overline{Y_1},\overline{Z_1}),r(\overline{X},\overline{Y_2},\overline{Z_2}) \tag{7.10}$$

where r denotes a relation and we again work with vectors of variables. Again, one can simplify the notation and write this as $\mathsf{X} \rightsquigarrow \mathsf{Y}$ where X and Y represent the attributes used in the relation.

Inclusion dependencies are clauses of the form $\mathsf{r.X} \subseteq \mathsf{s.Y}$ where $\mathsf{r.X}$ and $\mathsf{s.Y}$ represent the attributes used in the relations r and s. They basically state that every value that appears in $\mathsf{r.X}$ must also appear in $\mathsf{s.Y}$.

7.6 Conclusions

In this chapter we have investigated the complications that arise when learning or revising theories. Different types of theories and settings have been considered. First, we have looked into abductive logic programming, where possible revisions are restricted to the addition of facts. We have also seen that integrity constraints in the form of general clauses provide us with a powerful and declarative means to incorporate additional background knowledge into the induction process and to restrict the space of possible solutions. Second, we have considered the problem of inferring a theory or model by interacting with an oracle. The same types of techniques are applicable to program synthesis from examples. Third, we have also used two propositional learners, that is, Horn [Angluin et al., 1992] and Duce [Muggleton, 1987], to illustrate computational complexity issues in both theory revision and heuristic search. Finally, we have concluded the chapter by investigating how integrity theories can be induced from data.

Throughout the chapter, several new principles for logical and relational learning have been introduced. These include the use of abductive reasoning for inferring missing facts that explain particular examples, the use of operators at the level of theories, the use of an oracle to support interaction, and the use and the induction of integrity constraints.

7.7 Bibliographic Notes

The induction, completion and revision of theories were studied extensively in the 1980s and the early 1990s. Research started with the Model Inference System of Shapiro [1983] and the Marvin system of Sammut and Banerji [1986]. During that early inductive logic programming period, a lot of attention was also devoted to program synthesis from examples, a closely related topic, as programs can be regarded as theories. In this context of program synthesis, the seminal Model Inference System by Ehud Shapiro is important as it was the first to realize program synthesis from examples in a logic programming context. Other, more recent approaches to program synthesis are discussed in [Bergadano and Gunetti, 1995] and an overview of logic program synthesis and transformation is given by Flener and Yilmaz [1999]. Other theory revision systems that were influential at the time include Mobal [Morik et al., 1993] and Clint [De Raedt, 1992, De Raedt and Bruynooghe, 1994], which focused on applications in knowledge acquisition, Cigol [Muggleton and Buntine, 1988], an upgrade of Duce [Muggleton, 1987] to first-order logic, and Forte [Richards and Mooney, 1995], a heuristic theory revision system not relying on an oracle.

The efficient propositional theory revision algorithm Horn was developed by Angluin et al. [1992] from a computational learning theory perspective; cf. also Sect. 10.2. Horn learns a propositional Horn theory efficiently using equivalence and subset queries. A detailed account of the use of queries for learning is given in [Angluin, 1987]. Various researchers have also tried to upgrade Horn to a proper inductive logic programming setting with interesting theoretical and practical results; cf. [Arias and Khardon, 2002, Khardon, 1999, Reddy and Tadepalli, 1998] and Sect. 10.2. The combination of learning and reasoning was studied by Khardon and Roth [1997].

The introduction of integrity constraints in theory revision systems is due to De Raedt and Bruynooghe [1992b]. It was motivated by an analogy between the view-updating problem in deductive databases (briefly sketched at the end of Sect. 7.2.3).

Abduction has fascinated researchers since the introduction of the term by the philosopher Charles S. Peirce [Hartshorne and Weiss, 1965]. In a computational logic context, early works include [Poole, 1988, Shanahan, 1989] and overviews are given by [Kakas et al., 1992, 1998]. The relationship among induction and abduction is discussed in [Flach and Kakas, 2000], and a useful view in an inductive logic programming context is presented in [Adé and Denecker, 1995].

The induction of integrity constraints in various forms has been studied by various researchers in the data mining community, for instance [Mannila and Raiha, 1992, Flach and Savnik, 1999]. Within an inductive logic programming setting it was studied by Helft [1989], De Raedt and Bruynooghe [1993], De Raedt and Dehaspe [1997] and Flach and Lachiche [2001].

8

Probabilistic Logic Learning

So far, this book has focused on logical aspects of learning and on learning in logic. Logic has, however, severe limitations for representing and reasoning about uncertainty. Therefore, this chapter adds another dimension to logic and learning: probability. Probability theory provides us with an elegant and formal basis for reasoning about uncertainty. In the past few decades, several probabilistic knowledge representation formalisms have been developed to cope with uncertainty, and many of these formalisms can be learned from data. Unfortunately, most such formalisms are propositional, and hence they suffer from the same limitations as traditional propositional learning systems. This chapter studies how to alleviate these problems by introducing probabilistic logic learning, *sometimes also called* probabilistic inductive logic programming *or* statistical relational learning. *This is a newly emerging area in artificial intelligence lying at the intersection of reasoning about uncertainty, machine learning and knowledge representation. Our presentation of probabilistic logic learning starts from an inductive logic programming perspective and extends logical learning with probabilistic methods. In this regard, it follows the* upgrading approach *of Chapter 6. The chapter will also show how the frameworks of learning from interpretations and learning from entailment can be upgraded to a probabilistic context, yielding representational formalisms such as stochastic logic programs (SLPs) [Eisele, 1994, Muggleton, 1996, Cussens, 2000], PRISM [Sato, 1995] and ICL [Poole, 1993b], Markov logic networks [Richardson and Domingos, 2006], and Bayesian logic programs [Kersting and De Raedt, 2007].*

While the chapter provides a brief review of probabilistic models, it does not contain an elaborate overview of probabilistic methods in artificial intelligence, as this by itself is an important topic of study. Such overviews can be found, for instance, in [Russell and Norvig, 2004, Jensen, 2001, Cowell et al., 1999, Baldi et al., 2003]. At the same time, the chapter is not intended as a complete survey of statistical relational learning (see [De Raedt and Kersting, 2003, Getoor and Taskar, 2007, De Raedt et al., 2008]), but rather focuses on the principles that underlie this new and exciting subfield of artificial intelligence.

8.1 Probability Theory Review

This section briefly reviews some key concepts from probability theory that are used within artificial intelligence. The reader not familiar with this topic is encouraged to consult [Russell and Norvig, 2004] for an excellent overview from an artificial intelligence perspective. Our notation closely corresponds to that of Russell and Norvig [2004], and the introduction to Bayesian networks in Sect. 8.2.1 summarizes that by Russell and Norvig [2004].

Let X be a random variable over a domain $D(X)$. Because we will combine probability theory with logic, we will focus on *logical* random variables in this book; these variables have as domain $\{true, false\}$. The notation $\mathbf{P}(X)$ will be used to denote the probability distribution of the random variable, and $P(X = x)$ or $P(x)$ to denote the probability that the random variable X takes the value x. For instance, $\mathbf{P}(\mathsf{fever})$ denotes the probability distribution over the random variable (or proposition) fever, and $P(\mathsf{fever} = false)$ (or, in shorthand notation, $P(\neg\mathsf{fever})$) denotes the probability that $\mathsf{fever} = false$.

We also need to define the *joint probability distribution* $\mathbf{P}(X_1, \cdots, X_n)$ over a set of random variables $\{X_1, \cdots, X_n\}$. This allows one to represent the joint probability that $\mathsf{fever} = true$ and $\mathsf{headache} = true$ at the same time. Generalizing the above notation, we will use the notations $\mathbf{P}(\mathsf{fever}, \mathsf{headache})$ and $P(\mathsf{fever} = true, \mathsf{headache} = true)$, respectively.

Some useful definitions and properties of probability theory can now be listed:

- *Conditional probability*:

$$\mathbf{P}(X|Y) = \frac{\mathbf{P}(X,Y)}{\mathbf{P}(Y)} \tag{8.1}$$

The use of $\mathbf{P}$ in equalities is a shorthand notation that denotes that the equality is valid *for all* possible values of the involved random variables. Indeed, if X and Y are logical random variables, the above equality actually denotes:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)} \tag{8.2}$$

$$P(X|\neg Y) = \frac{P(X,\neg Y)}{P(\neg Y)} \tag{8.3}$$

$$P(\neg X|\neg Y) = \frac{P(\neg X,\neg Y)}{P(\neg Y)} \tag{8.4}$$

$$P(\neg X|Y) = \frac{P(\neg X,Y)}{P(Y)} \tag{8.5}$$

Observe that, due to the division, it is required that $\mathbf{P}(Y) \neq \mathbf{0}$, that is, $\forall y \in dom(Y) : P(y) \neq 0$.

- The *chain rule*: :

$$\mathbf{P}(X_1, \cdots, X_n) = \mathbf{P}(X_1) \prod_{i=2}^{n} \mathbf{P}(X_i | X_{i-1}, \cdots, X_1) \tag{8.6}$$

- *The law of Bayes (assuming* $\mathbf{P}(Y) \neq \mathbf{0}$*)*:

$$\mathbf{P}(X|Y) = \frac{\mathbf{P}(Y|X)\ \mathbf{P}(X)}{\mathbf{P}(Y)} \tag{8.7}$$

- *Marginalization*:

$$\mathbf{P}(X) = \sum_{y \in dom(Y)} \mathbf{P}(X, y) \tag{8.8}$$

  or, alternatively, *conditioning* (combining marginalization and the chain rule):

$$\mathbf{P}(X) = \sum_{y \in dom(Y)} \mathbf{P}(X|y)\ \mathbf{P}(y) \tag{8.9}$$

- *Conditional independence*: the (sets of) variables X and Y are conditionally independendent given Z (assuming $\mathbf{P}(Y) \neq \mathbf{0}$) if and only if

$$\mathbf{P}(X, Y|Z) = \mathbf{P}(X|Z)\ \mathbf{P}(Y|Z) \tag{8.10}$$

  or equivalently, if and only if

$$\mathbf{P}(X|Y, Z) = \mathbf{P}(X|Z). \tag{8.11}$$

  This is sometimes denoted as $(X \perp Y|Z)$. Intuitively, Eq. 8.11 states that when X and Y are conditionally independent given Z, knowing the value of Y does not influence (that is provide any information about) the value of X, provided one already knows the value of Z.
  If the conditional independency property holds without needing to condition on Z, we say that X and Y are *marginally independent*, notation $(X \perp Y)$. Obviously, when X and Y are conditionally independent given Z, then Y and X are also conditionally independent given Z, more formally $(X \perp Y|Z) \leftrightarrow (Y \perp X|Z)$. Thus the conditional independence relation is symmetric.

8.2 Probabilistic Logics

Within logic, two types of semantics can be distinguished: a *model-theoretic* one and a *proof-theoretic* one. A similar distinction is made when learning logical representations, where learning from interpretations corresponds to the model-theoretic perspective, and learning from entailment corresponds to a proof-theoretic one. This distinction also applies to probabilistic knowledge representation formalisms. Indeed, Bayesian networks essentially define a probability distribution over *interpretations* or *possible worlds*, and stochastic grammars define a distribution over *proofs*, derivations or traces. We will therefore study these two basic frameworks (as well as some variants) as they form the basis for contemparary statistical relational learning.

A key distinction between logical learning and statistical learning is the form of coverage relation employed. The covers relation between the language of examples and of hypotheses in a logical framework was introduced in Chapter 4. It states when an example is covered by a hypothesis and it yields the value either true or false. On the other hand, when working with probabilistic models, the coverage relation becomes a probabilistic one. Rather than stating in absolute terms whether the example is covered or not, a probability will be assigned to the example being covered. This corresponds to a *probabilistic* coverage relation. The logical coverage relation can be re-expressed as a probabilistic one by stating that the probability is 1 or 0 of being covered. This distinction will be further elaborated on.

The terminology employed is also different. In particular, when working with probaiblistic representations, the term *model* is used to refer to a particular hypothesis. This is not to be confused with models referring to interpretations or possible worlds in a logical sense.

8.2.1 Probabilities on Interpretations

In this section, we first introduce Bayesian networks, while closely following Russell and Norvig [2004], and then study a well-known variant of this framework: Markov networks.

Bayesian Networks

The most popular formalism for defining probabilities on possible worlds is that of Bayesian networks. As an example of a Bayesian network, consider the network that is graphically illustrated in Figure 8.1 (similar to the famous alarm network of Pearl [1988] used by Russell and Norvig [2004]). Formally speaking, a *Bayesian network* is

- an augmented, directed acyclic graph,
- where each node corresponds to a random variable X_i,
- where each edge indicates a direct influence among the random variables, and
- where there is a *conditional probability distribution (CPD)* associated with each node.

The conditional probability distribution $\mathit{cpd}(X)$ associated with the node X is defined in terms of the parents of the node X, which we denote by $\mathbf{Pa}(X)$. It specifies $\mathit{cpd}(X) = \mathbf{P}(X \mid \mathbf{Pa}(X))$.

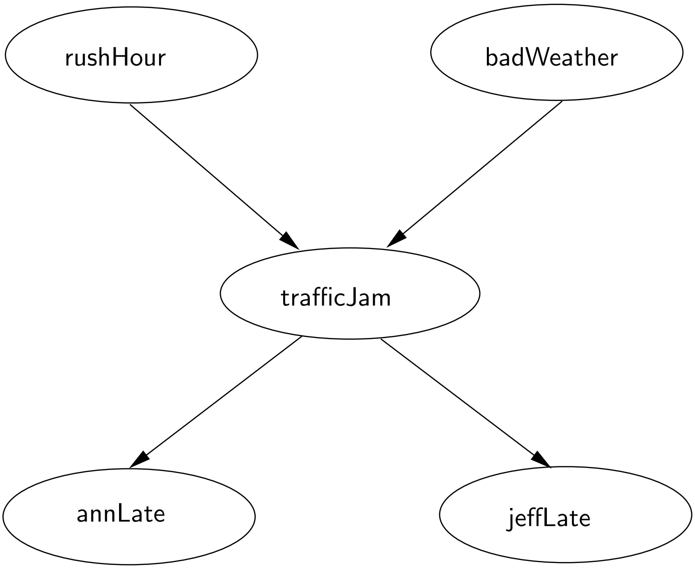


Fig. 8.1. A Bayesian network

$\mathbf{P}$(badWeather)
$(0.10, 0.90)$

$\mathbf{P}$(rushHour)
$(0.25, 0.75)$

badWeather	rushHour	$\mathbf{P}$(trafficJam)
true	*true*	$(0.95, 0.05)$
true	*false*	$(0.70, 0.30)$
false	*true*	$(0.40, 0.60)$
false	*false*	$(0.05, 0.95)$

trafficJam	$\mathbf{P}$(jeffLate)
true	$(0.60, 0.40)$
false	$(0.05, 0.95)$

trafficJam	$\mathbf{P}$(annLate)
true	$(0.70, 0.30)$
false	$(0.01, 0.99)$

Table 8.1. The conditional probability distributions associated with the nodes in the traffic network; cf. Figure 8.1. The distributions are specified over $\{true, false\}$

Example 8.1. Consider the Bayesian network in Fig. 8.1. It contains the random variables rushHour, badWeather, annLate, jeffLate and trafficJam. The CPDs associated with each of the nodes are specified in Table 8.1. They include the CPDs $\mathbf{P}(\mathsf{trafficJam} \mid \mathsf{badWeather}, \mathsf{rushHour})$, $\mathbf{P}(\mathsf{badWeather})$, etc.

The Bayesian network thus has two components: a *qualitative* one, the directed acyclic graph, and a *quantitative* one, the conditional probability distributions. Together they specify the joint probability distribution $\mathbf{P}(X_1, ..., X_n)$ over a fixed and finite set of random variables $\{X_1, \ldots, X_n\}$. For simplicity, we assume that all random variables have the domain $\{true, false\}$; this actually amounts to specifying a probability distribution on the set of all possible interpretations. Indeed, in our traffic example, the Bayesian network defines a probability distribution over truth assignments to {trafficJam, rushHour, annLate, jeffLate, badWeather}.

Intuitively, an arc from node X to node Y in the directed acyclic graph indicates that X influences Y. In the traffic example, having bad weather or during rush hour the probability that there is a traffic jam increases. Also, if there is a traffic jam, Jeff and Ann are likely to be late. More formally, the qualitative component of a Bayesian network specifies the following set of conditional independence assumptions (sometimes called the *local Markov assumptions*):

Assumption 1 *(cf. Russell and Norvig [2004]) Each node X_i in the graph is conditionally independent of any subset* $\mathbf{A}$ *of nodes that are* non-descendants *of X_i given a joint state of $\mathbf{Pa}(X_i)$, that is,* $\mathbf{P}(X_i \,|\, \mathbf{A}, \mathbf{Pa}(X_i)) = \mathbf{P}(X_i \,|\, \mathbf{Pa}(X_i))$.

For example, annLate is conditionally independent of badWeather given its parent trafficJam, notation (annLate⊥badWeather|trafficJam). Intuitively, it states that knowledge of the value of badWeather is not influencing, that is, not providing more information about the value of annLate if one already knows the value of trafficJam. Because of the local Markov assumptions, we can write down the joint probability density as

$$\mathbf{P}(X_1, \ldots, X_n) = \prod_{i=1}^{n} \mathbf{P}(X_i \mid \mathbf{Pa}(X_i)) \tag{8.12}$$

by applying the independency assumption and the chain rule to the joint probability distribution. Let us illustrate this derivation on our example:

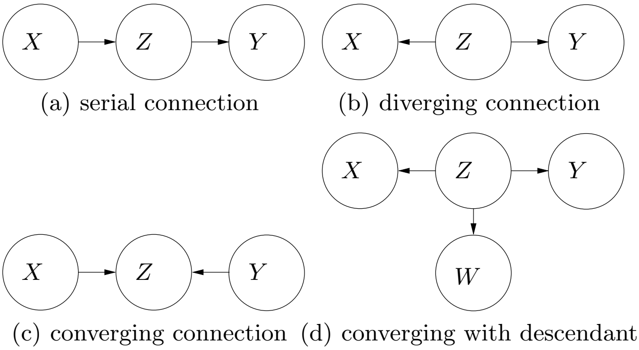
$$\begin{aligned}
\mathbf{P}(B, R, T, J, A) &= \mathbf{P}(A|B, R, T, J)\ \mathbf{P}(B, R, T, J)\\
&= \mathbf{P}(A|T)\ \mathbf{P}(B, R, T, J)\ \text{(Assumption 1)}\\
&= \mathbf{P}(A|T)\ \mathbf{P}(J|B, R, T)\ \mathbf{P}(B, R, T)\\
&= \mathbf{P}(A|T)\ \mathbf{P}(J|T)\ \mathbf{P}(T|B, R)\ \mathbf{P}(B, R)\\
&= \mathbf{P}(A|T)\ \mathbf{P}(J|T)\ \mathbf{P}(T|B, R)\ \mathbf{P}(B|R)\ \mathbf{P}(R)\\
&= \mathbf{P}(A|T)\ \mathbf{P}(J|T)\ \mathbf{P}(T|B, R)\ \mathbf{P}(B)\ \mathbf{P}(R) \qquad (8.13)
\end{aligned}$$

In this type of derivation, we have to topologically sort the nodes, that is, to find an order of the nodes such that the index i of all nodes is strictly larger than that of all of its parent nodes. So, we start from the deepest nodes of the directed acyclic graphs and work backwards towards those nodes that do not posses any parents.

To capture the intuitions behind the conditional independence relations that hold in a Bayesian network, it is useful to consider the following special cases, depicted in Fig. 8.2:

- In a *serial connection* $X \rightarrow Z \rightarrow Y$, one can prove that $(X \perp Y | Z)$ because the node Z *blocks* the flow of information from X to Y;
- In a *diverging connection* $X \leftarrow Z \rightarrow Y$, one can also prove that $(X \perp Y | Z)$ and again Z blocks the flow from X to Z.
- A *converging connection* $X \rightarrow Z \leftarrow Y$ for which $(X \perp Y)$ holds but $(X \perp Y | Z)$ does not hold. This is called *explaining away.* This is illustrated in the example below. In a converging connection the flow of information from X to Y is only blocked if Z is not known. Visually this can be interpreted as the information flowing away in the "sink" Z.

Example 8.2. Suppose we know trafficJam = *true*, then knowing that rushHour = *true* makes it less likely that badWeather is also *true*. Therefore, badWeather and rushHour are not conditionally independent given trafficJam. However, if the value of trafficJam (and its descendants) are not known then badWeather and rushHour are marginally independent. Notice that the explaining away argument also holds for descendents of Z (cf. Fig. 8.2d), for instance, in the traffic example, when annLate = *true*.



(a) serial connection (b) diverging connection

(c) converging connection (d) converging with descendant

Fig. 8.2. Various types of connections in a Bayesian network

These findings are also valid when taking into account intermediate nodes. For instance, replacing $X \rightarrow Z \rightarrow Y$ with $X \rightarrow U_1 \rightarrow ... \rightarrow U_k \rightarrow Z \rightarrow V_1 \rightarrow ... \rightarrow V_m \rightarrow Y$ still yields a serial connection with the same properties. In the literature on Bayesian networks, these notions are used in the definition of *d-separation*. Two sets of nodes X and Y are *d-separated* given evidence nodes Z provided that there is no connection from a node in X to one in Y along which information can flow. So, all connections from nodes in X to nodes in Y should be blocked by nodes in Z. The notion of d-separation coincides with that of conditional independence.

Bayesian networks are a popular knowledge representation formalism because they allow one to explicitly encode conditional independency assumptions, which result in a compact representation of the joint probability distribution. This is much more efficient than directly encoding the joint probability distribution in a table, as the traffic network example illustrates.

Exercise 8.3. How many probability values are required to explicitly list the probability of each interpretation for the $\mathsf{traffic}$ problem? How many are required when the Bayesian network is used?

So, Bayesian networks define a probability distribution on the possible worlds, that is, the interpretations. To compute the probability of the interpretation $\{\mathsf{t}, \mathsf{a}, \mathsf{b}\}$, observe that $\neg\mathsf{r}$ and $\neg\mathsf{j}$ hold. Therefore, the probability of this interpretation is

$$\begin{aligned} P(\mathsf{b}, \neg\mathsf{r}, \mathsf{t}, \neg\mathsf{j}, \mathsf{a}) &= P(\mathsf{a}|\mathsf{t})P(\neg\mathsf{j}|\mathsf{t})P(\mathsf{t}|\mathsf{b}, \neg\mathsf{r})P(\mathsf{b})P(\neg\mathsf{r}) && (8.14)\\ &= 0.7 \times 0.4 \times 0.7 \times 0.1 \times 0.75 && (8.15)\\ &= 0.0147 && (8.16)\end{aligned}$$

The resulting probability distribution on possible worlds induces a probability distribution on the propositions and propositional formulae. The probability of a logical formula f is the sum of the probabilities of all interpretations i that are a model of the formula f:

$$P(f) = \sum_{i \models f} P(i). \qquad (8.17)$$

This definition is often applied to *marginalize* variables away. For instance, consider

$$\begin{aligned} P(\mathsf{t} \wedge \mathsf{a}) &= \sum_{i \models \mathsf{t} \wedge \mathsf{a}} P(i) && (8.18)\\ &= \sum_{B \in \{t,f\}} \sum_{R \in \{t,f\}} \sum_{J \in \{t,f\}} P(B, R, \mathsf{t}, J, \mathsf{a}) && (8.19)\end{aligned}$$

where we have abbreviated the variables and states appropriately.

Bayesian networks are typically used to compute a posterior probability of some random variables V given that one knows values for some of the other variables, the so-called *evidence*. For instance, in the traffic example, one might wonder what the posterior probability $P(\mathsf{a}|\mathsf{b})$ is that $\mathsf{annLate}=$ *true* given that there is $\mathsf{badWeather}$. The naive way of computing such probabilities is to apply the definition of posterior probability and then to apply the methods illustrated above. There exist, however, more efficient methods to compute these probabilities, even though Bayesian network inference is, in general, NP-complete. For a more detailed introduction on Bayesian networks, we refer the reader to Russell and Norvig [2004].

The most important points to remember are 1) that Bayesian networks encode a probability distribution over the interpretations of the variables in the network, 2) that this encoding is more efficient than explicitly representing the joint probability distribution, and 3) that the encoding is based on conditional independence assumptions. From a logical perspective, the key limitation of Bayesian networks is that a probability distribution is defined only on *propositional* interpretations. Using Bayesian networks, it is impossible to model relational or logical worlds. This problem is similar to that with traditional propositional mining and learning systems, and it will be alleviated in Sect. 8.4.1, when introducing Bayesian Logic Programs.

Exercise 8.4. Compute $P(\mathsf{t} \vee \mathsf{a})$ for the traffic network. Compute also $P(\mathsf{t} \vee \mathsf{a} | \mathsf{r})$.

Exercise 8.5. List the conditional independence statements for the traffic network.

Markov Networks*

Whereas Bayesian networks represent the joint probability distribution and a set of conditional independency assumptions using directed acyclic graphs, Markov networks (sometimes also called Markov random fields) employ undirected graphs. As in Bayesian networks, the nodes in the graphs represent the random variables and the missing edges encode independencies between the corresponding random variables.

The qualitative component specifies a set of a conditional independence assumptions. More formally, in Markov networks, the following conditional independency assumption is made:

Assumption 2 *Each node X_i in the graph is conditionally independent of any subset* $\mathbf{A}$ *of nodes that are not neighbours of X_i given a joint state of* $\mathbf{Ne}(X_i)$*, that is,* $\mathbf{P}(X_i \mid \mathbf{A}, \mathbf{Ne}(X_i)) = \mathbf{P}(X_i \mid \mathbf{Ne}(X_i))$*. The expression* $\mathbf{Ne}(X_i)$ *refers to the set of all neighbours of X_i.*

It is the equivalent of the local Markov assumptions for Bayesian networks.

A further modification w.r.t. Bayesian networks is that Markov networks associate *potential* functions to *cliques* in the graphs. A *clique* c is a maximal subgraph that is fully connected, that is, has an edge between any two nodes in the subgraph. The set of random variables in the clique c will be denoted by $\{X_{c1}, \cdots, X_{ck_c}\}$. A *potential* function $f(X_1, \cdots, X_m)$ on the random variables $X_1, \cdots, X_m$ is a mapping from the set of states over $X_1, \cdots, X_m$ to the nonnegative real numbers. The key difference between a probability and a potential function is that potentials need not sum to 1. However, higher values of potentials still yield more likely states. Like probability densities, $\mathbf{f}(X_1, \cdots, X_n)$ denotes the potential, and $f(X_1 = x_1, \cdots, X_n = x_n)$ denotes its value for a specific state $(x_1, \cdots, x_n)$.

The joint probability distribution $\mathbf{P}(X_1, \cdots, X_n)$ defined by a Markov network factorizes as

$$\mathbf{P}(X_1, \cdots, X_n) = \frac{1}{Z} \prod_{c:clique} \mathbf{f}_c(X_{c1}, \cdots, X_{ck_c}) \tag{8.20}$$

where Z is a normalization constant needed to obtain a probability distribution (summing to 1). It is defined by

$$Z = \sum_{x_1, \cdots, x_n} \prod_{c:clique} f_c(X_{c1} = x_{c1}, \cdots, X_{ck_c} = x_{ck_c}) \tag{8.21}$$

Using marginalization and conditioning, one can then compute conditional probability distributions.

Example 8.6. An example Markov network is shown in Fig. 8.3. Its cliques are: {trafficJam, rushHour, badWeather}, {trafficJam, annLate}, and {trafficJam, jeffLate}. So, the joint probability distribution factorizes as

$$\mathbf{P}(B, R, T, J, A) = \frac{1}{Z}\mathbf{f}_{T,R,B}(T, R, B)\ \mathbf{f}_{J,T}(J, T)\ \mathbf{f}_{A,T}(A, T) \tag{8.22}$$

It is often convenient to use functionals that are exponentiated weighted sums of features of the variables, that is, use functions of the form

$$f_c(X_{c1}, \cdots, X_{ck_c}) = e^{w_i F_i(X_{c1}, \cdots, X_{ck_c})} \tag{8.23}$$

where the F_i denote features of the state of the $X_{c1}, \cdots, X_{ck_c}$ and the w_i are weights. This results in a so-called *log-linear* model because taking the logarithm of Eq. 8.20 then yields a weighted sum of features, which is easier to handle than a product.

The graphs of Bayesian and Markov nets encode a set of (conditional) independency assumptions. A natural question that arises in this context is whether Markov and Bayesian nets can encode the same set of conditional independency assumptions. The answer to this question is, however, negative. Indeed, the assumptions encoded by the Bayesian net depicted in Fig. 8.2c, the converging connection, cannot be encoded using a Markov net.

We now turn our attention to probabilistic representations that define probability distributions on sequences; they are often used in the natural language processing community [Manning and Schütze, 1999].

8.2.2 Probabilities on Proofs

In this section, we first introduce probabilistic context-free grammars, and then study a closely related framework: Markov models.

Probabilistic Context-Free Grammars

Probabilistic grammars, which have been employed successfully in bioinformatics [Durbin et al., 1998] and natural language processing [Manning and

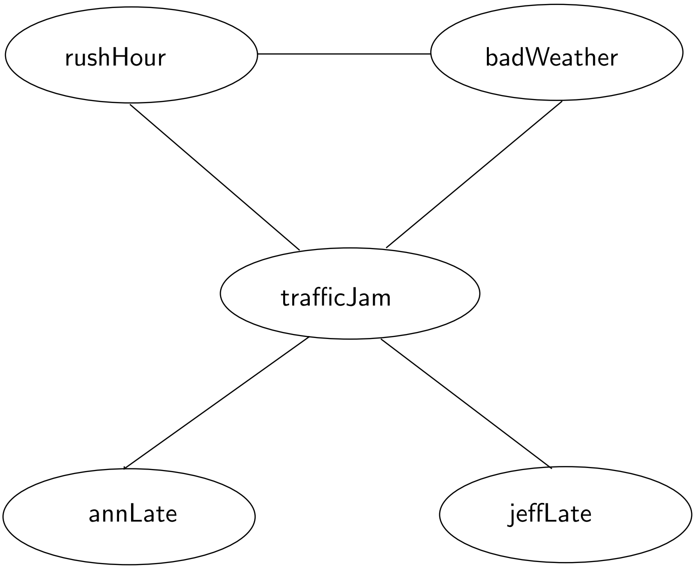


Fig. 8.3. A Markov network

Schütze, 1999], define a probability distribution on proofs. Because there is a close correspondence between derivations in a (context-free) grammar and resolution proofs (cf. Sect. 4.6), probabilistic grammars can provide ideas for probabilistic resolution proofs.

Probabilistic context-free grammars are context-free grammars (cf. Sect. 4.6) where the rules possess a probability label. More formally, whereas the usual rules in a context-free grammar have the form $n \rightarrow s$ (where n is a non-terminal symbol in N and s is a string over $N \cup \Sigma$), the rules in a probabilistic context-free grammars are of the form $p : n \rightarrow s$, where p is a probability value. Furthermore, it is required that

$$\forall n \in N : \sum_{p:n \rightarrow s_i} p = 1 \tag{8.24}$$

that is, the sum of the probabilities associated with rules for each non-terminal symbol n must be equal to 1.

Example 8.7. Consider the following probabilistic context-free grammar, adapted from Allen [1987], where N= $\{\mathsf{S}, \mathsf{NP}, \mathsf{VP}, \mathsf{N}, \mathsf{PP}, \mathsf{V}, \mathsf{P}\}$ and $\{\mathsf{rice}, \mathsf{flies}, \mathsf{like}, \mathsf{sand}, ...\} \subseteq \Sigma$, and where we have omitted many of the terminal symbols for brevity.

1.0 : S → NP, VP.
0.2 : NP→ N, N.
0.8 : NP→ N.
0.6 : VP→ V, NP.
0.4 : VP→ V, PP.
1.0 : PP→ P, NP.

0.01 : N→ rice.
0.01 : N→ flies.
...
0.02 : V→ like.
0.02 : V→ flies.
...
0.05 : P→ like.
...

Observe, for instance, that the sum of the probabilities for the non-terminal VP $= 0.4 + 0.6 = 1$.

We can now employ the rules to define the probability of a derivation (or proof) and the probability of a sentence. The probability $P(d)$ of a derivation d, in which the rules $r_1, ..., r_n$ with associated probability labels $p_1, ..., p_n$ are used $m_1, ..., m_n$ times, is

$$P(d) = \prod_i p_i^{m_i} \tag{8.25}$$

Example 8.8. Consider the following derivation of the sentence "*Rice flies like sand*" with the above-mentioned stochastic grammar. The associated probabilities are listed on the right.

S → NP VP.	1.0
S → N N VP.	0.2
S → Rice N VP.	0.2×0.01
S → Rice flies VP.	$0.2 \times 0.01 \times 0.01$
S → Rice flies V NP.	$0.2 \times 0.01 \times 0.01 \times 0.6$
S → Rice flies like NP.	$0.2 \times 0.01 \times 0.01 \times 0.6 \times 0.02$
S → Rice flies like N.	$0.2 \times 0.01 \times 0.01 \times 0.6 \times 0.02 \times 0.8$
S → Rice flies like sand.	$0.2 \times 0.01 \times 0.01 \times 0.6 \times 0.02 \times 0.8 \times 0.1$

Each time a rule is applied in a proof, the probability of the rule is multiplied with the overall probability. This induces a probability distribution over all derivations for the same non-terminal symbol.[1] The probabilities on derivations can be aggregated to obtain a probability distribution over all sentences $s \in \Sigma^*$ that can be derived from a non-terminal symbol n. More formally,

$$P(n \rightarrow s) = \sum_{d(n \rightarrow s)} P(d(n \rightarrow s)) \tag{8.26}$$

[1] There exist some context-free grammars for which the sum of the probabilities over all the derivations is not equal to 1; see [Prescher, 2003] for more details and a way to repair this.

where $d(n \rightarrow s)$ denotes a derivation of $n \rightarrow s$.

Indeed, consider again the sentence "*Rice flies like sand*" for which there are two derivations, one with probability p_1 (as illustrated above) and another one with probability p_2. The sum of these probabilities specifies the probability that a random sentence generated by the non-terminal symbol S is indeed "*Rice flies like sand*". These distributions can be used for a wide variety of purposes.

For instance, in a natural language processing context, the following problems are typically addressed (cf. [Manning and Schütze, 1999]):

- *Decoding*: computing the probability $P(s)$ of a sentence s.
- *Most likely parse tree*: computing the most likely derivation $d(S \rightarrow s)$ of a particular sentence s:
$$MLD(d) = \arg \max_{d(S \rightarrow s)} P(d(S \rightarrow s)) \tag{8.27}$$
The most likely derivation is useful for disambiguation. For instance, the sentence "*Rice flies like sand*" possesses two parse trees. A natural language understanding system must choose among these different parse trees, and the natural choice is the most likely one.
- *Sampling:* generating a representative sample of sentences at random from the grammar. Sample sentences can be generated by generating derivations at random. Sampling a derivation proceeds by probabilistically selecting at each choice point during the derivation the derivation step according to the probabilities of the rules in the grammar.
- *Learning* the structure or the parameters of the probabilistic-context free grammar, cf. Sect. 8.3.

Exercise 8.9. Formalize the algorithm for sampling derivations from a stochastic context-free grammar and show how it works on an example.

Exercise 8.10. Design an algorithm to find the most likely parse tree of a sentence. Finding a naive algorithm is not so hard (and the purpose of the exercise). However, there exists also a polynomial time algorithm based on dynamic programming; cf. [Manning and Schütze, 1999].

For stochastic context-free grammars there exist efficient (polynomial) algorithms for the key problems (decoding, most likely parse tree, and learning the parameters); cf. [Manning and Schütze, 1999]. However, their key limitation is their expressive power. On the one hand, they only deal with *flat* sequences, that is, sequences of unstructured symbols. On the other hand, context-free grammars (and therefore stochastic context-free grammars) are not Turing equivalent, that is, they cannot be used as a programming language as definite clause logic can, as in Prolog. Therefore, it is useful to upgrade the underlying grammar representation to that of a (definite) logic program, a Turing-equivalent language; cf. [Muggleton, 1996].

Markov Models*

Markov models have numerous applications, most notably in speech recognition [Rabiner, 1989] and bioinformatics [Baldi and Brunak, 1998], where (hidden) Markov models are used, for instance, to determine how likely it is that a given protein belongs to some fold [Durbin et al., 1998]. Markov models can be regarded as a kind of stochastic regular grammar. A stochastic regular grammar is a special case of a stochastic context-free grammar in which all rules have exactly one symbol on the right-hand side. A key difference between Markov models and stochastic regular grammars, however, is that hidden Markov models define a probability distribution over sentences of the same length, whereas in stochastic grammars the distribution is over sentences of variable length. Markov models are also a special type of Bayesian network, which is convenient for introducing them. In a *visible* Markov model, there is a sequence of n variables $X_0, \cdots, X_n$, where the domain of each variable X_i is the same, that is, the set of values they can take are identical. Furthermore, it is assumed that

$$\forall i, j \geqslant 1: \ \mathbf{P}(X_i|X_{i-1}) = \mathbf{P}(X_j|X_{j-1}) \tag{8.28}$$

$$\forall i, j \geqslant 1: \ \mathbf{P}(X_i|X_{i-1}, X_{i-2}, \cdots, X_0) = \mathbf{P}(X_i|X_{i-1}). \tag{8.29}$$

These assumptions imply that one only needs two sets of parameters, one that models the *prior* $\mathbf{P}(X_0)$ and one that models the state-transition probabilities $\mathbf{P}(X_i|X_{i-1})$. Because the transition probababilities are the same for all i, Markov models employ *parameter tying* (see below). A Markov model thus corresponds to the type of Bayesian network shown in Fig. 8.4 and the Markov model factorizes as

$$\mathbf{P}(X_0, \cdots, X_n) = \mathbf{P}(X_0) \prod_{i>0} \mathbf{P}(X_i|X_{i-1}) \tag{8.30}$$

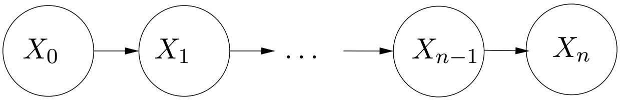


Fig. 8.4. Visible Markov model structure

In a *hidden* Markov model, there are additional variables $Y_0, \cdots, Y_n$ that depend only on the X_i. Furthermore, it is assumed that

$$\forall i, j \geqslant 0: \mathbf{P}(Y_i|X_i) = \mathbf{P}(Y_j|X_j) \tag{8.31}$$

and that the model has the structure indicated in Fig. 8.5. Thus the hidden Markov model factorizes as

$$\mathbf{P}(Y_0, \cdots, Y_n, X_0, \cdots, X_n) = \mathbf{P}(X_0)\mathbf{P}(Y_0|X_0)\prod_{i>0}^{n}\mathbf{P}(Y_i|X_i)\mathbf{P}(X_i|X_{i-1}) \tag{8.32}$$

In hidden Markov models, the X_i denote the hidden states and are typically not observed, the Y_i correspond to the observations.

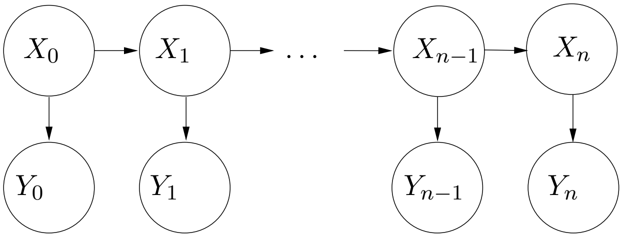


Fig. 8.5. Hidden Markov model structure

Hidden Markov models are extremely popular in many application domains such as bioinformatics and natural language processing. One of the reasons is that efficient inference procedures (based on dynamic programming) exist for the following central tasks:

- *Decoding:* computing the probability $\mathbf{P}(Y_0 = y_0, \cdots, Y_n = y_n)$ of a particular observation sequence.
- *Most likely state sequence:* computing the most likely hidden state sequence corresponding to a particular observation sequence, that is,

$$\arg\max_{x_0,\cdots,x_n} P(X_0 = x_0, \cdots, X_n = x_n|Y_0 = y_o, \cdots, Y_n = y_n) \tag{8.33}$$

  This is realized using the well-known Viterbi algorithm; cf. Rabiner [1989].
- *State prediction:* computing the most likely state based on a particular observation sequence, that is,

$$\arg\max_{x_t} P(X_t = x_t, \cdots, X_n = x_n|Y_0 = y_o, \cdots, Y_n = y_n) \tag{8.34}$$

  If $t < n$, this is called *smoothing*; if $t = n$, it is called *filtering*, and if $t > n$, it is called *prediction* (of the future states based on the past).
- *Parameter estimation*, a form of learning which will be dealt with in Sect. 8.3.1.

There also exist many variants of hidden Markov models, such as input-output HMMs [Bengio and Frasconi, 1994] and conditional random fields [Lafferty et al., 2001].

8.3 Probabilistic Learning

In this section, we briefly review some important probabilistic learning principles. While doing so, we concentrate on the representation of Bayesian networks. Nevertheless, these principles and techniques also apply to the other representations introduced in the previous section. Such adaptations will be discussed extensively when dealing with probabilistic logic learning techniques in Sect. 8.5.

One typically distinguishes two problems within the statistical learning community. First, there is the problem of *parameter estimation*, where the goal is to estimate appropriate values for the parameters of a model, whose structure is fixed, and second, there is the problem of *structure learning*, where the learner must infer both the structure and the parameters of the model from data.

8.3.1 Parameter Estimation

The problem of *parameter estimation* can be formalized as follows:

Given

- a set of examples E,
- a probabilistic model $M = (S, \lambda)$ with structure S and parameters λ,
- a probabilistic coverage relation $P(e|M)$ that computes the probabilty of observing the example e given the model M,
- a scoring function $score(E, M)$ that employs the probabilistic coverage relation $P(e|M)$

Find the parameters λ^* that maximize $score(E, M)$, that is,

$$\lambda^* = \arg\max_{\lambda} score(E, (S, \lambda)) \tag{8.35}$$

This problem specification abstract the specific class of models considered, and actually can be instantiated w.r.t. the different representations introduced above. However, rather than going into the details of specific representations (and corresponding algorithmic optimizations), we will focus on the underlying principles of parameter estimation. The problem specification shows that parameter estimation is essentially an optimization problem that depends on the scoring function and type of model employed.

The standard scoring function is the probability of the model or hypothesis given the data. This yields the *maximum a posteriori* hypothesis H_{MAP}

$$H_{MAP} = \arg\max_{H} P(H|E) = \arg\max_{H} \frac{P(E|H) \times P(H)}{P(E)} \tag{8.36}$$

It can be simplified into the *maximum likelihood* hypothesis H_{ML} by applying Bayes' law and assuming that all hypotheses H are, a priori, equally likely, yielding:

$$H_{ML} = \arg\max_{H} P(E|H) \tag{8.37}$$

The likelihood function shown in Eq. 8.37 will be employed throughout the rest of the chapter. The reason is that the maximum aposteriori approach is more complicated as a prior on the hypotheses must be specified and be taken into account during the learning. This corresponds to the *bayesian* approach where the prior is viewed as a kind of background knowledge. For an introduction to this type of approach, the reader may want to consult [Russell and Norvig, 2004, Bishop, 2006].

It is typically assumed that the examples are independently and identically distributed (i.i.d.), which allows one to rewrite the expression in the following form (where the e_i correspond to the different examples):

$$H_{ML} = \arg\max_{H} \prod_{e_i \in E} P(e_i|H) \tag{8.38}$$

Notice that at this point, the probabilistic coverage relation $P(e|H)$ is employed. It indicates the likelihood of observing e given the hypothesis H.

Typically, the goal is to learn a *generative* model, that is, a model that could have generated the data. As an example of such a model, consider a stochastic context-free grammar. To learn such a grammar, one employs *possible* examples e, for which $P(e|H) > 0$, that is, sentences with non-zero probability. Examples that are impossible, that is, for which $P(e|H) = 0$, are typically not used. This contrasts with the traditional inductive logic programming setting, which is *discriminative*. The positive examples in the traditional inductive logic programming setting have a strictly positive probabilistic coverage, the negative ones have a 0 probabilistic coverage. This observation implies that

$$P(e|H) > 0 \text{ if and only if } \mathbf{c}(e, H) = 1 \tag{8.39}$$

showing that logical coverage *can be* a prerequisite for probabilistic coverage. For instance, in a stochastic context-free grammar, the sentences must be logically covered by (or provable from) the context-free grammar in order to have a non-zero probability. On the other hand, in Bayesian networks, this is not the case.

Nevertheless, within probabilistic learning, discriminative learning has been studied and a wide variety of techniques is available. Within the above problem specification discriminative learning can be modeled by choosing an alternative scoring function. One popular scoring function for discriminative learning maximizes the *conditional likelihood* function

$$H_{CL} = \arg\max_{H} \prod_{e_i \in E} P(class(e_i)|des(e_i), H) \tag{8.40}$$

where the examples e_i are split up into the class of interest $class(e_i)$ and the description of the example $des(e_i)$. Throughout the rest of this chapter, we will concentrate on the generative case.

In the remainder of the section, we study how to address the parameter estimation problem under different assumptions, depending on the type of data and model under consideration. The exposition in the next subsection closely corresponds to that of Russell and Norvig [2004].

Learning from Fully Observable Data

The first case to consider is that where the data are fully observable, that is, where the value of all random variables in the example are completely known. This case is also the easiest one, because, under common assumptions, parameter estimation is reduced to frequency counting. Let us illustrate this on the simple Bayesian network that consists of only the nodes trafficJam and annLate, shown in Fig. 8.6 and Table 8.2. Fully observable for this example means that the example takes the form of a table in attribute-value format as indicated in Table 8.3.

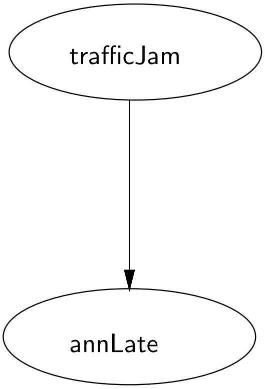


Fig. 8.6. A simple Bayesian network. $P(\mathsf{t}) = \lambda_0; P(\mathsf{a}|\mathsf{t}) = \lambda_1; P(\mathsf{a}|\neg\mathsf{t}) = \lambda_2$

$\mathbf{P}$(trafficJam)
$(\lambda_0, 1-\lambda_0)$

trafficJam	$\mathbf{P}$(annLate)
true	$(\lambda_1, 1-\lambda_1)$
false	$(\lambda_2, 1-\lambda_2)$

Table 8.2. The conditional probability distributions associated with the nodes in the simple traffic network; cf. Figure 8.6. The distributions are specified over $\{true, false\}$

Notice that the likelihood for one example satisfying trafficJam $= true$ and annLate $= false$ is

$$P(\mathsf{t}, \neg\mathsf{a}) = \lambda_0(1-\lambda_1)$$

	trafficJam	annLate
e_1	true	true
e_2	true	true
e_3	false	false
e_4	false	true
...		

Table 8.3. A completely observed data set

Therefore, under the i.i.d. assumption, for n examples, the likelihood of the data can be written as (where we use $|x|$ to denote the number of examples satisfying the logical condition x)

$$\begin{aligned} P(E|M) &= \prod_i^n P(e_i|M) \\ &= \lambda_0^{|t|}\ (1-\lambda_0)^{|\neg t|}\ \lambda_1^{|t \wedge a|}(1-\lambda_1)^{|t \wedge \neg a|}\ \lambda_2^{|\neg t \wedge a|}(1-\lambda_2)^{|\neg t \wedge \neg a|} \end{aligned}$$

This function can be maximized by maximizing the logarithm of the function instead, which is easier as well as justified because the logarithm is a monotonic function. The log likelihood can then be maximized by computing the derivatives, setting them to 0, and solving for the λ_i, yielding the following steps:

$$\begin{aligned} L = \log P(E|M) = &\ |t| \log \lambda_0 + |\neg t| \log(1-\lambda_0) + \\ &\ |t \wedge a| \log \lambda_1 + |t \wedge \neg a| \log(1-\lambda_1) + \\ &\ |\neg t \wedge a| \log \lambda_2 + |\neg t \wedge \neg a| \log(1-\lambda_2) \end{aligned} \tag{8.41}$$

The derivatives are:

$$\frac{\partial L}{\partial \lambda_0} = \frac{|t|}{\lambda_0} - \frac{|\neg t|}{1-\lambda_0} \tag{8.42}$$

$$\frac{\partial L}{\partial \lambda_1} = \frac{|t \wedge a|}{\lambda_1} - \frac{|t \wedge \neg a|}{1-\lambda_1} \tag{8.43}$$

$$\frac{\partial L}{\partial \lambda_2} = \frac{|\neg t \wedge a|}{\lambda_2} - \frac{|\neg t \wedge \neg a|}{1-\lambda_2} \tag{8.44}$$

Setting these equal to 0 and solving for the λ_i yields

$$\lambda_0 = \frac{|t|}{|t| + |\neg t|} \tag{8.45}$$

$$\lambda_1 = \frac{|t \wedge a|}{|t \wedge a| + |t \wedge \neg a|} \tag{8.46}$$

$$\lambda_2 = \frac{|\neg t \wedge a|}{|\neg t \wedge a| + |\neg t \wedge \neg a|} \tag{8.47}$$

which shows that maximum likelihood estimation corresponds to counting, that is, to computing the relative frequencies in the data set.

Although the example has been given in the context of a (simple) Bayesian network, the results are similar for other representational frameworks such as Markov models and probabilistic context-free grammars.

	trafficJam	annLate
e_1	true	true
e_2	true	?
e_3	false	false
e_4	?	true

Table 8.4. A data set with missing values

Expectation Maximization

The Expectation Maximisation (EM) algorithm deals with the case where the data are not fully observable, but only partially observable. To continue our illustration, this corresponds to the type of example illustrated in Table 8.4. If values for some variables are occasionally unobserved, there is *missing data.* If values for some variables are always unobserved, the variables are called *latent.*

We would like to maximize the (log-)likelihood of the data. The log-likelihood is, as before, a function of the parameters λ of the model. Let us call this function $Q(\lambda)$. The difficulty now is that the function $Q(\lambda)$ depends on the unobserved values. A natural way of dealing with these values is to compute the *expected* likelihood function $Q(\lambda)$, where the expectation is taken over the missing values of the examples. Let the examples $e_i = x_i \cup y_i$ be composed of an observed part x_i and an unobserved part y_i. In the expectation step, the expected values of the y_i will be computed. To do this, we need a model. Therefore, the expectation maximization algorithm assumes that there is a current model $M(\lambda_j)$ and uses it to compute the expected values of the y_i, which are then used to compute $Q(\lambda)$. More formally, this yields (where $\mathbf{E}$ stands for the expectation taken with regard to the current model $M(\lambda_j)$ and the missing data y_i, and $L(\lambda)$ for the likelihood as a function of the parameters λ as indicated in Eq. 8.41):

$$\begin{aligned}
Q(\lambda) &= \mathbf{E}[L(\lambda)] \\
&= \mathbf{E}[\log\ P(E|M(\lambda)] \\
&= \mathbf{E}\big(\sum_{e_i \in E} log\ P(e_i|M(\lambda))\big) \\
&= \mathbf{E}\big(\sum_{e_i \in E} log\ P(x_i, y_i|M(\lambda))]\big) \\
&= \sum_{e_i \in E} P(y_i|x_i, M(\lambda_j)) \log\ P(x_i, y_i|M(\lambda)) \qquad (8.48)
\end{aligned}$$

To compute the expected likelihood $Q(\lambda)$, one first needs to compute the $P(y_i|x_i, M(\lambda_j))$. These values are typically computed using the normal inference procedures of the underlying probabilistic model. Doing so corresponds to estimating the likelihood that the examples x_i are completed with the y_i given the current model $M(\lambda_j)$ – the *estimation step*. Once these estimates are known, the expected likelihood of the data, $Q(\lambda)$ can be computed. $Q(\lambda)$ is then used, in the *maximization step*, to determine the maximum likelihood estimators. This step is analogous to that for fully observable data discussed earlier. Thus, the general expectation maximization (EM) algorithm can be summarized as follows:

E-Step: On the basis of the observed data and the present parameters of the model, compute a distribution over all possible completions of each partially observed data case.

M-Step: Using each completion as a fully observed data case weighted by its probability, compute the updated parameter values using (weighted) frequency counting.

The frequencies over the completions are called the *expected counts*. The algorithm is sketched more formally in Algo. 8.1.

Algorithm 8.1 The EM algorithm

```
$j := 0$
initialize $\lambda_j$
**repeat**
    Compute the $P(y_i|x_i, M(\lambda_j))$ for all $i$
    $\lambda_{j+1} := \arg\max_\lambda Q(\lambda) = \arg\max_\lambda \sum_i P(y_i|x_i, M(\lambda_j)).log\ P(x_i|M(\lambda))$
    $j := j + 1$
**until** convergence
```

It can be shown that an EM re-estimation step never decreases the likelihood function, and hence that the EM algorithm must converge to a stationary point of the likelihood function. If this function has a single maximum, the EM algorithm will converge to a global maximum; otherwise it will converge to a local maximum or a saddle point.

Example 8.11. On the example data set shown in Table 8.2, the EM algorithm first completes the data set. The result is that the example e_2 (or e_4) is split into two fractional examples $e_{2,1}$ and $e_{2,2}$. The first has the value true for a and receives a probability (weight) of $P(\mathsf{a}|\mathsf{t})$ and the second has the value false and probability $P(\neg\mathsf{a}|\mathsf{t})$. These fractional examples can then be used to compute the *expected* counts and perform the maximization step. For instance, the parameter λ_0 can be re-estimated as

$$\lambda_0 = \frac{ec(t)}{ec(t) + ec(\neg t)} \tag{8.49}$$

where $ec_\lambda(t)$, the expected count of the number of occurrences of t given the current set of parameters $\lambda = (\lambda_0, \lambda_1, \lambda_2)$, now replaces $|t|$ in Eq. 8.45; that is

$$ec_\lambda(t) = \sum_{i \in \{1,...,4\}} P(t|e_i) = 2 + P(\mathsf{t}|e_4) \tag{8.50}$$

where $P(\mathsf{t}|e_4)$ has to be estimated using λ; that is,

$$\begin{aligned} P(\mathsf{t}|e_4) &= P(\mathsf{t}|\mathsf{a}) \\ &= \frac{P(\mathsf{t} \wedge \mathsf{a})}{P(\mathsf{a})} \\ &= \frac{\lambda_0\lambda_1}{\lambda_0\lambda_1 + (1-\lambda_0)\lambda_2} \end{aligned} \tag{8.51}$$

It must be mentioned that there also exist alternative techniques to cope with missing data. Most notably, there are gradient-based methods that can also be used to optimize the likelihood in the presence of missing values; see [Mitchell, 1997, Russell and Norvig, 2004].

Parameter Tying

One technique that is often applied in probabilistic models is *parameter tying.* The idea is that various parameters of a probabilistic model are tied together, that is, are made identical. We already saw one example of parameter tying. Indeed, when introducing the Markov models, we assumed that the state transition probability distribution was shared amongst all different states over time.

Example 8.12. As an illustration, consider a variant of the traffic network, shown in Fig. 8.7, where we now assume that both Jeff and Ann can be late. Assume also that for this Bayesian network there are additional parameters λ_3 and λ_4 to model $\mathbf{P}(\mathsf{jeffLate}|\mathsf{trafficJam})$, which implies that the likelihood of the examples is

$$\begin{aligned} P(E|M) = {} & \lambda_0^{|t|}(1-\lambda_0)^{|\neg t|}\lambda_1^{|t\wedge a|}(1-\lambda_1)^{|t\wedge\neg a|}\lambda_2^{|\neg t\wedge a|}(1-\lambda_2)^{|\neg t\wedge\neg a|} \\ & \lambda_3^{|t\wedge j|}(1-\lambda_3)^{|t\wedge\neg j|}\lambda_4^{|\neg t\wedge j|}(1-\lambda_4)^{|\neg t\wedge\neg j|} \end{aligned}$$

Parameter tying can be applied to this network, for instance, by setting $\lambda_1 = \lambda_3$ and $\lambda_2 = \lambda_4$. This corresponds to the assumption that the behavior of Ann and Jeff, and hence the underlying probability distributions $\mathbf{P}(J|T) = \mathbf{P}(A|T)$, are the same. Under this assumption the likelihood of the data simplifies to

$$\begin{aligned} P(E|M) = \lambda_0^{|t|}(1-\lambda_0)^{|\neg t|}\lambda_1^{|t\wedge a|+|t\wedge j|}(1-\lambda_1)^{|t\wedge\neg a|+|t\wedge\neg j|} \\ \lambda_2^{|\neg t\wedge a|+|\neg t\wedge j|}(1-\lambda_2)^{|\neg t\wedge\neg a|+|\neg t\wedge\neg j|} \end{aligned}$$

The advantage of parameter tying is that the number of parameters is reduced and the probability distribution is simplified, which in turn makes inference and learning more efficient. Parameter tying is a central concept within statistical relational learning, where one typically has templates (in the form of rules or clauses) that are instantiated in different ways. The idea is then to tie all parameters of all instantiations (of one rule) together. Parameter tying is also a key concept in the procedural probabilistic language PLATES [Buntine, 1994], in which its possible to specify a template (such as person coming late) as well as the number of times it occurs. In this way, PLATES can model any number of persons.

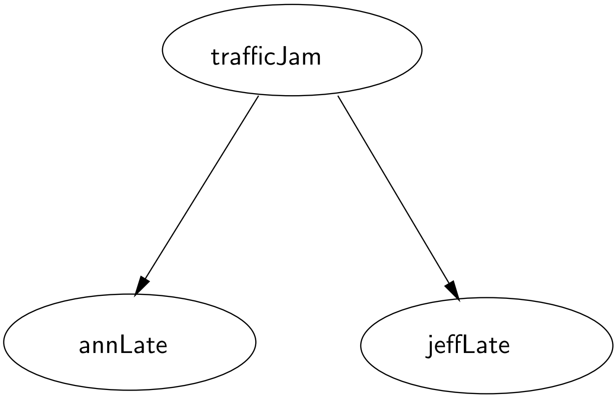


Fig. 8.7. A Bayesian network. $P(\mathsf{t}) = \lambda_0; P(\mathsf{a}|\mathsf{t}) = \lambda_1; P(\mathsf{a}|\neg\mathsf{t}) = \lambda_2$

Exercise 8.13. Show that computing the maximum likelihood estimators for the Bayesian network involving Jeff and Ann essentially corresponds to frequency counting.

Other methods to reduce the number of parameters of a probabilistic model exist as well. One of them involves representing large conditional probability tables in a more compact manner. This can, for instance, be realized by employing a decision tree to encode conditional probability distributions.

8.3.2 Structure Learning

So far, we have addressed only the parameter estimation problem, and have assumed that the structure of the probabilistic model is given and fixed. This assumption is not always satisfied, and indeed in various applications it is often hard to come up with a reasonable structure for the model. Therefore, various techniques have been developed to also learn the structure of the model from data. The *structure learning* problem can be defined as follows:

Given

- a set of examples E
- a language $\mathcal{L}_M$ of possible models of the form $M = (S, \lambda)$ with structure S and parameters λ
- a probabilistic coverage relation $P(e|M)$ that computes the probability of observering the example e given the model M,
- a scoring function $score(E, M)$ that employs the probabilistic coverage relation $P(e|M)$

Find the model $M^* = (S, \lambda)$ that maximizes $score(E, M)$, that is,

$$M^* = \arg\max_M score(E, M) \tag{8.52}$$

This problem is – like most other problems in artificial intelligence and machine learning – essentially a search problem. There is a space of possible models to be considered, defined by $\mathcal{L}_M$, and the goal is to find the best one according to the scoring function. So solution techniques traverse the space of possible models in $\mathcal{L}_M$. By analogy with the learning techniques seen in Chapter 3, one can now devise operators for traversing the space, and also determine extreme points in the search space. For instance, in the case of Bayesian networks, the extreme points could be fully connected Bayesian networks (where there is an edge between any pair of random variables) and one that contains no links at all. At the same time, possible operators include the addition (or deletion) of an edge as well as the reversal thereof. So, the generation and search process closely corresponds to those employed in tradtional relational and logical learning, especially to those in a theory revision setting.

To evaluate a candidate structure S, the parameters λ are first estimated (using the methods developed earlier), and then the scoring function is used to determine the overall score of the resulting model. The problem with the scoring function employed so far, the maximum likelihood, is that it always prefers a fully connected network. Indeed, it can be shown that adding a dependency to a network can never decrease the likelihood. Therefore, the scoring function is typically corrected to penalize for complexity of the network. This can also be justified in Bayesian terms. Indeed, recall from Eq. 8.36 that the goal is to find the model or hypothesis H_{MAP} with the maximum posterior probability:

$$\begin{aligned}
H_{MAP} &= \arg\max_H P(H|E) \\
&= \arg\max_H \frac{P(E|H)P(H)}{P(E)} \\
&= \arg\max_H P(E|H)P(H) \\
&= \arg\max_H \log P(E|H) + \log P(H) \qquad (8.53)
\end{aligned}$$

If the prior probability is higher for simpler models, then the term $\log P(H)$ actually penalizes more complex models.

The state-of-the-art structure learning algorithm is the structural EM (SEM) algorithm [Friedman, 1998]. It adapts the standard EM algorithm for structure learning. The key idea is that the expected counts are not computed anew for every structure that is proposed, but only after several iterations, which yields only an approximation but improves the efficiency.

For completeness, let us mention that there are also other structure learning techniques for graphical models; they often rely on conditional independency tests between the random variables. These tests are then used to constrain the structure of the model.

8.4 First-Order Probabilistic Logics

In this section, we upgrade the earlier propositional probabilistic logics for the use of relational and first-order logics. While doing so, we concentrate on those frameworks that are grounded in (computational) logic and that build upon the frameworks introduced earlier. In order to upgrade logic programs to a probabilistic logic, two changes are necessary:

1. clauses are annotated with probability values, and
2. the covers relation becomes a probabilistic one.

Various choices can be made for realizing this; they have resulted in the wide variety of probabilistic logics that are available today and that are described in two recent textbooks [Getoor and Taskar, 2007, De Raedt et al., 2008]. We will introduce Bayesian logic programs [Kersting and De Raedt, 2007], Markov logic networks [Richardson and Domingos, 2006], stochastic logic programs [Eisele, 1994, Muggleton, 1996, Cussens, 2000], PRISM [Sato, 1995] and ICL [Poole, 1993b]. Bayesian logic programs and Markov logic networks define probability distributions over possible worlds or interpretations, whereas stochastic logic programs, PRISM and ICL define them over derivations or proofs. The former are introduced in Sect. 8.4.1, the latter in Sect. 8.4.2.

8.4.1 Probabilistic Interpretations

Bayesian Logic Programs

In order to integrate probabilities into the learning from interpretation setting, we need to find a way to assign probabilities to interpretations covered by an annotated logic program. In the past few years, this question has received a lot of attention, and various approaches, such as [Pfeffer, 2000], have been developed . In this book, we choose Bayesian logic programs [Kersting and De Raedt, 2001] as the probabilistic logic programming system because Bayesian logic programs combine Bayesian networks [Pearl, 1988], which represent probability distributions over propositional interpretations, with definite clause logic. Furthermore, Bayesian logic programs have already been employed for learning.

The idea underlying Bayesian logic programs is to view ground atoms as random variables that are defined by the underlying definite clause programs. Furthermore, two types of predicates are distinguished: deterministic and probabilistic. The former are called *logical*, the latter *Bayesian*. Likewise, we will also speak of Bayesian and logical atoms. A Bayesian logic program now consists of a set of of Bayesian (definite) clauses, which are expressions of the form $A \mid A_1, \ldots, A_n$ where A is a Bayesian atom, $A_1, \ldots, A_n$, $n \geqslant 0$, are Bayesian and logical atoms and all variables are (implicitly) universally quantified. To quantify probabilistic dependencies, each Bayesian clause c is annotated with its conditional probability distribution $cpd(c) = \mathbf{P}(A \mid A_1, \ldots, A_n)$, which quantifies as a macro the probabilistic dependency among ground instances of the clause.

Let us illustrate Bayesian logic programs using the stud farm example of Jensen [2001], which describes the processes underlying a life-threatening hereditary disease.

Example 8.14. (from [De Raedt and Kersting, 2004]) Consider the following Bayesian clauses:

```
carrier(X) | founder(X).
carrier(X) | mother(M, X), carrier(M), father(F, X), carrier(F).
suffers(X) | carrier(X).
```

They specify the probabilistic dependencies governing the inheritance process. For instance, the second clause states that the probability of a horse being a carrier of the disease depends on whether or not its parents are carriers.

In this example, mother, father, and founder are logical predicates, whereas the other predicates, such as carrier and suffers, are Bayesian. The logical predicates are then defined using a classical definite clause program that constitutes the background theory for this example. It is listed in Ex. 8.15.

Example 8.15. The background theory B is:

father(henry, bill) ←
father(alan, benny) ←
father(bill, carl) ←
father(carl, dennis) ←
mother(ann, betsy) ←
mother(alice, benny) ←
mother(bonnie, cecily) ←
founder(henry) ←
founder(an) ←
founder(alice). ←
father(alan, betsy) ←
father(brian, bonnie) ←
father(benny, cecily) ←
mother(ann, bill) ←
mother(ann, bonnie) ←
mother(betsy, carl) ←
mother(cecily, dennis) ←
founder(alan) ←
founder(brian) ←

The conditional probability distributions for the Bayesian clause are

P(carrier(X) $= true$)
0.6

carrier(X)	P(suffers(X) $= true$)
$true$	0.7
$false$	0.01

carrier(M)	carrier(F)	P(carrier(X) $= true$)
$true$	$true$	0.6
$true$	$false$	0.5
$false$	$true$	0.5
$false$	$false$	0.0

Observe that logical atoms, such as mother(M, X), do not affect the distribution of Bayesian atoms, such as carrier(X), and are therefore not considered in the conditional probability distribution. They only provide variable bindings, for instance, between carrier(X) and carrier(M).

The semantics (and hence, the covers relations) of Bayesian logic programs are defined through a process called *knowledge-based model construction.* In this process, a propositional Bayesian net is constructed by grounding the Bayesian clauses. The resulting net then defines the probability distribution. Often, the knowledge-based model construction process takes into account a particular query to the network, and generates only that grounded part that is needed to answer the query, though we shall ignore such optimizations in the exposition below. For Bayesian logic programs, the random variables A of the constructed Bayesian network are Bayesian ground atoms in the least Herbrand model I of the annotated logic program. A Bayesian ground atom b, say carrier(alan), directly influences another Bayesian ground atom h, say carrier(betsy), if and only if there exists a Bayesian clause c such that

$$b \in body(c)\theta \subseteq I \text{ and } h = head(c)\theta \in I \tag{8.54}$$

The Bayesian network constructed from the Baysian logic program for the stud farm problem is depicted in Fig. 8.8. The nodes are the ground Bayesian atoms in the least Herbrand model of the program, and the direct influence is derived using Eq. 8.54. For instance, in the example, carrier(alan) is a parent node of carrier(betsy) in the constructed model, as it is an instance of the second clause in Ex. 8.14.

Example 8.16. As another example to illustrate Eq. 8.54, consider the carrier Bayesian logic program from Ex. 8.14 together with background theory founder(adam) ← . According to the Eq. 8.54, the least Herbrand model is

$$\{\mathsf{founder(adam)}, \mathsf{carrier(adam)}, \mathsf{suffers(adam)}\}$$

and hence founder(adam) is the only parent of carrier(adam), which in turn the only parent of suffers(adam).

Note that in Bayesian logic programs it is required that the induced network be acyclic and have a finite branching factor. The reason is that Bayesian networks must be directed acyclic graphs.

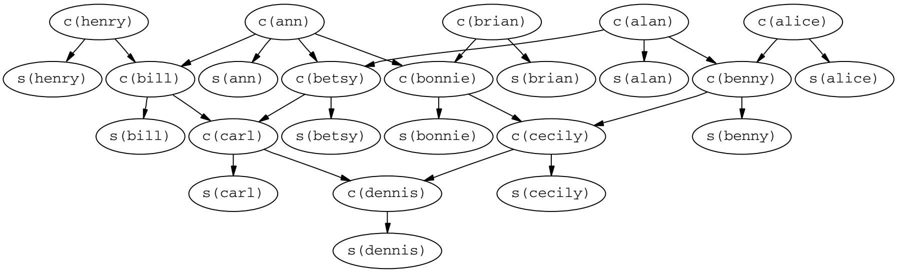


Fig. 8.8. The Bayesian network induced by the stud farm. Reprinted with permission from [De Raedt and Kersting, 2004]

To complete the network, we still need to associate conditional probability distributions with the nodes of the network. Nodes $head(c)\theta$ constructed using Eq. 8.54 are assigned $cpd(c\theta)$ as their associated conditional probability distribution. However, when there exist multiple ground instances in I with the same head, a complication arises with these assignments.

Example 8.17. Consider the Bayesian logic program (adapted from [De Raedt and Kersting, 2003]), defining fever:

fever | cold	cold.
fever | flu	flu.
fever | malaria	malaria.

All three clauses apply, and hence we obtain three conditional probability distributions: $\mathbf{P}(\mathsf{fever}|\mathsf{cold})$, $\mathbf{P}(\mathsf{fever}|\mathsf{flu})$ and $\mathbf{P}(\mathsf{fever}|\mathsf{malaria})$. However, as cold, flu and malaria are parents of fever we need to have a conditional probability distribution defining $\mathbf{P}(\mathsf{fever}|\mathsf{cold}, \mathsf{flu}, \mathsf{malaria})$. Notice that the problem also occurs when there is only one Bayesian clause (with multiple ground instances in I satisfying the conditions of Eq. 8.54 for a particular ground Bayesian atom).

There are two approaches to dealing with this problem. The first solution is adopted in Bayesian logic programs and employs *combining rules*. The second one, *aggregation*, is used in Probablistic Relational Models [Getoor et al., 2001a], which we briefly discuss below.

A *combining rule* is a function that maps finite sets of conditional probability distributions onto one (*combined*) conditional probability distribution. Examples of combining rules are *noisy-or* and *noisy-and*; cf. [Jensen, 2001].

Noisy-or is used in graphical models to reduce the number of parameters. To introduce noisy-or, let us assume that there is a node X in a Bayesian network with a large number (say k) of parents Y_i. As the node has many parents, the CPD associated with the node X becomes very large, and hence hard to specify or learn. If the parents are independent causes for the variable X, the noisy-or rule applies. The noisy-or rule defines the conditional probability distribution as

$$P(X = true|Y_1, ..., Y_k) = 1 - \prod_{Y_i = true} (1 - P(X = true|Y_i = true)) \quad (8.55)$$

The advantage is that one only needs to specify the probability values $P(X = true|Y_i = true)$. This is linear in k instead of exponential. The noisy-or combining rule can also be understood as a more complex network, involving a logical or.

Example 8.18. The fever example can be rewritten as a Bayesian network involving a *logical or*:

```
fever ← cold′          cold′|cold
fever ← flu′           flu′|flu
fever ← malaria′       malaria′|malaria
cold.                  flu.
malaria
```

So, fever is defined in a logical manner, and the intermediate Bayesian predicates cold′, flu′, and malaria′ are introduced.

Exercise 8.19. Verify that Ex. 8.18 correctly represents the noisy-or combining rule as applied to the earlier fever network.

The noisy-or rule can directly be applied to solve the problem with Bayesian logic programs sketched in Ex. 8.17. That network consists of Bayesian clauses of the form $X|Y_i$. There are then k clauses with body literals Y_i that influence X. Using noisy-or, the conditional probability distribution of X can be defined.

An alternative to combining rules is to employ *aggregate functions* as in probabilistic relational models [Koller, 1999, Pfeffer, 2000]. Aggregate functions are well-known from the database literature, and were already discussed in Sect. 4.13. Example aggregate functions include MIN, MAX, SUM, AVG, etc. To illustrate their use in the present context, consider the Bayesian clause

$$\mathsf{influential(Paper)} \mid \mathsf{iscitedby(Paper, Author)}$$

and assume that paper jmlr55 is cited by mary and john. Then there are multiple ground rules satisfying Eq. 8.54 for the Bayesian atom influential(jmlr5r). For this type of rule, the particular Author citing the paper is less important than the number of authors citing it. Therefore, the aggregate function COUNT could be applied to obtain the number of authors citing a particular paper. The conditional probability distribution of the Bayesian clause defining influential could then be defined in terms of the aggregate value, for instance, if the count is larger than 100, the probability could be 0.80, and otherwise 0.15. As this examples illustrates, aggregation reduces information in a natural way. This is often advantageous but it may also result in a loss of information; cf. [Jensen and Neville, 2002] and Sect. 4.13.

The influential paper example also points to what is still largely an open research question: *lifted probabilistic inference* [Poole, 2003, de Salvo Braz et al., 2007]. The problem with the knowledge-based model construction approach is that it requires us to generate all instances of the influential clause in order to determine the probability of influential(jmlr55). That corresponds to grounding almost a complete program. It sharply contrasts with inference procedures for clausal logic, such as those based on resolution, which avoids grounding whenever possible. Approaches to realizing this type of inference are discussed by Poole [2003] and de Salvo Braz et al. [2007]. The idea is that groundings are avoided where possible. This is, however, much more complex than resolution. One reason is that the probabilities often depend on the *number* of possible instances.

Example 8.20. Assuming that there are n authors for which iscitedby(jmlr55, X) is true, that these atoms are marginally independent of one another, and true with probability p, and that a paper is influential (with probability 1) if it is cited once; then

$$P(\mathsf{influential(jmlr55)}) = 1 - (1-p)^n \tag{8.56}$$

To compute this probability in this example, the complete model need not be constructed, as long as one knows the number of possible authors and the independency assumptions hold.

The probability distribution induced by the Bayesian logic program on the possible world is:

$$\mathbf{P}(I|H) = \prod_{\text{Bayesian atom } A \in I} \mathbf{P}(A|\mathbf{Pa}(A)) \tag{8.57}$$

where the $\mathbf{Pa}(A)$ are constructed using Eq. 8.54 and, where necessary, the $\mathbf{P}(A|\mathbf{Pa}(A))$ are constructed using combining rules or aggregation functions.

Example 8.21. For instance, in the stud farm, using the above definition, the probability of the interpretation

$$\begin{array}{lll}\{\mathsf{carrier(henry)} = false, & \mathsf{suffers(henry)} = false, & \mathsf{carrier(ann)} = true, \\ \mathsf{suffers(ann)} = false, & \mathsf{carrier(brian)} = false, & \mathsf{suffers(brian)} = false, \\ \mathsf{carrier(alan)} = false, & \mathsf{suffers(alan)} = false, & \mathsf{carrier(alice)} = false, \\ \mathsf{suffers(alice)} = false, & \ldots\} & \end{array}$$

can be computed using a standard Bayesian network inference engine on the constructed model. Thus, Eq. 8.57 defines a probabilistic coverage relation. In addition, various types of inference would be possible. One might, for instance, ask for the probability $P(\mathsf{suffers(henry)}|\mathsf{carrier(henry)} = true)$, which can be computed using a standard Bayesian network inference engine.

Probabilistic Relational Models*

Probabilistic relational models [Getoor et al., 2001a] are a relational extension of Bayesian networks. Because they are quite popular, we briefly describe their relationship to Bayesian logic programs. Probabilistic relational models do not use definite clause logic as the underlying representational framework, but rather the entity-relationship model presented in Sect. 4.4. In Probabilistic relational models, the information about one entity type is stored in one relation. For instance, in the carrier illustration, persons can be represented using the relation person(Person, Carrier, Suffers), where Person is the key of the relation and Carrier and Suffers the attributes. So, each ground atom or tuple stores information about multiple attributes and the dependencies are defined at the level of these attributes. Two types of dependencies are allowed: *direct dependencies* from other attributes or dependencies via so-called *slot chains.* As an example of a direct dependency, the attribute Suffers may depend on the attribute Carrier). This can be written as

$$\mathsf{person(Person).Suffers} \mid \mathsf{person(Person).Carrier}$$

Slot chains are binary relations (or binary projections of other relations) that relate the attributes of one entity to those of others. For instance, in the carrier illustration, the Carrier attribute of a Person depends on the Carrier attribute of its mother and father through the relations (that is, slot chains in probabilistic relational models) mother and father. This can be represented using the notation

$$\begin{array}{ll}\mathsf{person(Person).Carrier} \mid & \mathsf{mother(Person, Mother), person(Mother).Carrier,} \\ & \mathsf{father(Person, Father), person(Father).Carrier}\end{array}$$

In the basic probabilistic relational model setting such slot-chains are assumed to be deterministic and given (cf. [Getoor et al., 2001a]), which implies that it is impossible to specify that the probability is 0.70 that Jef is the father of Mary, though some partial solutions for dealing with probabilistic relations have been developed; cf. [Getoor et al., 2001b, Getoor, 2001]. The carrier example shows that, at the logical level, probabilistic relational models essentially define via single clauses probabilistic dependencies between the attributes of various entities. There exists also an appealing graphical notation for probabilistic relational models.

Probabilistic relational models are closely related to the CLP(BN) formalism of Costa et al. [2003a], which employs the same ideas in the context of a (constraint) logic programming language.

Markov Logic*

Markov logic combines first-order logic with Markov networks. The idea is to view logical formulae as soft constraints on the set of possible worlds, that is, on the interpretations. If an interpretation does not satisfy a logical formula, it becomes less probable, but not necessarily impossible as in traditional logic. Hence, the more formulae an interpretation satisfies, the more likely it becomes. In a Markov logic network, this is realized by associating a weight with each formula that reflects how strong the constraint is. More precisely, a Markov logic network consists of a set of weighted clauses $H = \{c_1, \cdots, c_n\}$.[2] The weights w_i of the clauses then specify the strength of the clausal constraint.

Example 8.22. Consider the following example (adopted from [Richardson and Domingos, 2006]). Friends & Smokers is a small Markov logic network that computes the probability of a person having lung cancer on the basis of her friends smoking. This can be encoded using the following weighted clauses:

$$\begin{aligned}
&1.5 : \mathsf{cancer(P)} \leftarrow \mathsf{smoking(P)}\\
&1.1 : \mathsf{smoking(X)} \leftarrow \mathsf{friends(X,Y), smoking(Y)}\\
&1.1 : \mathsf{smoking(Y)} \leftarrow \mathsf{friends(X,Y), smoking(X)}
\end{aligned}$$

The first clause states the soft constraint that smoking causes cancer. So, interpretations in which persons that smoke have cancer are more likely than those where they do not (under the assumptions that other properties remain constant). The second and third clauses state that the friends of smokers are also smokers.

A Markov logic network together with a Herbrand domain (in the form of a set of constants $\{d_1, \cdots, d_k\}$) then induces a grounded Markov network, which defines a probability distribution over the possible Herbrand interpretations.

The nodes, that is, the random variables in the grounded network, are the atoms in the Herbrand base. Furthermore, for every substitution θ that grounds a clause c_i in H, there will be an edge between any pair of atoms $a\theta, b\theta$ that occurs in $c_i\theta$. The Markov network obtained for the constants anna and bob is shown in Fig. 8.9. To obtain a probability distribution over the Herbrand intepretations, we still need to define the potentials. The probability distribution over intepretations I is

[2] Markov logic networks, in principle, also allow one to use arbitrary logical formulae, not just clauses. However, for reasons of simplicity, we will only employ clauses throughout this section, and make some further simplifications.

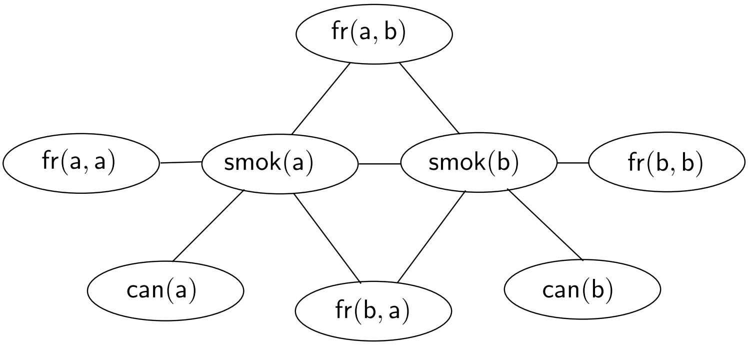


Fig. 8.9. The Markov network for the constants ann and bob. Adapted from [Richardson and Domingos, 2006]

$$\mathbf{P}(I) = \frac{1}{Z} \prod_{c:clause} \mathbf{f}_c(I) \tag{8.58}$$

where the f_c are defined as

$$f_c(I) = e^{n_c(I) w_c} \tag{8.59}$$

and $n_c(I)$ denotes the number of substitutions θ for which $c\theta$ is satisfied by I, and Z is a normalization constant. The definition of a potential as an exponential function of a weighted feature of a clique is common in Markov networks. The reason is that the resulting probability distribution is easier to manipulate.

Note that for different (Herbrand) domains, different Markov networks will be produced. Therefore, one can view Markov logic networks as a kind of template for generating Markov networks, and hence Markov logic is based on knowledge-based model construction. Notice also that like Bayesian logic programs, Markov logic networks define a probability distribution over interpretations, and nicely separate the qualitative from the quantitive component.

Exercise 8.23. Specify the probability of the Herbrand interpretation

$$\{\mathsf{smokes(ann), smokes(bob), friends(ann, bob), cancer(ann)}\}$$

according to the Markov logic network.

8.4.2 Probabilistic Proofs

Many probabilistic logic programming formalisms do not explicitly encode a set of conditional independency assumptions, as in Bayesian or Markov networks, but rather extend proof procedures with probabilistic choices. Stochastic logic programs [Muggleton, 1996, Cussens, 2001] directly upgrade stochastic context-free grammars towards definite clause logic, whereas PRISM [Sato, 1995], Probabilistic Horn Abduction [Poole, 1993b] and the more recent Independent Choice Logic (ICL) [Poole, 1997] specify probabilities on facts from which further knowledge can be deduced. We discuss these two types of framework in turn.

Stochastic Logic Programs

Stochastic logic programs combine principles from computational logic with those of stochastic or probabilistic context-free grammars. A stochastic logic program is a definite clause program, where each of the clauses is labeled with a probability value. Furthermore, as in context-free grammars, it is required that the sum of the probability values for all clauses defining a particular predicate be equal to 1 (though less restricted versions have been considered as well).

Example 8.24. The stochastic logic program below defines a probability distribution on a card deck with 32 cards. The suits are d(iamonds), h(earts), s(pades), and c(lubs). The ranks are a(ce), 7, 8, 9, 10, and f(armer), q(ueen), and k(ing).

$$
\begin{array}{ll}
1 : \mathsf{card}(\mathsf{X},\mathsf{Y}) \leftarrow \mathsf{rank}(\mathsf{X}), \mathsf{suit}(\mathsf{Y}) & \\
0.25 : \mathsf{suit}(\mathsf{d}) \leftarrow & 0.25 : \mathsf{suit}(\mathsf{c}) \leftarrow \\
0.25 : \mathsf{suit}(\mathsf{h}) \leftarrow & 0.25 : \mathsf{suit}(\mathsf{s}) \leftarrow \\
0.125 : \mathsf{rank}(\mathsf{a}) \leftarrow & 0.125 : \mathsf{rank}(10) \leftarrow \\
0.125 : \mathsf{rank}(7) \leftarrow & 0.125 : \mathsf{rank}(\mathsf{k}) \leftarrow \\
0.125 : \mathsf{rank}(8) \leftarrow & 0.125 : \mathsf{rank}(\mathsf{q}) \leftarrow \\
0.125 : \mathsf{rank}(9) \leftarrow & 0.125 : \mathsf{rank}(\mathsf{f}) \leftarrow
\end{array}
$$

Although the present program defines the uniform probability distribution over the 32 cards, it is easy to bias it towards specific ranks or suits by changing the corresponding probability labels. We can also model the predicate

$$1.0 : \mathsf{sameSuit}(\mathsf{S1},\mathsf{S2}) \leftarrow \mathsf{suit}(\mathsf{S1}), \mathsf{suit}(\mathsf{S2}), \mathsf{S1} = \mathsf{S2}.$$

which succeeds for pairs of cards of equal rank.

Example 8.25. Consider the following *definite clause grammar* (see Chapter 4 for an introduction to definite clause grammars).

$$
\begin{array}{l}
1 : \mathsf{sentence}(\mathsf{A},\mathsf{B}) \leftarrow \mathsf{nounPhrase}(\mathsf{C},\mathsf{A},\mathsf{D}), \mathsf{verbPhrase}(\mathsf{C},\mathsf{D},\mathsf{B}). \\
1 : \mathsf{noun} - \mathsf{phrase}(\mathsf{A},\mathsf{B},\mathsf{C}) \leftarrow \mathsf{article}(\mathsf{A},\mathsf{B},\mathsf{D}), \mathsf{noun}(\mathsf{A},\mathsf{D},\mathsf{C}). \\
1 : \mathsf{verb} - \mathsf{phrase}(\mathsf{A},\mathsf{B},\mathsf{C}) \leftarrow \mathsf{intransitiveVerb}(\mathsf{A},\mathsf{B},\mathsf{C}). \\
\frac{1}{3} : \mathsf{article}(\mathsf{singular},\mathsf{A},\mathsf{B}) \leftarrow \mathsf{terminal}(\mathsf{A},\mathsf{a},\mathsf{B}). \\
\frac{1}{3} : \mathsf{article}(\mathsf{singular},\mathsf{A},\mathsf{B}) \leftarrow \mathsf{terminal}(\mathsf{A},\mathsf{the},\mathsf{B}). \\
\frac{1}{3} : \mathsf{article}(\mathsf{plural},\mathsf{A},\mathsf{B}) \leftarrow \mathsf{terminal}(\mathsf{A},\mathsf{the},\mathsf{B}). \\
\frac{1}{2} : \mathsf{noun}(\mathsf{singular},\mathsf{A},\mathsf{B}) \leftarrow \mathsf{terminal}(\mathsf{A},\mathsf{turtle},\mathsf{B}). \\
\frac{1}{2} : \mathsf{noun}(\mathsf{plural},\mathsf{A},\mathsf{B}) \leftarrow \mathsf{terminal}(\mathsf{A},\mathsf{turtles},\mathsf{B}). \\
\frac{1}{2} : \mathsf{intransitiveVerb}(\mathsf{singular},\mathsf{A},\mathsf{B}) \leftarrow \mathsf{terminal}(\mathsf{A},\mathsf{sleeps},\mathsf{B}). \\
\frac{1}{2} : \mathsf{intransitiveVerb}(\mathsf{plural},\mathsf{A},\mathsf{B}) \leftarrow \mathsf{terminal}(\mathsf{A},\mathsf{sleep},\mathsf{B}). \\
1 : \mathsf{terminal}([\mathsf{A}|\mathsf{B}],\mathsf{A},\mathsf{B}) \leftarrow
\end{array}
$$

It covers the proof tree shown in Fig. 8.10.

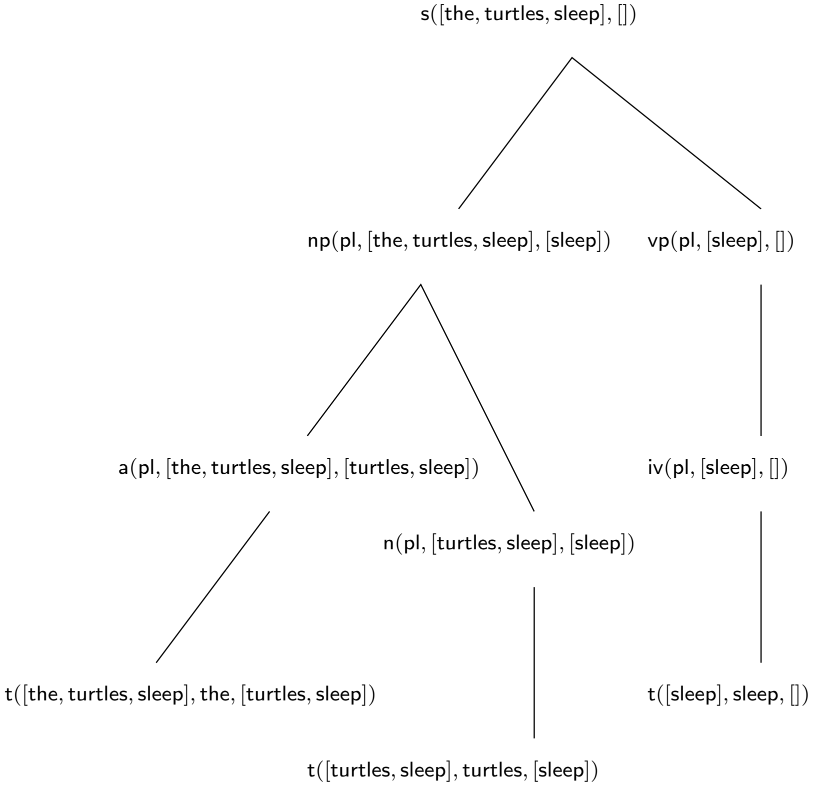

s([the, turtles, sleep], [])
np(pl, [the, turtles, sleep], [sleep])
vp(pl, [sleep], [])
a(pl, [the, turtles, sleep], [turtles, sleep])
iv(pl, [sleep], [])
n(pl, [turtles, sleep], [sleep])
t([the, turtles, sleep], the, [turtles, sleep])
t([sleep], sleep, [])
t([turtles, sleep], turtles, [sleep])

Fig. 8.10. A proof tree

As probabilistic context-free grammars, stochastic logic programs define probability distributions over derivations and atoms. However, there is one crucial difference between context-free grammars and logic programs. Resolution derivations for logic programs can fail, whereas derivations in a context-free grammar never fail. Indeed, if an intermediate rule of the form $S \rightarrow t_1, ..., t_k, n_1, ..., n_m$ is derived in a grammar, where the t_i are terminal symbols and the n_j are non-terminal ones, it is always possible to "resolve" a non-terminals n_j away using any rule defining n_j. In contrast, in a logic program, this is not always possible because the two literals may not unify.

Example 8.26. Consider the SLD-tree for the goal $\mathsf{sameSuit}(\mathsf{X}, \mathsf{Y})$ in the card example. Even though there are $4 \times 4 = 16$ leaves of the SLD-tree, only four of them succeed due to unification.

We now define three different probability distributions; we will, for simplicity, assume that all SLD-trees are finite; cf. Cussens [2001] for a more general treatment. First, let us define the distribution over SLD-derivations of an atom a of the form $p(X_1, ..., X_n)$ where all the arguments are different logical variables. The derivations correspond to success or failure branches in the SLD-trees. Such derivations are similar to those encountered in probabilistic context-free grammars, and hence we can define the probability $P_D(d(a))$ of a derivation $d(a)$ for the atom a, in which the clauses $c_1, ..., c_n$ with associated probability labels $p_1, ..., p_n$ are used $m_1, ..., m_n$ times, as

$$P_D(d(a)) = \prod_i p_i^{m_i} \tag{8.60}$$

Observe that the probability distribution P_D also assigns a non-zero probability to failed derivations. Usually, we are interested in successful derivations, that is, refutations ending in $\square$. The probability distribution $P_R(r(a))$, defined on refutations $r(a)$ for an atom a and induced by P_D and the logic program, can be obtained by normalizing P_D:

$$P_R(r(a)) = \frac{P_D(r(a))}{\sum_{r(g)} P_D(r(g))} \tag{8.61}$$

where the sum ranges over all SLD-refutations $r(g)$ of the goal g.

Example 8.27. Continuing the poker example shows that all derivations d for the $\mathsf{sameSuit(X, Y)}$ predicate have probability $P_D(d) = 0.25 \times 0.25$. Because only four of them are also refutations, the probability of a refutation r is

$$P_R(r) = \frac{0.25 \times 0.25}{4 \times 0.25 \times 0.25} = 0.25 \tag{8.62}$$

The probability distribution P_R is analoguous to that defined on parse trees in stochastic context free grammars. Therefore, it can be used to define the probability $P_A(a\theta)$ of a ground atom $a\theta$:

$$P_A(a\theta) = \sum_{r_i(a\theta)} P_R(r_i(a\theta)) \tag{8.63}$$

where r_i ranges over all possible refutations for $a\theta$. This is the equivalent of the statement that the probability of a sentence in a stochastic grammar is the sum of the probabilities of its parse trees.

Example 8.28. The value P_D of the proof tree u in Fig. 8.10 is $v_u = \frac{1}{3} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{12}$. The only other ground proofs s_1, s_2 of atoms over the predicate $\mathsf{sentence}$ are those of $\mathsf{sentence([a, turtle, sleeps], [])}$ and $\mathsf{sentence([the, turtle, sleeps], [])}$. Both get value $= \frac{1}{12}$. Because there is only one proof for each of the sentences, $P_A(\mathsf{sentence([the, turtles, sleep], [])}) = \frac{1}{3}$, and this is also the probability P_R of the corresponding refutations.

Like Markov models and stochastic context-free grammars, stochastic logic programs can be employed for:

- *Decoding*: computing the probability $P_A(a)$ of an atom a.
- *Most likely proof tree*: computing the most likely proof tree for a given query or atom a, that is, $\arg\max_{r(a)} P_R(r(a))$.
- *Sampling* instances from a predicate according to the P_A distribution.
- *Learning* the parameters or the structure of a stochastic logic program from data; cf. Sect. 8.5.

Probabilities on Proofs Using Facts

Stochastic logic programs directly upgrade stochastic context-free grammars, and therefore define a probability distribution P_A for each predicate. This distribution defines how likely it is to sample particular atoms, and can also be used to find the most likely proof tree or explanation for a query, but somehow does not naturally specify the probability that a fact or a possible world holds. Therefore, there is also another approach to defining probabilities over proofs.

This approach, which is incorporated in probabilistic logics such as PRISM [Sato and Kameya, 2001, Sato, 1995, Sato and Kameya, 1997], Probabilistic Horn Abduction [Poole, 1993b], Independent Choice Logic [Poole, 1997], LPADs [Vennekens et al., 2004] and PROBLOG [De Raedt et al., 2007b], assigns probabilities to facts. During the exposition of this approach we shall, as usual, focus on the underlying principles (rather than the available implementations) and simplify them where useful. The key idea underlying these approaches is that some facts f for *probabilistic* predicates are annotated with a probability value. This value indicates the degree of belief, that is the probability, that any ground instance $f\theta$ of f is true. It is also assumed that the $f\theta$ are marginally independent. The probabilistic facts are then augmented with a set of definite clauses defining further predicates (which should be disjoint from the probabilistic ones). An example adapted from [De Raedt et al., 2007b] is given below.

Example 8.29. Consider the facts

$$\begin{array}{ll} 0.9 : \mathsf{edge(a,c)} \leftarrow & \quad 0.7 : \mathsf{edge(c,b)} \leftarrow \\ 0.6 : \mathsf{edge(d,c)} \leftarrow & \quad 0.9 : \mathsf{edge(d,b)} \leftarrow \end{array}$$

which specify that with probability 0.9 there is an edge from a to c. Consider also the following (simplified) definition of path/2.

$$\begin{array}{l} \mathsf{path(X,Y)} \leftarrow \mathsf{edge(X,Y)} \\ \mathsf{path(X,Y)} \leftarrow \mathsf{edge(X,Z)}, \mathsf{path(Z,Y)} \end{array}$$

One can now define a probability distribution on (ground) proofs. The probability of a ground proof is then the product of the probabilities of the (ground) clauses (here, facts) used in the proof. For instance, the only proof for the goal $\leftarrow \mathsf{path(a,b)}$ employs the facts edge(a, c) and edge(c, b); these facts are marginally independent, and hence the probability of the proof is $0.9 \cdot 0.7$. The probabilistic facts used in a single proof are sometimes called an *explanation.*

It is now tempting to define the probability of a ground atom as the sum of the probabilities of the proofs for that atom. However, this does not work without additional restrictions, as shown in the following example.

Example 8.30. The fact $\mathsf{path(d,b)}$ has two explanations:

- $\{\mathsf{edge(d,c)}, \mathsf{edge(c,b)}\}$ with probability $0.6 \times 0.7 = 0.42$, and
- $\{\mathsf{edge(d,b)}\}$ with probability 0.9.

Summing the probabilities of these explanations gives a value of 1.32, which is clearly impossible.

The reason for this problem is that the different explanations are not mutually exclusive, and therefore their probabilities may not be summed. This contrasts with stochastic logic programs, where at each choice point during a proof, only one clause can be selected, and hence the possible choices are mutually exclusive. The probability $P(\mathsf{path(d,b)} = \mathit{true})$ is, however, equal to the probability that *a* proof succeeds, that is

$$P(\mathsf{path(d,b)} = \mathit{true}) = P[(\mathsf{e(d,c)} \wedge \mathsf{e(c,b)}) \vee \mathsf{e(d,b)}] \tag{8.64}$$

which shows that computing the probability of a derived ground fact reduces to computing the probability of a boolean formula in disjunctive normal form (DNF), where all random variables are marginally independent of one another. Computing the probability of such formulae is an NP-hard problem, the *disjoint-sum* problem. The naive approach for realizing it employs the *inclusion-exclusion* principle from set theory. This principle, applied to our problem, states:

$$\begin{aligned}
P(X_1 \vee ... \vee X_n) = & \sum_{i=1}^{n} P(X_i) - \sum_{1 \leqslant i_1 < i_2 \leqslant n} P(X_{i_1} \wedge X_{i_2}) && (8.65)\\
& + \sum_{1 \leqslant i_1 < i_2 < i_3 \leqslant n} P(X_{i_1} \wedge X_{i_2} \wedge X_{i_3}) && (8.66)\\
& - ... + (-1)^n P(X_1 \wedge ... \wedge X_n) && (8.67)
\end{aligned}$$

Example 8.31. Applied to our path example, we obtain:

$$\begin{aligned}
P(\mathsf{path(d,b)} = \mathit{true}) &= P[(\mathsf{e(d,c)} \wedge \mathsf{e(c,b)}) \vee \mathsf{e(d,b)}] && (8.68)\\
&= P(\mathsf{e(d,c)} \wedge \mathsf{e(c,b)}) + P(\mathsf{e(d,b)}) && (8.69)\\
&\quad -P((\mathsf{e(d,c)} \wedge \mathsf{e(c,b)}) \wedge \mathsf{e(d,b)}) && (8.70)\\
&= 0.6 \times 0.7 + 0.9 - 0.6 \times 0.7 \times 0.9 = 0.942 && (8.71)
\end{aligned}$$

There exist more effective ways to compute the probability of such DNF formulae; cf. [De Raedt et al., 2007b], where binary decision diagrams [Bryant, 1986] are employed to represent the DNF formula. The ProbLog system has also been applied to link mining problems in large biological networks.

The above example has shown how the probability of a specific fact is defined and can be computed. The distribution at the level of individual facts (or goals) can easily be generalized to a possible world semantics, specifying a probability distribution on interpretations. It is formalized in the *distribution semantics* of Sato [1995], which is defined by starting from the set of all probabilistic ground facts F for the given program. For simplicity, we shall assume that this set is finite, though Sato's results also hold for the infinite case. The distribution semantics then starts from a probability distribution $P_F(S)$ defined on subsets $S \subseteq F$:

$$P_F(S) = \prod_{f \in S} P(f) \prod_{f \notin S} (1 - P(f)) \tag{8.72}$$

Each subset S is now interpreted as a set of logical facts and combined with the definite clause program R that specifies the logical part of the probabilistic logic program. Any such combination $S \cup R$ possesses a unique least Herbrand model $M(C)$ (cf. Sect. 2.3), which corresponds to a possible world. The probability of such a possible world is then the sum of the probabilities of the subsets S yielding that possible world, that is:

$$P_W(M) = \sum_{S \subseteq F : M(S \cup R) = M} P_F(S) \tag{8.73}$$

For instance, in the path example, there are 16 possible worlds, which can be obtained from the 16 different truth assignments to the facts, and whose probabilities can be computed using Eq. 8.73. As we have seen when discussing Bayesian networks (cf. Eq. 8.17), the probability of any logical formulae can be computed from a possible world semantics (specified here by P_W).

Prism, PHA and ICL

Because computing the probability of a fact or goal under the distribution semantics is hard, researchers such as Taisuke Sato and David Poole have introduced the probabilistic logics Prism [Sato, 1995, Sato and Kameya, 1997, 2001], Probabilistic Horn Abduction (PHA) [Poole, 1993b] and the Independent Choice Logic (ICL) [Poole, 1997] that impose additional restrictions that can be used to improve the efficiency of the inference procedure.

The key assumption is that the explanations for a goal are *mutually exclusive*, which overcomes the disjoint-sum problem. If the different explanations of a fact do not overlap, then its probability is simply the sum of the probabilities of its explanations. This directly follows from the inclusion-exclusion formulae as under the exclusive-explanation assumption the conjunctions (or intersections) are empty.

In addition, PHA, ICL and PRISM employ *disjoint statements* of the form $\mathsf{disjoint}(p_1 : a_1; ...; p_n : a_n) \leftarrow$, where the a_i are atoms for a particular predicate, and the p_i probability values satisfy $\sum_i p_i = 1$. For instance, the statement

$$\mathsf{disjoint}(0.3 : \mathsf{gene}(\mathsf{P}, \mathsf{a}); 0.15 : \mathsf{gene}(\mathsf{P}, \mathsf{b}); 0.55 : \mathsf{gene}(\mathsf{P}, \mathsf{o})) \leftarrow$$

states that 1) the probabilities that (an instance) of the corresponding fact for gene is true, and 2) an atom of the form gene(P, X) instantiates to exactly one of these options. This corresponds to enforcing the constraints:

$$\begin{array}{l} \mathsf{false} \leftarrow \mathsf{gene}(\mathsf{P}, \mathsf{a}), \mathsf{gene}(\mathsf{P}, \mathsf{b}) \\ \mathsf{false} \leftarrow \mathsf{gene}(\mathsf{P}, \mathsf{a}), \mathsf{gene}(\mathsf{P}, \mathsf{c}) \\ \mathsf{false} \leftarrow \mathsf{gene}(\mathsf{P}, \mathsf{b}), \mathsf{gene}(\mathsf{P}, \mathsf{c}) \end{array}$$

The advantage of using disjoint statements is that it becomes much easier to model random variables over domains other than $\{$*true, false*$\}$.

Example 8.32. Consider the previous disjoint statement together with the following definite clause program (adapted from Sato and Kameya [1997]). It states the well-known rules for determining the bloodtype of a person depending on the genes she inherits from her parents. To have bloodtype a, the genes of both parents should be a, or else a combined with o; the rules are similar for b, and then bloodtype ab corresponds to a combined with a, and o corresponds to both genes being o. The clauses defining dominates, pgtype, gtype and btype define deductive knowledge. The predicate gene, defined in the above disjoint statement, is a probabilistic predicate. It states the probabilities of the genes a, b and o.

$$\begin{array}{l} \mathsf{btype}(\mathsf{Btype}) \leftarrow \mathsf{gtype}(\mathsf{Gf}, \mathsf{Gm}), \mathsf{pgtype}(\mathsf{Btype}, \mathsf{Gf}, \mathsf{Gm}) \\ \mathsf{gtype}(\mathsf{Gf}, \mathsf{Gm}) \leftarrow \mathsf{gene}(\mathsf{mother}, \mathsf{Gf}), \mathsf{gene}(\mathsf{father}, \mathsf{Gm}) \\ \mathsf{pgtype}(\mathsf{X}, \mathsf{X}, \mathsf{X}) \leftarrow \\ \mathsf{pgtype}(\mathsf{X}, \mathsf{X}, \mathsf{Y}) \leftarrow \mathsf{dominates}(\mathsf{X}, \mathsf{Y}) \\ \mathsf{pgtype}(\mathsf{X}, \mathsf{Y}, \mathsf{X}) \leftarrow \mathsf{dominates}(\mathsf{X}, \mathsf{Y}) \\ \mathsf{pgtype}(\mathsf{ab}, \mathsf{a}, \mathsf{b}) \leftarrow \\ \mathsf{pgtype}(\mathsf{ab}, \mathsf{b}, \mathsf{a}) \leftarrow \\ \mathsf{dominates}(\mathsf{a}, \mathsf{o}) \leftarrow \\ \mathsf{dominates}(\mathsf{b}, \mathsf{o}) \leftarrow \end{array}$$

A labeled atom in a disjoint statement states the probability with which (an instance of) the fact is true. This induces a probability distribution at the level of the proofs. Indeed, using the disjoint statement, the facts gene(mother, a) and gene(father, b) are true with probability 0.3 and 0.15 respectively. Therefore, the probability of the proof of bloodtype(ab) that involves these two facts is $0.3 \times 0.15 = 0.045$.

Using the distribution semantics, the probability distribution for proofs induces a probability distribution for the atoms of predicates in R, and hence for possible worlds, or interpretations. Indeed, because there is only one possible proof for btype(o), the probability of this atom being true is $0.55 \times 0.55 = 0.3025$. If there are more proofs, then one has to sum the probabilities of the corresponding explanations. This is safe under the *exclusive-explanation* assumption.

Example 8.33. For instance, for bloodtype(a), there are three proofs, relying on

- gene(m, a), gene(f, a) with probability $0.3 \times 0.3 = 0.09$
- gene(m, a), gene(f, o) with probability $0.3 \times 0.55 = 0.165$
- gene(m, o), gene(f, a) with probability $0.55 \times 0.3 = 0.165$

So, the probability of the atom bloodtype(a) $= 0.09 + 0.165 + 0.165 = 0.42$.

Example 8.34. The probability of the least Herbrand interpretation of the deductive part of the PRISM program with {gene(m, a), gene(f, a)} is 0.09. It constitutes one possible world and includes the fact btype(a).

Probabilistic Abduction

The representations of PHA, ICL and PRISM were originally motivated as a kind of probabilistic abduction. It is therefore interesting to link these representations to the abductive logic programming framework introduced in Sect. 7.2.3. The procedure for generating an abductive explanation of a ground atom sketched in Algo. 7.2 can be adapted to the representations of PHA, ICL and PRISM, which work with disjoint statements under the exclusive-explanation assumption. The resulting algorithm is shown in Algo. 8.2; it generates also the probabilities of the explanations. The theory T is the deductive part of the program. The integrity constraints I are obtained from the disjoint statements, as indicated above. The function is called with the parameters $\leftarrow g$, $\{\}$ and 1.

As the function **abduce** of Sect. 7.2.3 could be modified to yield only the minimal explanations (using a breadth-first strategy), **prob-abduce** can be extended into a best-first search strategy that will generate the most likely explanations first; cf. [Poole, 1993a]. This involves selecting the (goal, explanation, probability) tuple with maximal probability first from the queue of candidates instead of following a last-in-first-out strategy. In addition, one must verify whether the explanations are minimal.

Notice that their key difference with the Bayesian network approaches is that the ICL and PRISM approaches rely on *logical* inference methods, whereas Bayesian networks rely more on *probabilistic* inference mechanisms.

Exercise 8.35. Simulate **prob-abduce** on the fact bloodtype(a).

Exercise 8.36. * Represent the Bayesian traffic network as an ICL or a PRISM program?

Exercise 8.37. ** Can a stochastic logic program be represented as an ICL or a PRISM program? Is this always possible?

Algorithm 8.2 The function **prob-abduce**($\leftarrow q_1, ..., q_n$; F; P) for probabilistic abduction in a theory T

```
if $n = 0$ then
  return $F$, $P$
else if the predicate in $q_1$ is probabilistic then
  compute a ground substitution $\sigma$ such that $q_1\sigma$ unifies
  with a fact $f$ with probability $p$ in a disjoint statement
  if $F \cup T \cup \{q_1\sigma\}$ satisfies $I$ then
    call prob-abduce($\leftarrow q_2\sigma, ..., q_n\sigma$; $I$; $F \cup \{q_1\sigma\}$, $p \times P$)
  else
    fail
  end if
else if possible then
  select the next clause $q \leftarrow r_1, ..., r_m$ in $T$ for which $mgu(q, q_1) = \theta$
  call prob-abduce($\leftarrow r_1\theta, ..., r_m\theta, q_2\theta, ..., q_n\theta$; $F$; $P$)
else
  fail
end if
```

Logical and Relational Markov Models

The frameworks introduced above are all expressive probabilistic models with logical representations. This, however, comes at a computational cost. Indeed, as we discussed in Sect. 8.2.2, there exist classes of graphical models, such as Markov models and hidden Markov models, for which inference and learning can be realized efficiently, that is, in polynomial time. This contrasts with, for instance, Bayesian networks, for which inference is NP-complete. Therefore, several researchers have been interested in upgrading Markov models to use relations and logic, which has yielded relational Markov models (RMMs) [Anderson et al., 2002] and logical hidden Markov models (LOHMMs) [Kersting et al., 2006]. Whereas the resulting approaches are typically less expressive than the ones introduced above, the advantages are that 1) they are closely related to the underlying representations (and hence define probability distributions over sequences), 2) they are, in principle, more efficient, and 3) the learning algorithms for the underlying representations can almost directly be applied.

These approaches can also be conveniently viewed as downgrading one of the more expressive probabilistic logics presented above. Therefore, rather than introducing relational or logical Markov models directly, we will show how (a variant of) logical Markov models can be modelled as a stochastic logic program. In doing so, we closely follow an example by Anderson et al. [2002] modeling a web navigation problem. A toy example for web navigation within a given academic site is illustrated in Figure 8.11. It is a Markov model displayed as a finite state automaton, with probabilities instead of symbols from an alphabet associated with each of the edges. Alternatively, it can be

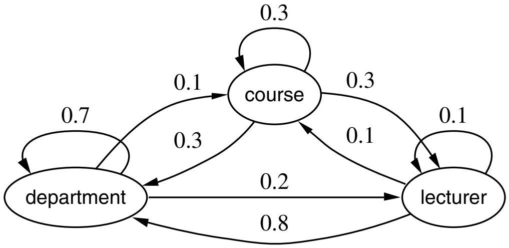


Fig. 8.11. The graph structure of a Markov model for web navigation (from [De Raedt and Kersting, 2003])

regarded as a stochastic regular grammar. The model denotes the probabilities that if one is at a web page of type X the next web page will be of type Y. For instance, in the automaton listed above, the probability of moving from department to lecturer is 0.2. It should be clear that this model is too simple to be useful for web user modelling because the abstraction level is too high. An alternative is to build a model with one node for each of the web pages. Such model would, however, be so huge that it would hardly be usable, let alone learnable. Therefore, a better way, supported by relational and logical Markov models, is to use proper relational or logical *atoms* to represent the state instead of propositional symbols.

Example 8.38. A sequence of possible states could then be, for instance,

$$\begin{aligned}&\mathsf{dept(cs)} \rightarrow \mathsf{course(cs, dm)} \rightarrow \mathsf{lecturer(cs, pedro)} \rightarrow \\ &\mathsf{course(cs, stats)} \rightarrow \ldots\end{aligned}$$

The sequence of pages represented includes the web page of the department of cs, the dm course, its lecturer pedro and the course stats taught by the same lecturer. Notice the similarity with the logical sequences designed in Sect. 4.10. This type of logical sequence can be modeled using the following types of (grammar) rules:

$$\begin{array}{lrl}
(0.7) & \mathsf{dept(D)} & \rightarrow \mathsf{course(D,C)}.\\
(0.2) & \mathsf{dept(D)} & \rightarrow \mathsf{lecturer(D,L)}.\\
 & & \ldots\\
(0.3) & \mathsf{course(D,C)} & \rightarrow \mathsf{lecturer(D,L)}.\\
(0.3) & \mathsf{course(D,C)} & \rightarrow \mathsf{dept(D)}.\\
(0.3) & \mathsf{course(D,C)} & \rightarrow \mathsf{course(D',C')}.\\
 & & \ldots\\
(0.1) & \mathsf{lecturer(X,L)} & \rightarrow \mathsf{course(X,C)}.\\
 & & \ldots
\end{array}$$

Now, starting from any ground state, for instance, dept(cs), one can determine the possible transitions as well as their probabilities. The above logical Markov model specifies, however, only the probability of a transition to an *abstract state*; these are non-ground states, states that contain variables such as course(cs, C) and lecturer(cs, L) in our example. To determine the corresponding real states, one also needs to know 1) the domains of the various arguments (in our examples, the set of all lecturers and the set of all courses, say dom(L) and dom(C)), and 2) a probability distribution on these domains (that is, $\mathbf{P}_L$ and $\mathbf{P}_C$) that specifies the probability of selecting a particular instance from these domains; for instance, the probability of selecting pedro as a lecturer would be given by $P_L(\mathsf{pedro})$. Given these domains and their corresponding distributions, the abstract transitions can be instantiated. For instance, starting from dept(cs) there is a probability of $0.7 \cdot P_L(\mathsf{pedro})$ of going to lecturer(cs, pedro). This illustrates the key ideas underlying these representations: proof steps correspond to time steps.

The resulting model can be conveniently represented as the stochastic logic program sketched in the following example.

Example 8.39. The logical Markov model specified as a stochastic logic program:

$$\begin{array}{l}
\mathsf{0.7 : dept(D) \leftarrow domcourse(C), course(D,C)}\\
\mathsf{0.2 : dept(D) \leftarrow domlect(L), course(D,L)}\\
\ldots\\
\mathsf{0.3 : course(D,C) \leftarrow domlect(L), lecturer(D,L)}\\
\mathsf{0.3 : course(D,C) \leftarrow dept(D)}\\
\mathsf{0.3 : course(D,C) \leftarrow domcourse(C1), domdept(D1), course(D1,C1)}\\
\ldots\\
\mathsf{0.1 : lecturer(X,L) \leftarrow domcourse(C), course(X,C).}\\
\ldots\\
\mathsf{0.1 : domlect(pedro) \leftarrow}\\
\ldots
\end{array}$$

The probability of the proofs in the stochastic logic program then corresponds to the probability of the trace through the Markov model. The domain predicates are introduced only to bind those variables that appear in the body of clauses but not in the head. In the example, we ignored the starting and the terminal states of the Markov model. This is left as an exercise to the reader.

Exercise 8.40. Discuss how to deal with starting and terminal states of the Markov model within the stochastic logic program.

Exercise 8.41. Can you model the Markov model as a PRISM or ICL program?

8.5 Probabilistic Logic Learning

In this section, we tackle the true topic of this chapter: probabilistic logic learning. More specifically, we show how the traditional settings for inductive logic programming carry over to probabilistic logic learning. The resulting techniques are typically combinations of logical learning with the principles of probabilistic learning seen at the start of the chapter. Therefore, this section is rather brief, and does not go into the (often very technical) details of the different approaches. We first introduce learning from interpretations, then learning from entailment, and finally learning from proofs and traces.

8.5.1 Learning from Interpretations

Examples

As we argued earlier, many first-order probabilistic logics, including Bayesian logic programs, Markov logic networks, and probabilistic relational models [Getoor et al., 2001a], define a probability distribution over possible worlds, or interpretations. This is not really surprising as they upgrade Bayesian networks (or Markov networks), which define such a distribution at the propositional level. In Sect. 8.3, we showed how such networks can be learned from interpretations. It is therefore natural to also use interpretations as examples when learning Bayesian logic programs, probabilistic relational models or Markov logic. As before, we can distinguish fully observable from partially observable data. In the fully observable case, one obtains as examples logical interpretations. Indeed, reconsider the interpretation in the stud farm example mentioned in Ex. 8.21 of a Bayesian logic program. In this interpretation, all values of the random variables are known, and hence this possible world is a logical interpretation. Similar examples can be provided for learning Markov logic. In the partially observable case, however, not all values of the random variables are known, and we obtain a *partial interpretation.*

Example 8.42. For instance, in the stud farm domain, a partial interpretation that can be used as an example is

$$\begin{array}{lll}\{\mathsf{carrier(henry)} = false, & \mathsf{suffers(henry)} = false, & \mathsf{carrier(ann)} = true, \\ \mathsf{suffers(ann)} = false, & \mathsf{carrier(brian)} = false, & \mathsf{suffers(brian)} = false, \\ \mathsf{carrier(alan)} = ?, & \mathsf{suffers(alan)} = false, & \mathsf{carrier(alice)} = false, \\ \mathsf{suffers(alice)} = false, & \ldots\} & \end{array}$$

where ? denotes an unobserved state or value.

Using the notion of a partial interpretation, we can directly apply the principles of probabilistic learning discussed in Sect. 8.3 to estimate the parameters and learn the structure of Bayesian logic programs, Markov logic, etc.

Parameter Estimation

Because formalisms such as Bayesian logic programs and Markov logic networks use knowledge-based model construction, the example interpretations can be used to construct a grounded propositional network, and hence parameter estimation techniques for Bayesian and Markov networks directly apply to a grounded network. However, to upgrade these parameter estimation techniques to first-order probabilistic logics, three subtleties need to be considered.

First, as the clauses in the first-order probabilistic logics act as a kind of template, the ground propositional network will typically contain multiple instantiations of the same clause. Therefore, care has to be taken that the corresponding parameters of the different instantiations are *tied together* as discussed in Sect. 8.3.1.

Second, different logical interpretations typically result in different propositional networks. Indeed, consider another stud family, in which john, mary and jef occur. This will result in a different network, and the learner should exercise care to ensure that correct results are obtained. This problem does not only occur in first-order logics; it also occurs in Markov models, where different observation sequences can posses different lengths. The solution in Markov models is to average over the different example sequences. This solution also applies to our learning problem: we need to average over the different propositional networks.

Third, the probabilistic logics may contain combining or aggregation rules, which the learning algorithm should take into account. Many of these rules can, however, be eliminated from a network by reformulating the model. Recall that we were able to remove noisy-or from the fever network by adding auxiliary random variables and using a logical or in Ex. 8.18. The usual parameter estimation procedure can then be applied to the reformulated network, whose parameters correspond to those of the original network. While estimating the parameters of the reformulated network, the deterministic nodes (such as the logical or) are fixed and not modified.

We now sketch how these ideas can be applied to obtain an EM algorithm for learning Bayesian logic programs. Assume that we are trying to re-estimate a parameter p that models the conditional probability $P(h = v|b = w)$ of the Bayesian clause $h|b$ w.r.t. the current set of parameters λ. Then we obtain the following formula for the expected counts:

$$ec_{\lambda}(h = v \wedge b = w) = \sum_{e \in E} \sum_{\theta : h\theta \in e \wedge b\theta \subseteq e} P(h\theta = v \wedge b\theta = w|e) \quad (8.74)$$

$$ec_{\lambda}(b = w) = \sum_{e \in E} \sum_{\theta : h\theta \in e \wedge b\theta \subseteq e} P(b\theta = w|e) \quad (8.75)$$

$$p = \frac{ec_{\lambda}(h = \wedge b = w)}{ec_{\lambda}(b = w)} \quad (8.76)$$

The first summation ranges over all possible examples. The second one captures the parameter tying within a single example. It ranges over all relevant instantiations of the clause in the example. The relevant probabilities can be computed using the standard Bayesian logic program inference engine.

Structure Learning*

To learn the structure of a Bayesian or Markov network (cf. Sect. 8.3), one typically performs a heuristic search through a space of possible structures constructed using a kind of propositional refinement operator which adds, deletes or changes arcs in the network. Thus it seems natural to adapt these algorithms by employing a first-order refinement operator that works at the theory level, as discussed extensively in Chapter 5. There are, however, two complications that may arise when learning probabilistic logics.

First, the sets of random variables in each of the example interpretations should correspond to those of the probabilistic logic. For probabilistic relational models, it is typically assumed that the domain of discourse, that is, the relevant constants and deterministic relations, are known. For Bayesian logic programs, it is required that example interpretations are also a model for the Bayesian logic program in the logical sense.

Example 8.43. Reconsider the Bayesian logic program defining the stud farm and the logical facts defining mother, father and founder. Assume that there is a (logical) fact specifying that founder(jef) is true. Then, because of the (least Herbrand model) semantics of Bayesian logic programs, there must also be a random variable carrier(jef). If this random variable is not included in the example interpretation, the interpretation does not contain all random variables in the grounded Bayesian network, and hence contains latent variables, which are extremely difficult to deal with.

Therefore, when learning Bayesian logic programs, it is required that the ground facts in the interpretation also constitute a model in the logical sense of the Bayesian logic program. This can be enforced using the techniques seen in Sect. 7.5.

Secondly, when learning Bayesian logic programs or probabilistic relational models, care must be taken to ensure that the propositional Bayesian networks obtained by grounding a candidate model with an interpetation does not contain any loops, as Bayesian networks are by definition acyclic. Depending on the expressiveness of the probabilistic logic model, the acyclicity can be determined using the structure of the model alone, or by verifying that the grounded networks do not contain cycles.

8.5.2 Learning from Entailment

Examples

When learning a probabilistic logic from entailment, the examples are, as in the purely logical setting, ground facts that need to be entailed by the target program. For instance, to learn the stochastic logic program modelling the definite clause grammar in Ex. 8.28, a possible example is sentence([the, turtles, sleep], []), and when learning the Prism or ICL program, it is btype(ab).

Parameter Estimation

To address the parameter estimation problem, first consider the simpler case of probablistic context-free grammars. An example in the form of a sentence generated by the grammar contains the result of the inference process; it does not encode the proof or parse tree that explains how the sentence was generated. The proofs or parse trees are therefore essentially unobserved. This situation is akin to that encountered in Bayesian networks, where certain variables are unobserved. Therefore, the same solution applies here as well. The EM algorithm can be applied in order to deal with unobserved variables by computing the expected counts with which the different rules in the grammar are used to generate the data. For probabilistic context-free grammars, the EM algorithm is known as the inside-outside algorithm and it runs in polynomial time, thanks to the use of a dynamic programming technique; cf. [Manning and Schütze, 1999].

This very same idea can now be applied to learn stochastic logic or Prism programs. However, the key difference between these logical types of grammars and the context-free ones needs to be taken into account. This difference concerns the possibility of having failed derivations due to unification. Cussens [2001] introduced an extension of the EM algorithm, the so-called failure-adjusted maximisation algorithm (FAM), to learn the parameters of stochastic logic programs in this way. It was later adapted by Sato et al. [2005] for use in PRISM.

In Cussens's FAM approach, the atoms are treated as an incomplete example set derived from a complete example set of derivations (that is, proofs and failures), truncated to yield refutations only, and finally grouped to produce the set of observed atoms. As shown by [Cussens, 2001], this yields the following formula for computing the expected counts of a clause C_i given E, the set of examples, for the E-Step:

$$ec_\lambda(C_i|E) = \sum_{e \in E} \left(ec_\lambda(C_i|e) + (Z^{-1} - 1) ec_\lambda(C_i|\mathsf{fail}) \right) \tag{8.77}$$

Here, $ec_\lambda(C_i|e)$ (or $ec_\lambda(C_i|\mathsf{fail})$) denotes the expected number of times clause C_i has been used to derive atom e (or to derive a failure), and Z is the normalization constant

$$Z = \sum_{r(h)} P_D(r(h))$$

where the sum ranges over all proof-trees $r(h)$ of the variabilized head h of clauses C_i. In the M-Step, FAM computes the improved probability label p_i for each clause C_i as

$$p_i = \frac{ec_\lambda(C_i|E)}{\sum_{C'} ec_\lambda(C'|E)}$$

where the sum ranges over all clauses C' with the same predicate in the head as C_i.

Structure Learning

Learning both the structure and the parameters of a first-order probabilistic logic from entailment from scratch is extremely hard, if not impossible. The reason is that this requires a combined solution to the structure learning problem in both probabilistic models as well as in inductive logic programming. Both subproblems are very hard and largely unsolved today. For instance, from an inductive logic programming perspective, a solution to hard problems such as multiple predicate learning and theory revision would be required. Therefore, there seems little hope of solving the full problem of structure learning from entailment in the near future. Nevertheless, for some special cases, one may still be able to develop some interesting techniques. For instance, Muggleton [2003] has extended the single-predicate learning setting of the Progol system to induce stochastic logic programs.

Because structure learning is so hard when learning from entailment, one may want to incorporate more information into the learning process. This can be realized using the learning from proofs setting, as proofs carry much more information about the target models than facts.

8.5.3 Learning from Proof Trees and Traces

Examples

When learning from proofs, the examples are proof trees or traces. For instance, for the stochastic logic program denoting the grammar, the proof tree of Fig. 8.10 could be a valid example. Similarly, when working with Web logs, as in the logical Markov model, one may actually observe a trace, which can easily be mapped onto a proof, as argued earlier.

One question that naturally arises with this setting, is to what extent it is reasonable to require examples in the form of proof trees. Although it seems clear that in most applications this is unrealistic, it should be emphasized that there exist specific situations in which proof trees occur quite naturally or in which the examples can be reformulated as such trees or traces, for instance, when dealing with logs of users traversing a web-site, logs of smart phones, or traces of a GPS device or robot. Also, in a bioinformatics setting, such structured sequences often occur naturally. Finally, within the natural language processing community, many large and annotated corpora exist that provide parse trees for a set of sentences, which correspond to proof trees in a logical setting.

Parameter Estimation

In order to address the parameter estimation problem in the learning from proofs setting it is again adequate to start with the simpler case of probabilistic context-free grammars. When learning probabilistic context-free grammars from parse trees, everything is fully observable and, hence, it is straightforward to estimate the parameters of the probabilistic grammar by frequency counting.

On the other hand, when considering stochastic logic or PRISM programs, this need not be the case. There are two cases to consider. First, if one observes derivations (both successful, that is, refutations, as well as failed), then all variables are observed and frequency counting will yield a maximum likelihood hypothesis. Second, if one only observes refutations, that is, succesful derivations, then the failed ones are unobserved. Thus, a form of failure adjusted maximisation needs to be employed. Actually, it turns out that Eq. 8.77 given for the FAM algorithm can easily be simplified to this case. The key simplification is that the (successful) proofs are now observed and hence one can replace the expected counts $ec_\lambda(C_i|e)$ of the number of times the clause C_i is used in the proof of example e with the actual counts $c(C_i|e)$. This is the only modification needed to adapt the FAM algorithm to the learning from proofs setting.

Structure Learning

We can get some inspiration from the work on probabilistic context-free grammars for structure learning. In particular, in natural language processing probabilistic context-free grammars have been successfully learned from large corpora of parse trees. This work is known under the name of *tree-bank grammars* [Charniak, 1996]. It turns out that learning the structure of a probabilistic context-free grammar is easy. The learner only needs to collect all the rules used in the given parse trees.

Example 8.44. Reconsider the parse trees shown in Fig. 4.6. The rules employed in the rightmost parse tree are:

$$\begin{array}{ll}
\mathsf{S} \rightarrow \mathsf{NP\ VP} & \mathsf{Art} \rightarrow \mathsf{the} \\
\mathsf{VP} \rightarrow \mathsf{Verb} & \mathsf{Noun} \rightarrow \mathsf{cat} \\
\mathsf{NP} \rightarrow \mathsf{Art\ Noun} & \mathsf{Verb} \rightarrow \mathsf{bites}
\end{array}$$

They directly follow from the parse tree.

Once all rules in the tree-bank have been collected, one only needs to apply the parameter estimation method sketched above to obtain the maximum likelihood hypothesis.

The question now is what changes when working with stochastic logic or PRISM programs. When applying the algorithm sketched above to proof trees obtained for a stochastic logic progam, such as those sketched in Fig. 8.10, one will obtain a set of *ground* clauses. As variables and unification are responsible for the expressiveness of stochastic logic programs, a way is needed to generalize the resulting set of ground clauses. One natural way of realizing this (cf. [De Raedt et al., 2005]) is to employ the least general generalization operator presented in Chapter 5. This is akin to techniques in grammatical inference for computing minimal generalizations of two grammar rules c_1 and c_2, and replacing them by their generalization. As usual, there is, however, a subtlety that needs to be taken into account in order to guarantee correct results. Indeed, it must be the case that if two rules in the stochastic logic program are generalized, the original proof trees are still legal proof trees for the generalized program. This imposes a logical constraint on the rules of the program, which may be violated by the unrestricted application of the lgg operator as the following example illustrates.

Example 8.45. Consider the clauses $h \leftarrow q$ and $h \leftarrow r$ and assume that the hypothesis H also contains the fact $q \leftarrow$. Then the proof tree for h in H is no longer a valid proof tree in H' consisting of $h \leftarrow$ and $q \leftarrow$.

The logical constraint therefore provides us with a powerful means to prune away uninteresting generalizations. Indeed, whenever a candidate lgg does not preserve the proof trees, it is overly general and should be pruned. A naive way of verifying this condition would be to compute the proofs for the generalized hypothesis H'. However, this would be computationally very expensive. Fortunately, it is possible to guarantee this condition much more efficiently. Indeed, it suffices to verify that

1. $\exists$ substitutions $\theta_1, \theta_2 : lgg(c_1, c_2)\theta_1 = c_1$ and $lgg(c_1, c_2)\theta_2 = c_2$
2. there is a one-to-one mapping from literals in $lgg(c_1, c_2)\theta_i$ to literals in c_i

If the first condition is violated, then there are literals in c_i that do not occur in $lgg(c_1, c_2)\theta_i$. These literals would be lost from the corresponding proofs (as sketched in the example), and so the proofs would not be preserved. If the second condition is violated, a standard theorem prover would derive different proofs (containing more children), and again the proofs would not be preserved. Furthermore, as is common in inductive logic programming, one can assume that all clauses in the target (and intermediate) logic programs are reduced w.r.t. θ-subsumption. The above conditions allow one to verify whether the proofs are preserved *after* computing the lgg. However, one needs to compute and consider only the lgg of two clauses if the multi-set of predicates occurring in these two clauses are identical.

The operation whereby two clauses are replaced by their least general generalization can now be used to heuristically search through a space of possible candidate models. They can be scored using, for instance, a maximum likelihood score penalized for model complexity.

Even though the techniques introduced for learning from proofs were largely illustrated using stochastic logic programs, it is, in principle, also possible to apply these ideas to the learning for PRISM programs and logical or relational Markov models. At the time of this writing, structure learning for PRISM programs has not yet been considered.

8.6 Relational Reinforcement Learning*

Having studied how to combine logic, probability and learning, we add, in this section, another dimension. It is concerned with taking actions to maximize one's utility. Learning which actions to take is studied in the area of reinforcement learning [Sutton and Barto, 1998]. This section provides a brief introduction to relational reinforcement learning, which studies reinforcement learning using logical and relational representations. We first introduce Markov Decision Processes and their use for learning before upgrading them to the use of relational representations. Because (relational) reinforcement learning is an active field of research with many different approaches and results, we are only able to define the problem and sketch some of the basic approaches. Further information about reinforcement learning can be found in [Sutton and Barto, 1998, Russell and Norvig, 2004, Mitchell, 1997] and about its relational upgrade in [van Otterlo, 2008, Sanner and Kersting, 2009, to appear].

8.6.1 Markov Decision Processes

The underlying model used in reinforcement learning is that of Markov Decision Processes. Formally, a *Markov Decision Process* consists of

- a set of possible *states* S,
- a set of *actions* A,

- a *transition* function $T : S \times A \times S \longrightarrow [0,1]$ such that $T(s,a,\cdot)$ is a probability distribution over S for any $s \in S$ and $a \in A$, that is, $\sum_{s' \in S} T(s,a,s') = 1$, and
- a *reward* function $R : S \times A \longrightarrow \mathbb{R}$.

In a Markov Decision Process the agent is at every point in time t in a state s_t, has to select an action $a_t \in A(s_t)$, the set of possible actions in s_t, and then moves to the next state s_{t+1} with probability $T(s_t, a_t, s_{t+1})$ obtaining the reward $r_t = R(s_t, a_t)$. To this end, the agent employs a policy $\pi : S \longrightarrow A$, which specifies in each state the action $\pi(s)$ the agent will execute. Note that, in general, policies may be stochastic, but for simplicity we shall assume that they are deterministic in this book.

Example 8.46. To illustrate Markov Decision Processes, let us consider a simple world in which we have exactly two blocks, named a and b. Thus we have three possible configurations

$$\begin{array}{|c|}\hline \mathsf{a}\\\hline \mathsf{b}\\\hline\end{array} \quad \text{and} \quad \begin{array}{|c|}\hline \mathsf{b}\\\hline \mathsf{a}\\\hline\end{array} \quad \text{and} \quad \boxed{\mathsf{a}}\quad\boxed{\mathsf{b}},$$

where the order of the stacks does not matter. In the blocks world, one can move a block if it is clear from one position to another. This means that we have the following possible actions for the states

$$A(\begin{array}{|c|}\hline \mathsf{a}\\\hline \mathsf{b}\\\hline\end{array}) = \{\mathsf{move\_a\_to\_floor}\}$$

$$A(\begin{array}{|c|}\hline \mathsf{b}\\\hline \mathsf{a}\\\hline\end{array}) = \{\mathsf{move\_b\_to\_floor}\}$$

$$A(\boxed{\mathsf{a}}\quad\boxed{\mathsf{b}}) = \{\mathsf{move\_a\_to\_b; move\_b\_to\_a}\}.$$

Assume that the actions succeed with probability 0.9 and fail with probability 0.1, where failure implies that executing the state is left unchanged, and that the reward function yields 10 only when the action leads to the next state $\begin{array}{|c|}\hline \mathsf{a}\\\hline \mathsf{b}\\\hline\end{array}$ and 0 otherwise. This reward function implicitly encodes that the goal is to enter this state as often as possible. Because encoding the reward function in terms of the state resulting from the action is quite convenient and understandable, we shall from now on encode rewards functions in this way.

The goal of the agent is to maximize the *expected return*, that is, the expected discounted sum of rewards over time:

$$E\Big[\sum_{t=0}^{\infty} \gamma^t r_t\Big], \tag{8.78}$$

where r_t is the reward received at time t and $0 \leqslant \gamma \leqslant 1$ is a discount factor, which implies that immediate rewards are more important than future ones.

The *state-value function* $V^\pi : S \to \mathbb{R}$ specifies for each state s the expected return $V^\pi(s)$ the agent will receive when starting in s and executing policy π. Formally, V^π is defined as

$$V^\pi(s) = E_\pi\Big[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\Big], \tag{8.79}$$

that is, the expected return when starting in s and selecting actions according to π. Similarly, the *action-value function* $Q^\pi : S \times A \to \mathbb{R}$ specifies for each state action pair (s, a) the expected return $Q^\pi(s, a)$ the agent will receive when starting in state s, executing action a and then following the policy π. The two value functions of a policy π are related by the equation

$$V^\pi(s) = \max_a Q^\pi(s, a), \tag{8.80}$$

which specifies that the state-value function yields the value of the best action in that state.

The task of the agent is then to compute an *optimal policy* π^*. A policy π^* is optimal if and only if for all other policies π' and for all states $s \in S$:

$$V^{\pi^*}(s) \geqslant V^{\pi'}(s). \tag{8.81}$$

Following the optimal policy yields the highest expected returns. The Bellman optimality equations characterize the state-value function V^* of an optimal policy π^*:

$$V^*(s) = \max_a \Big[R(s, a) + \gamma \sum_{s'} T(s, a, s') V^*(s')\Big]. \tag{8.82}$$

A similar equation holds for the action-value function Q^*:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q^*(s', a'). \tag{8.83}$$

From the optimal value functions, the optimal policy can be derived because

$$\pi^*(s) = \arg\max_a Q^*(s, a) \tag{8.84}$$

$$= \arg\max_a \Big[R(s, a) + \gamma \sum_{s'} T(s, a, s') V^*(s')\Big]. \tag{8.85}$$

This equation shows a clear advantage of the Q^* function: the transition and reward functions need not be known to compute the optimal policy.

Example 8.47. Continuing the blocks world example, the Bellman optimality equation for the state

$$\boxed{\mathsf{a}}\ \boxed{\mathsf{b}}$$

states

$$\begin{aligned} V^*(\boxed{\mathsf{a}}\ \boxed{\mathsf{b}}) = \max\Big\{ \ & 10+\gamma\big(0.9V^*(\begin{array}{|c|}\hline \mathsf{a}\\\hline \mathsf{b}\\\hline\end{array}) + 0.1V^*(\boxed{\mathsf{a}}\ \boxed{\mathsf{b}})\big);\\ & 0+\gamma\big(0.9V^*(\begin{array}{|c|}\hline \mathsf{b}\\\hline \mathsf{a}\\\hline\end{array}) + 0.1V^*(\boxed{\mathsf{a}}\ \boxed{\mathsf{b}})\big)\Big\}. \end{aligned}$$

It is possible to improve upon a non-optimal policy π using *policy improvement*, which computes an improved policy π's:

$$\pi'(s) = max_a Q^{\pi}(s,a). \tag{8.86}$$

Because the policy π' greedily selects actions, it improves upon π, that is, for all states $s \in S : V^{\pi'}(s) \geqslant V^{\pi}(s)$; see [Sutton and Barto, 1998] Sect. 4.2 for a proof. If the policy π was already optimal, then $\pi' = \pi$ and the Bellman optimality equations hold. Policy improvement is typically used in a process called generalized policy iteration, in which one starts from a given policy, computes its value function, uses the value function to improve the policy, and iterates; see Sutton and Barto [1998].

8.6.2 Solving Markov Decision Processes

Numerous approaches exist for solving Markov Decision Processes. They are primarily distinguished by whether they are model-based, that is, whether they use a model or not. A model, in this context, refers to the transition and reward functions T and R.

Model-based solution techniques for Markov Decision Processes are typically based on dynamic programming. As one example of this popular class of techniques, we consider the *value iteration* algorithm sketched in Algo. 8.3. This algorithm initializes the Q-function arbitrarily (here, by setting all the values to 0), and then repeatedly updates the Q-function for all state action pairs. The update rule that is employed is essentially the Bellman optimality equation of Eq. 8.83. This process continues until the values converge and the updates become sufficiently small. The value function Q computed by the value iteration algorithm will converge in the limit towards the optimal value function Q^*. Therefore, the Q-function computed by the algorithm can be used as an encoding of the optimal policy π^*.

In *model-free* approaches to reinforcement learning, the functions T and R are unknown to the agent, but the task essentially remains the same: computing (an approximation of) the optimal policy. However, in this case the

Algorithm 8.3 Value iteration

```
initialize $V(s) := 0$ for all $s \in S$
repeat
  $\mathit{\Delta} := 0$
  for all $s \in S$ do
    $v := V(s)$
    for each $a \in A(s)$ do
      $Q(s,a) := R(s,a) + \gamma \sum_{s'} T(s,a,s')V(s')$
    end for
    $V(s) := max_{a \in A(s)} Q(s,a)$
    $\mathit{\Delta} := \max(\mathit{\Delta}, |v - V(s)|)$
  end for
until $\mathit{\Delta} < \sigma$
```

agent will gather evidence about these functions through interaction with its environment, which means that the agent will perform actions and learn from their outcomes. One line of model-free approaches learns a model, that is, an approximation of the transition and reward functions T and R, from these interactions. To this end, it keeps track of statistics for each possible transition $T(s,a,t)$ and reward $R(s,a)$. If all possible state-action pairs are visited a sufficient number of times, the estimated model will approximate the true model. Model-based techniques, such as value iteration, can then be applied to compute the optimal policy.

Another type of model-free approach is based on Monte Carlo methods. This type of method estimates the value functions directly. This is only possible when the environment is *episodic*, that is, sequences of actions are guaranteed to reach a terminal state. Episodic environments naturally occur, for instance, in game playing. In other cases, the environment can be made episodic. For instance, in the blocks world example, if the goal is to reach a particular state, say $\boxed{\mathsf{a}}\ \boxed{\mathsf{b}}$, this state could be made *absorbing*, which means that no matter what action is executed in that state, the agent remains in the state. In episodic environments, Monte Carlo methods for learning a Q^π function for a given policy π compute $Q^\pi(s,a)$ as the average of all returns received when executing action a in state s and following the policy π afterward.

The final and most interesting class of model-free approaches from a machine learning perspective is that of *temporal difference learning.* Characteristic for temporal difference learning is that it reestimates its current value function for a particular state (or state-action) using estimates for other states. This realizes a kind of bootstrapping. One of the most prominent examples in this category is Q-learning. Q-learning employs the following update rule to reestimate $Q(s,a)$ after executing action a in state s, receiving the reward r and observing the next state s':

$$Q(s,a) := Q(s,a) + \alpha\Big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big), \tag{8.87}$$

where $0 < \alpha < 1$ is the learning rate. The higher the learning rate, the more important the new experience is. Therefore, the value of α is gradually decreased over time. This leads to the famous Q learning algorithm; see Algo. 8.4. This algorithm will, under certain conditions, converge to the optimal Q^* function. While learning, the agent has to select and execute actions. After each action, it performs an update of the Q-function according to the above formula. The formulation of Q-learning in Algo. 8.4 assumes an episodic setting.

Recall that the agent's goal is to learn how to maximize the expected future return, and that, while learning, the agent has to take action in its environement. The strategy for selecting the next action is important in this regard. There is a typical trade-off between *exploration* and *exploitation* in reinforcement learning. To maximize the expected return, it is tempting to select the actions greedily, according to the current value functions. This is called exploitation. However, if the approximations of the value functiions are still imperfect, greedy action selection might lead to suboptimal behavior. Therefore, the agent should also explore its environment, that is, ensure that it visits all parts of the search-space a sufficient number of times so that good approximations can be learned. Because exploration and exploitation are conflicting goals, one has to find the right balance between these two ways of selecting actions. One way of realizing this employs an ϵ-greedy method to select the next action to execute. For a given Q-function, this method selects with probability $1-\epsilon$ the best action according to Q, that is, $\arg\max_a Q(s,a)$, and with probability ϵ a random action $a \in A(s)$.

Algorithm 8.4 Q-learning

```
initialize $Q(s,a) := 0$ for all $s \in S, a \in A(s)$
for each episode do
  let $s$ be the starting state
  repeat
    choose an action $a \in A(s)$ and execute it
    observe the new state $s'$ and reward $r$
    $Q(s,a) := \alpha\Big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big)$
    $s := s'$
  until $s$ is a terminal state
end for
```

One problem with the above formulation of Markov Decision Processes and their solution techniques is that the value functions are represented in explicit tabular form. For each state or state-action pair, the corresponding value must be stored in some kind of table. Even in simple worlds, the state space rapidly becomes prohibitively large, and the number of transitions to take into account is even larger. Therefore, representing the value functions in tabular form is impractical. What is needed is a more compact representation for the value function, for instance, using the languages of the representation hierarchy sketched in Chapter 4. The simplest of these is a propositional logic or attribute-value representation. States then correspond to item-sets or propositional interpretations. This enables us, in turn, to represent the value function by a model, such as a decision tree, a linear equation, or a neural network. To illustrate this idea, consider the Q-learning procedure sketched in Algo. 8.4 and assume that we are using a neural network to represent the Q-function. The value of $Q(s, a)$ for state s and action a is then the value obtained by using the example consisting of the interpretation describing the state s together with the action a as input for the neural network. The output of the network is the predicted Q-value. To work with this representation the Q-learning algorithm needs to be adapted at the point where the Q-function is updated. Rather than computing the new value for $Q(s, a)$ and storing it in the table for Q, the algorithm will now generate an example for the neural network learning algorithm. This example consists of the description (s, a) together with the desired target value $Q(s, a)$. The neural network will then update its weights to accommodate this change. Neural networks are often employed as function approximators because they are incremental although other paradigms can be used as well. Because the function learned by the function approximator is only an approximation of the Q-function in tabular form, the algorithm will be efficient. However, depending on the approximator used, it may also lose its convergence properties. In practice, it often works well, as shown by, for instance, the famous TD-gammon player by Tesauro [1995]. Rather than illustrating this technique using propositional representations, we will use relational representations. Before doing so, we introduce the concept of a relational Markov Decision Process.

8.6.3 Relational Markov Decision Processes

Relational Markov Decision Processes are an upgrade aimed at using relational representations in which states correspond to Herbrand interpretations like in planning [Russell and Norvig, 2004]. The advantage of relational representations is, as usual, that they enable one to work with a variable number of entities and the relationships amongst them. For instance, while the simple blocks world example shown above consisted of only two blocks, artificial intelligence typically studies such worlds with an arbitrary number of blocks and relations. Our goal is now to adapt relational reinforcement learning to work with such worlds. This goal is in line with the probabilistic logic learning approach pursued in this chapter. The key question then is how to represent such Relational Markov Decision Processes? To address this question, it is useful to start from existing formalisms employed within the domain of planning, and to investigate how they can be upgraded using the methodology of Chapter 6.

To upgrade the definition of Markov Decision Processes to cope with relational states and actions, we need to make the definitions of the transition and reward functions relational. It is convenient to start from one of the popular planning formalisms, Strips by Fikes and Nilsson [1971], to realize this. In Strips, each action is defined using a precondition and a set of effects, where the precondition is specified using a conjunction of atoms and the effects specify which atoms to add and which ones to delete from the state. More formally, an action is a finite set of actions rules of the form

$$p_i : A :: H_i \leftarrow B, \tag{8.88}$$

where B and the H_i are conjunctions of atoms, the p_i satisfy $\sum_i p_i = 1$, and it is required that all variables occurring in the H_i occur also in B. The actions define the transition function. If the conjunction B subsumes an interpretation I with substitution θ, there is a transition with p_i to the state $[I - B\theta] \cup H_i\theta$. Thus the atoms in $B\theta$ are removed from the state and those in $H_i\theta$ added to the state. This corresponds to the add and delete lists of the Strips formalism. In what follows, we employ the notion of OI-subsumption to avoid having to write many inequalities between the variables; cf. Sect. 5.5.1.

For instance, we can specify the action $\mathsf{move(X,Y,Z)}$, which moves a block X to a block Y from a block Z, using

$$\begin{array}{l} \mathsf{0.9 : move(X,Y,Z) :: on(X,Y), cl(X), cl(Z) \leftarrow cl(X), cl(Y), on(X,Z)} \\ \mathsf{0.1 : move(X,Y,Z) :: cl(X), cl(Y), on(X,Z) \leftarrow cl(X), cl(Y), on(X,Z)} \end{array}$$

where the object identity assumption implies that the variables X, Y and Z have to bind to different constants. Applied to the state

$$\{\mathsf{on(a,c), cl(b), cl(a)}\}$$

the action $\mathsf{moves(a,b,c)}$ yields with probability 0.9 the state

$$\{\mathsf{on(a,b), cl(c), cl(a)}\}$$

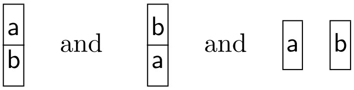
To represent the reward function, a logical decision tree or a set of rules can be applied. For instance, the reward function could be defined using the decision list

$$\begin{array}{l} \mathsf{reward(10) \leftarrow cl(a), !.} \\ \mathsf{reward(0) \leftarrow} \end{array}$$

which states that the reward received is 10 if in the state just entered block a is clear, and 0 otherwise.

The semantics of a Relational Markov Decision Process can be defined following the principles of knowledge-based model construction, which we employed while analyzing probabilistic logics such as Bayesian Logic Programs and Markov logic. The Relational Markov Decision Process consists of the transition rules, the reward function and a set of constants determining the objects in the domain. The constants and the relations occurring in the transition rules determine the state space. Each Herbrand interpretation over these relations and constants is a possible state. The actions, rewards and transitions between the states are defined by the transition rules and the reward function, and determine the possible transitions between these “grounded” states. It can be proved that for any set of constants, a Markov Decision Process is constructed in this way; see, for instance, [Kersting and De Raedt, 2003] for more details. By varying the set of constants, different state spaces and Markov Decision Processes are obtained. Observe also that the STRIPS formalism supports parameter tying.

Example 8.48. If we use as constants a and b we obtain the STRIPS representation of our initial Markov Decision Process. The states



are not directly representable using the relational Markov Decision Process just defined. The reason is that we do not explicitly take into account the floor of the blocks. Taking into account a unique floor requires the adaptation of the transition rules, and the introduction of new transition rules. We use simpler state descriptions, in which it is assumed that there are a fixed number of places (like f1 and f2) where blocks can be placed. This is the view where f1 and f2 are treated as floor blocks. It leads to the following possible states for the blocks a and b and places f1 and f2

$$\begin{array}{ll}
\{\mathsf{on(a,b)}, \mathsf{on(b,f1)}, \mathsf{cl(f2)}, \mathsf{cl(a)}\} & \{\mathsf{on(a,b)}, \mathsf{on(b,f2)}, \mathsf{cl(f1)}, \mathsf{cl(a)}\} \\
\{\mathsf{on(b,a)}, \mathsf{on(a,f1)}, \mathsf{cl(f2)}, \mathsf{cl(b)}\} & \{\mathsf{on(b,a)}, \mathsf{on(a,f2)}, \mathsf{cl(f1)}, \mathsf{cl(b)}\} \\
\{\mathsf{cl(a)}, \mathsf{cl(b)}, \mathsf{on(a,f1)}, \mathsf{on(b,f2)}\} & \{\mathsf{cl(a)}, \mathsf{cl(b)}, \mathsf{on(a,f2)}, \mathsf{on(b,f1)}\}
\end{array}$$

Exercise 8.49. Can you adapt the transition rules for the case where there is a unique floor on which one can place an arbitrary number of stacks? The transition should support the representation of the blocks world that we introduced first.

8.6.4 Solving Relational Markov Decision Processes

Now that Relational Markov Decision Processes have been defined, we look into possible approaches for solving them. More specifically, we investigate how both the value iteration and the Q-learning approaches can be adapted. We start with the simplest of these two, that is, Q-learning. Then we present one of its variants, abstract Q-learning, before discussing relational value iteration.

Q-Learning

To adapt Q-learning to work with relational representations, all that is needed is a relational function approximator to learn the Q-function. This can be realized by using, for instance, a logical regression tree for representing the Q-function. The decision list resulting from the tree might, for the blocks world, look like

```
qvalue(10) ← cl(a), !.
qvalue(9.0) ← on(X, a), cl(X), cl(F1), move(X, F1, a), !.
qvalue(8.1) ← on(X, Y), on(Y, a), cl(X), cl(F1), cl(F2), move(X, F1, Y), !.
qvalue(7.2) ←
```

This definition of the Q-function can be applied to a state and an action represented as a set of ground facts. It then returns the Q-value.

Although the induction of such regression trees proceeds essentially along the same lines as for other decision trees, there are some subtle difficulties that occur in a reinforcement learning setting. First, the algorithm should learn online, that is, it should process the examples as they occur and incrementally adapt the current hypothesis. Second, the examples provided to the regression algorithm might change over time. As the policy improves over time the values associated to a state action pair may change as well, which implies that later examples might invalidate former ones, and hence some way of forgetting is necessary. Further subtleties arise in the relational case. Even though the logical regression tree will predict a value for each state, the predicted value will be independent of the number of objects in the state, and this causes problems. The reason is that Q-values implicitly encode the distance to the goal. Let us illustrate this using an example.

Example 8.50. Assume the reward function returns 10 when entering a state where on(a, b) and 0 otherwise, that such states are absorbing, that we have n blocks, and more than four places. Thus, the goal is to stack a onto b, and, due to the absorbing states, the task is episodic. The optimal policy for this case first clears a and b, that is, it repeatedly removes the top blocks from the stacks to which a and b belong, and then moves a onto b. The state-value function for this policy is graphically depicted in Fig. 8.12 (for $\gamma = 0.9$). The figure shows clearly that the higher the number of needed actions, the lower the utility, and hence that the utility is a function of the distance to the goal. Nevertheless, the optimal strategy can be defined independent of the distance to the goal. It simply removes the next block from the stack above a or b until a and b are clear, after which a is moved on top of b.

The example indicates that policies are often easier to encode than value functions, which explains why some approaches try to learn and represent policies directly rather than through their value functions [Fern et al., 2007]. It is also possible to learn an explicit policy starting from the Q-function [Džeroski et al., 2001]. This can be realized by using the Q-function to generate examples consisting of state-action pairs. Positive examples are those for which the action associated with the state is optimal; the negative examples contain the non-optimal actions.

Abstract Q-Learning**

In the Q-learning approach just presented the learner generates the definition of the Q-function from a set of examples in the form of state-action-value

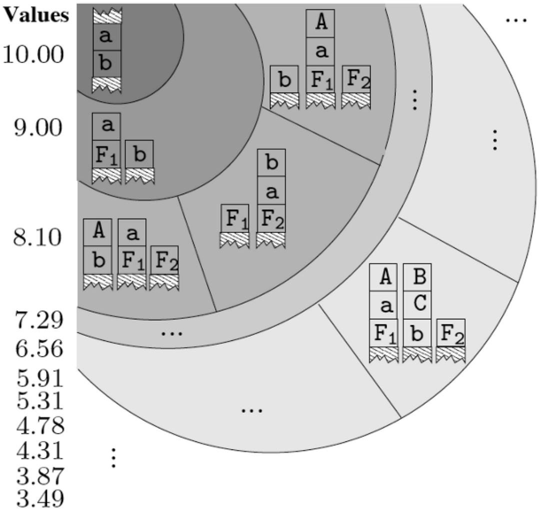


Fig. 8.12. Parts of the value function for on(a, b). Reproduced from [Kersting et al., 2004]. Uppercase characters denote logical variables, and the Fi can bind to blocks or to positions

tuples, and dynamically partitions the set of possible states. For instance, the above definition of the qvalue/1 predicate employs four partitions. These partitions are described by a kind of *abstract* state, that is, a logical condition, which matches several real states. For instance, the first rule matches all states in which cl(a) is true, the second one all states in which cl(a) is not true, but on(X, a), cl(X) is true, etc. The relational Q-learning approach sketched above thus needs to solve two tasks: finding the right partition and learning the right values for the corresponding abstract state-pairs. This is a hard task, and the question arises as to whether it can be simplified, for instance, by providing additional knowledge in the form of the partition to the learner. The answer to this question is affirmative. It is actually easy to devise a variant of the relational Q-learning approach that learns at an abstract level; cf. [van Otterlo, 2008, Kersting and De Raedt, 2003]. The abstract Q-learning algorithm starts from a partition of the state space in the form of a decision list of abstract state-action pairs $((S_1, A_1), \cdots, (S_n, A_n))$, where we assume that all possible abstract actions are listed for all abstract states. Each abstract state S_i is a conjunctive query, and each abstract action A_i contains a possibly variablized action. As one example of such an abstract state-action pair consider the condition of the third rule for qvalue/1 above, that is,

$$\mathsf{on(X,Y), on(Y,a), cl(X), cl(F1), cl(F2) - move(X,F1,Y)}$$

The abstract state matches any real state where there is a block X on a, X is clear and so are F1 and F2, and the action moves X from Y to F1.

The abstract Q-learning algorithm now turns the decision list into the definition of the qvalue/1 predicate, and then applies Q-learning using the qvalue/1 predicate as its table of state-action pairs. This means that every time a concrete state-action pair (s, a) is encountered, a Q-value q is computed using the current definition of qvalue/1, and then the abstract Q-function, that is, the definition of qvalue/1 is updated for the abstract state-action pair to which (s, a) belongs. It can be shown that this algorithm will converge to the optimal policy at the abstract level. However, the optimal policy at the abstract level does not necessarily coincide with the optimal policy of the underlying Relational Markov Decision Process as the following example shows.

Example 8.51. ** (Due to Robert Givan, closely following [Kersting and De Raedt, 2003].) Consider the following transition rules

$$\begin{array}{l} \mathsf{1.0 : a :: q} \leftarrow \mathsf{p, q} \\ \mathsf{1.0 : a ::} \emptyset \leftarrow \mathsf{p} \\ \mathsf{1.0 : a :: p} \leftarrow \end{array}$$

The resulting Markov Decision process has the states $\{\}$, $\{\mathsf{p}\}$, $\{\mathsf{q}\}$, and $\{\mathsf{p}, \mathsf{q}\}$ and as only possible action, a, which is deterministic. Let us also assume that the reward is 1.0 if the action a is executed in state $\{\mathsf{p}\}$ and 0 otherwise.

Let the abstraction level now consist of the decision list

$$\big((\mathsf{p}, \mathsf{a}), (\mathsf{q}, \mathsf{a}), (\mathsf{true}, \mathsf{a})\big)$$

This abstraction level results in the following partition of the states

$$\{\{\mathsf{p}\}, \{\mathsf{p}, \mathsf{q}\}\} \text{ and } \{\{\mathsf{q}\}\} \text{ and } \{\{\}\}$$

The abstract Markov Decision Process will now assign the same probabilities and rewards to the transitions from the second partition to the first one and from the third to the first one. Thus the abstract Markov Decision Process seems to have a non-Markovian nature. As a consequence the values for the second and third partition in the abstract Markov Decision Process are the same as the next state is the same, that is, the first partition. This is not the case for the (grounded) Relational Markov Decision Process, as the reader may want to verify.

The example shows that learning at the abstract level may result in losing information about the original problem, resulting in a kind of partially observable Markov Decision Process, which requires more advanced solution techniques. This shows that relational reinforcement learning is a hard problem, as it is important to get the abstraction level right. The relational Q-learning approach offers no guarantees in this respect.

Value Iteration**

Value iteration is a model-based technique, and therefore the model is available. The model of the Relational Markov Decision Process consists of the definitions of the transition and rewards functions. We now want to adapt the value iteration algorithm for use with relational representations. The key difficulty lies in the upgrading of the the update rule, based on the Bellman optimality equation Eq. 8.83. The update rule works backwards, that is, to update the value of a state it employs the values of the possible predecessor states. While the predecessors are explicitly encoded in a normal Markov Decision Process, they are now only implicitly available.

Example 8.52. For illustration, consider that we are in the goal state where $\mathsf{on}(\mathsf{a},\mathsf{b})$ is true, and assume that we want to update the value function. Then we need to update the abstract state, where $\mathsf{on}(\mathsf{a},\mathsf{b})$ is true, to identify all states in which executing the action $\mathsf{move}(\mathsf{X},\mathsf{Y},\mathsf{Z})$ leads to $\mathsf{on}(\mathsf{a},\mathsf{b})$ being true. In other words, we need to find the *weakest preconditions* under which executing the action $\mathsf{move}(\mathsf{X},\mathsf{Y},\mathsf{Z})$ leads to $\mathsf{on}(\mathsf{a},\mathsf{b})$. This is needed because the Bellman optimality equation defines the value of a state in terms of those of its predecessors. The problem of computing the weakest preconditions of an (abstract) state is known as *regression* in the planning literature; cf. [Russell and Norvig, 2004]. There are two cases to consider for computing the weakest preconditions for our example. First, assuming that the $\mathsf{move}(\mathsf{X},\mathsf{Y},\mathsf{Z})$ action caused $\mathsf{on}(\mathsf{a},\mathsf{b})$ to be true, implies that in the predecessor state the condition $\mathsf{cl}(\mathsf{a}),\mathsf{cl}(\mathsf{b}),\mathsf{on}(\mathsf{a},\mathsf{Z})$ must have been true $\mathsf{X}=\mathsf{a}$ and $\mathsf{Y}=\mathsf{b}$. Second, assuming that the move action did not cause $\mathsf{on}(\mathsf{a},\mathsf{b})$ to become true, the predecessor state must have satisfied $\mathsf{cl}(\mathsf{X}),\mathsf{cl}(\mathsf{Y}),\mathsf{on}(\mathsf{X},\mathsf{Z}),\mathsf{on}(\mathsf{a},\mathsf{b})$. Please note that we are still using OI-subsumption, which implies that $\mathsf{X}\neq\mathsf{Y}$, $\mathsf{X}\neq\mathsf{a}$, etc.

The example shows that a complex logical operation is needed to compute the abstract predecessors of an abstract state, and this makes adapting the value iteration algorithm quite involved, which explains why we refer to [Boutilier et al., 2001, Kersting et al., 2004, Wang et al., 2008] for more details. The idea, however, is that the value iteration algorithm takes as initial value function the reward function, and then propagates the conditions occurring in the rules defining the reward function backwards using regression. This process is then repeated on the resulting value function. Additional complications arise because the conditions of the rules defining the value function may be overlapping, and some rules might be redundant, which explains why some approaches [Kersting et al., 2004] employ further operations on these rules sets to compress and simplify the definition of the value function. Figure 8.12 shows the state value function that is the result of performing a 10-step relational value iteration algorithm. The reader should also realize that, when the potential number of blocks is unbounded, value iteration will never terminate as an infinite number of abstract states would be required.

To summarize, relational reinforcement learning aims at upgrading reinforcement learning techniques and principles to use logical and relational representations. Two main approaches exist: model-based and model-free reinforcement learning. The model-based approach is closely related to decision-theoretic planning, and borrows regression techniques to compute value functions. The model-free approach uses relational function approximators to learn value functions, or assume that an abstraction level in the form of a partition of the state space is given. We have also seen that many challenges in relational reinforcement learning remain open, which explains why it is an active field of research; see [Driessens et al., 2004, 2005] for two collections of recent papers on this topic.

8.7 Conclusions

We have provided an overview of the new and exciting area of probabilistic logic learning in this chapter. It combines principles of probabilistic reasoning, logical representation and statistical learning into a coherent whole. The techniques of probabilistic logic learning were analyzed starting from a logical and relational learning perspective. This turned out to be useful for obtaining an appreciation of the differences and similarities among the various frameworks and formalisms that have been contributed to date. In particular, the distinction between a model- and a proof-theoretic view was used for clarifying the relation among the logical upgrades of Bayesian networks (such as probabilistic relational models, Bayesian logic programs, etc.) and grammars (such as stochastic logic programs and PRISM). This distinction is not only relevant from a knowledge representation perspective but also from a machine learning perspective, because one typically learns from interpretations in the model-theoretic approaches, from entailment in the proof-theoretic ones, and from traces in the intermediate ones (such as RMMs and LOHMMs). Furthermore, principles of both statistical learning and logical and relational learning can be employed for learning the parameters and the structure of probabilistic logics. We have then also shown how important ideas such as knowledge-based model construction can be applied in a reinforcement learning setting, and how various techniques from reinforcement learning can be upgraded to relational representations.

8.8 Bibliographic Notes

This chapter is largely based on the survey papers [De Raedt and Kersting, 2003, De Raedt and Kersting, 2004], and, for the introduction to the probabilistic techniques, on the expositions by Russell and Norvig [2004], and Manning and Schütze [1999]. Two collections of recent research papers on statistical relational learning and probabilistic logic learning provide a detailed account of the state-of-the-art in the field; cf. [Getoor and Taskar, 2007, De Raedt et al., 2008]. Background material on graphical and probabilistic models can be found in [Russell and Norvig, 2004, Jensen, 2001].

Probabilistic logics were originally mostly studied from a knowledge representation and reasoning perspective. Many different representations and inference algorithms were developed from this perspective in the late 1980s and 1990s; see, for instance, [Nilsson, 1986, Bacchus, 1990, Poole, 1993b, Dantsin, 1992, Haddawy, 1994, Ngo and Haddawy, 1995, Muggleton, 1996, Jaeger, 1997, Koller and Pfeffer, 1997, Lakshmanan et al., 1997]. The interest in learning such logics started soon afterwards with work on PRISM [Sato, 1995, Sato and Kameya, 1997] and on probabilistic relational models [Friedman et al., 1999, Getoor et al., 2001a,b]. Especially the probabilistic relational models received a lot of attention in the artificial intelligence community, and soon afterwards the first workshops on statistical relational learning were organized [Getoor and Jensen, 2003, 2000]. Around that time, many further representations were conceived, including Bayesian Logic Programs [Kersting and De Raedt, 2007], CLP(BN) [Costa et al., 2003a], IBAL [Pfeffer, 2007], Markov logic [Richardson and Domingos, 2006], and CP-logic [Vennekens et al., 2006], and several learning techniques were developed, such as [Cussens, 2001, Kok and Domingos, 2005, Jaeger, 2007, Singla and Domingos, 2005, Taskar et al., 2001]. Furthermore, theoretical issues such as expressiveness of the representation languages Jaeger [2008] and lifted inference were being investigated [Poole, 2003, de Salvo Braz et al., 2007]. Also, some exciting applications of probabilistic logic learning have been contributed; see for instance [Getoor et al., 2004, Anderson et al., 2002, Bhattacharya et al., 2006, Fern et al., 2002, Liao et al., 2005, King et al., 2004, Limketkai et al., 2005].

Introductions to reinforcement learning can be found in [Russell and Norvig, 2004, Sutton and Barto, 1998]. The use of relational representations was introduced in reinforcement learning by Džeroski et al. [2001], and investigated from a decision theoretic perspective (value iteration) by Boutilier et al. [2001]. Since then several works have been devoted to these topics, including [Gärtner et al., 2003, Driessens et al., 2001, Wang et al., 2008, Kersting et al., 2004, Sanner and Boutilier, 2005]; an extensive overview of these developments can be found in [van Otterlo, 2008].

9

Kernels and Distances for Structured Data

Many contemporary machine learning approaches employ kernels or distance metrics, and in recent years there has been a significant interest in developing kernels and distance metrics for structured and relational data. This chapter provides an introduction to these developments, starting with a brief review of basic kernel- and distance-based machine learning algorithms, and then providing an introduction to kernels and distance measures for structured data, such as vectors, sets, strings, trees, atoms and graphs. While doing so, we focus on the principles used to develop kernels and distance measures rather than on the many variants and variations that exist today. At the same time, an attempt is made to draw parallels between the way kernels and distance measures for structured data are designed. Convolution and decomposition can be used to upgrade kernels for simpler data structures to more complex ones; cf. [Haussler, 1999]. For distances, decomposition is important, as is the relationship between the generality relation and the size of the data; cf. [De Raedt and Ramon, 2008].

9.1 A Simple Kernel and Distance

Distance- and kernel-based learning is typically addressed in the traditional function approximation settings of classification and regression introduced in Sect. 3.3. Both distances and kernels provide information about the similarity amongst pairs of instances. Indeed, a kernel function can directly be interpreted as measuring some kind of similarity, and the distance amongst two objects is inversely related to their similarity.

To introduce distances and kernels, we will in this section focus on the familiar Euclidean space, that is $\mathcal{L}_e = \mathbb{R}^d$. In this space, the instances correspond to vectors, for which we use the notation $\mathbf{x}$ or $(x_1, \cdots, x_d)$, with components x_i.

In the next section, we will then discuss the use of kernels and distances in machine learning using standard distance- and kernel-based approaches, such as the k-nearest neighbor and support vector machine algorithms. Afterwards, we shall investigate how distances and kernels for structured data can be constructed, in particular for strings, sets and graph-structured data. While doing so, we shall focus on the underlying principles rather than on the many specific distance measures and kernels that exist today.

A natural distance measure for $\mathbb{R}^d$ is the *Euclidean distance*:

$$d_e(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (x_i - y_i)^2} \tag{9.1}$$

The simplest kernel for $\mathbb{R}^n$ is the *inner product*:

$$< \mathbf{x}, \mathbf{y} >= \sum_i x_i y_i \tag{9.2}$$

The *norm* $\|\mathbf{x}\|$ of a vector $\mathbf{x}$ denotes its size. It can be computed from the inner product as follows:

$$\|\mathbf{x}\| = \sqrt{< \mathbf{x}, \mathbf{x} >} \tag{9.3}$$

Furthermore, the inner product $< \mathbf{x}, \mathbf{y} >$ of two vectors in $\mathbb{R}^d$ can be written as

$$< \mathbf{x}, \mathbf{y} >= \|\mathbf{x}\| \; \|\mathbf{y}\| cos(\alpha) \tag{9.4}$$

where α is the angle between the two vectors $\mathbf{x}$ and $\mathbf{y}$, which allows us to interpret the inner product in geometric terms. The equation states that when the two vectors are normalized (that is, have length 1), the inner product computes the cosine of the angle between the two vectors. It then follows that when $\alpha = 0$, $< \mathbf{x}, \mathbf{y} > = 1$, and when the two vectors are orthogonal, the inner product yields the value 0. This indicates that the inner product measures a kind of similarity.

Even though we will formally introduce kernels and distance measures later an interesting relationship amongst them can already be stated. Any (positive definite) kernel K induces a distance metric d_K as follows:

$$d_K(\mathbf{x}, \mathbf{y}) = \sqrt{K(\mathbf{x}, \mathbf{x}) - 2K(\mathbf{x}, \mathbf{y}) + K(\mathbf{y}, \mathbf{y})} \tag{9.5}$$

It is instructive to verify this statement using the kernel $K(\mathbf{x}, \mathbf{y}) =< \mathbf{x}, \mathbf{y} >$.

Exercise 9.1. Show that the distance d_e can be obtained from Eq. 9.5 using the inner product as a kernel, that is:

$$d_e(\mathbf{x}, \mathbf{y}) = \sqrt{< \mathbf{x}, \mathbf{x} > - 2 < \mathbf{x}, \mathbf{y} > + < \mathbf{y}, \mathbf{y} >}. \tag{9.6}$$

A key difference between a kernel and a distance measure is that kernels measure the similarity between instances whereas distances measure the dissimilarity.

Example 9.2. Assume that the instances are vectors over $\{-1,+1\}^d$ and that we are using, as before, the inner product kernel. The maximum value of the inner product over all possible pairs of instances will be d, and this value will be obtained only when the two instances are identical. The minimum value will be $-d$, which will be obtained when the two vectors contain opposite values at all positions. So, the maximum is reached when the vectors are identical and the minimum is reached when they are opposite. It is easy to see that using a distance measure this is just the other way around. The minimum of 0 will always be reached when the objects are identical.

9.2 Kernel Methods

In this section, we briefly review the key concepts underlying kernel methods in machine learning. More specifically, we first introduce the max margin approach, continue with support vector machines, and then conclude with the definitions of some simple kernels. This section closely follows the introductory chapter of the book by Schölkopf and Smola [2002]. The next section then provides an introduction to distance-based methods in machine learning.

9.2.1 The Max Margin Approach

The basic max margin approach tackles the binary classification task, in which the instances are vectors and the classes are $\{+1,-1\}$. The goal is then to find a hypothesis h in the form of a hyperplane that separates the two classes of examples. Hyperplanes can conveniently be defined using an inner product and the equation

$$< \mathbf{w}, \mathbf{x} > +b = 0 \tag{9.7}$$

where $\mathbf{w}$ is the weight vector and b is a constant. The hyperplane then consists of all vectors $\mathbf{x}$ that are a solution to this equation. Such a hyperplane h can then be used to classify an example e as positive or negative by computing

$$h(e) = sgn(< \mathbf{w}, \mathbf{e} > +b) \text{ where} \tag{9.8}$$

$$sgn(val) = \begin{cases} +1 & \text{if } val > 0 \\ -1 & \text{if } val \leqslant 0 \end{cases} \tag{9.9}$$

The idea is not just to compute any hyperplane h that separates the two classes, but the optimal one, which maximizes the margin. The *margin* of a hyperplane $h = (\mathbf{w}, b)$ w.r.t. a set of examples E is given by

$$margin(h, E) = \min\{\|\mathbf{x} - \mathbf{e}_i\| \mid \mathbf{e}_i \in E, < \mathbf{w}, \mathbf{x} > + \, b = 0\} \tag{9.10}$$

The margin can be interpreted as the minimum distance between an example and the hyperplane. The theory of statistical learning shows that maximizing the margin provides optimal generalization behavior. Thus the maximum margin approach computes the *optimal* hyperplane, that is, the hyperplane h^* that maximizes the margin:

$$h^* = \arg\max_h margin(h, E) \tag{9.11}$$

9.2.2 Support Vector Machines

Computing the maximum margin hyperplane is an optimization problem that is usually formalized in the following way. The first requirement is that the examples $(\mathbf{e}_i, f(\mathbf{e}_i))$ are correctly classified. Mathematically, this results in the constraints:

$$\forall \mathbf{e} \in E : f(\mathbf{e})\big(< \mathbf{w}, \mathbf{e} > + \; b \; \big) \; > 0 \tag{9.12}$$

Secondly, because any solution $(\mathbf{w}, b)$ to this set of equations can be scaled by multiplying with a constant, the convention is that those examples closest to the hyperplane satisfy

$$| < \mathbf{w}, \mathbf{e} > + \; b \; | = 1 \tag{9.13}$$

This results in the following optimization problem:

$$\begin{array}{rl} \min_{\mathbf{w},b} & < \mathbf{w}, \mathbf{w} > \\ \text{subject to} & \forall \mathbf{e} \in E : f(\mathbf{e})\big(< \mathbf{w}, \mathbf{e} > + \; b \; \big) \; \geqslant 1 \end{array} \tag{9.14}$$

It can be shown that

$$margin(h, E) = \frac{1}{\|\mathbf{w}\|} \tag{9.15}$$

(see [Schölkopf and Smola, 2002] for the proof). Therefore, minimizing $\|\mathbf{w}\|$ corresponds to maximizing the $margin(h, E)$. This is illustrated in Fig. 9.1.

The optimization problem is usually not solved in the form listed above. It is rather turned into the *dual* problem using the theory of Karush, Kuhn and Tucker and the Langragian; cf. [Cristianini and Shawe-Taylor, 2000]. This dual formulation is typically an easier optimization problem to solve:

$$\begin{array}{c} \max_{(\alpha_1, \cdots, \alpha_n)} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} f(\mathbf{e}_i) f(\mathbf{e}_j) \alpha_i \alpha_j < \mathbf{e}_i, \mathbf{e}_j > \\ \text{subject to} \quad \forall i : \alpha_i \geqslant 0 \text{ and } \sum_i f(\mathbf{e}_i) \alpha_i = 0 \end{array} \tag{9.16}$$

Several public domain quadratic programming solvers exist that solve this optimization problem. Using this formulation, the weight vector $\mathbf{w}$ is a linear combination of the data vectors:

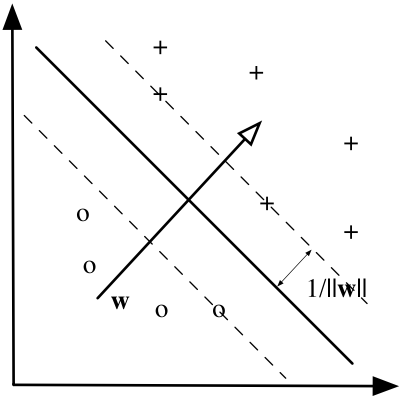


Fig. 9.1. The max margin approach and the support vectors. The positive examples are marked with a "+" and the negative ones with an "o". The maximum margin hyperplane is the middle line. The support vectors lying on the dotted lines, and the vector normal to the margin is indicated by **w**

$$\mathbf{w} = \sum_i \alpha_i f(\mathbf{e}_i)\ \mathbf{e}_i \tag{9.17}$$

Typically, many α_i will have the value 0. Those vectors $\mathbf{e}_i$ with non-zero α_i are the so-called *support vectors*. These are also the vectors that lie on the margin, that is, those for which

$$f(\mathbf{e}_i)\Big(< \mathbf{w}, \mathbf{e}_i > + \ b\ \Big) = 1 \tag{9.18}$$

These are the only ones that influence the position of the optimal hyperplane. The constant b can be obtained by substituting two support vectors, one for each class, in Eq. 9.18 and solving for b. The value $h(\mathbf{e})$ of a new data point e can then be computed as:

$$h(\mathbf{e}) = sgn\Big(\ \sum_i \alpha_i f(\mathbf{e}_i) < \mathbf{e}_i, \mathbf{e} > + \ b\ \Big) \tag{9.19}$$

Given that the α_i are non-zero only for the support-vectors, only those need to be taken into account in the above sum.

A central property of the formulation of the dual problem in Eq. 9.16 as well as the use of the resulting hyperplane for prediction in Eq. 9.19 is that they are entirely formulated in terms of inner products. To formulate the dual problem, one only needs access to the data points and to the so-called *Gram* matrix, which is the matrix containing the pairwise inner products $< \mathbf{e}_i, \mathbf{e}_j >$ of the examples.

There are numerous extensions to the basic support vector machine setting. The most important ones include an extension to cope with data sets that are not linearly separable (by introducing slack variables ξ_i for each of the instances) and several loss functions designed for regression.

9.2.3 The Kernel Trick

The support vector machine approach introduced above worked within the space $\mathbb{R}^d$, used the inner product as the kernel function and worked under the assumption that the examples were linearly separable. In this subsection, we will lift these restrictions and introduce a more formal definition of kernel functions and some of its properties.

Consider the classification problem sketched in Fig. 9.2. Using the max margin approach with the inner product kernel, there is clearly no solution to the learning task. One natural and elegant solution to this problem is to first transform the instances $\mathbf{e}$ to some vector $\Phi(\mathbf{e})$ in some other feature space and to then apply the inner product.

Example 9.3. Using the function

$$\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3 : (x_1, y_1) \rightarrow (x_1^2, \sqrt{2}x_1x_2, x_2^2)$$

the examples are separable using a hyperplane in the transformed space $\mathbb{R}^3$.

This motivates the use of the kernel

$$K(\mathbf{x}, \mathbf{y}) =< \Phi(\mathbf{x}), \Phi(\mathbf{y}) > \tag{9.20}$$

instead of the inner product in $\mathbb{R}^2$. The interesting point about this kernel K is that Φ does not have to be computed explicitly at the vectors, because

$$K(\mathbf{x}, \mathbf{y}) =< \mathbf{x}, \mathbf{y} >^2 \tag{9.21}$$

This is called the *kernel trick*. It often results in enormous computational savings as the dimension of the transformed space can be much larger than that of the input space $\mathcal{L}_e$. The example also illustrates that kernels are essentially inner products in *some* feature space that does not necessarily coincide with the input space $\mathcal{L}_e$.

Formally, functions have to satisfy some conditions for being a kernel and for the optimization approaches sketched above to work. In particular, one typically uses so-called Mercer kernels, which are symmetric functions that are positive definite.[1] For convenience, we will continue to talk about *kernels* instead of Mercer or positive definite kernels.

[1] A symmetric function is *positive definite* if and only if $\forall m \in \mathbb{N} : \forall \mathbf{x}_1, \cdots, \mathbf{x}_m \in \mathcal{L}_e : \forall a_1, \cdots, a_m \in \mathbb{R} : \sum_{i,j=1}^{m} a_i a_j K(\mathbf{x}_i, \mathbf{x}_j) \geqslant 0$, or, equivalently, if the eigenvalues of the possible Gram matrices are non-negative.

Many interesting kernels can be constructed using the following properties and the observation that the inner product in Euclidean space is a kernel. Basically, if $H(\mathbf{x},\mathbf{y})$ and $G(\mathbf{x},\mathbf{y})$ are kernels on $\mathcal{L}_e \times \mathcal{L}_e$ then the functions

$$\begin{aligned}
K_s(\mathbf{x},\mathbf{y}) &= H(\mathbf{x},\mathbf{y}) + G(\mathbf{x},\mathbf{y}) \\
K_p(\mathbf{x},\mathbf{y}) &= H(\mathbf{x},\mathbf{y})G(\mathbf{x},\mathbf{y}) \\
K_d(\mathbf{x},\mathbf{y}) &= (H(\mathbf{x},\mathbf{y}) + l)^d \\
K_g(\mathbf{x},\mathbf{y}) &= exp(-\gamma(H(\mathbf{x},\mathbf{x}) - 2H(\mathbf{x},\mathbf{y}) + H(\mathbf{y},\mathbf{y}))) \\
K_n(\mathbf{x},\mathbf{y}) &= \frac{H(\mathbf{x},\mathbf{y})}{\sqrt{H(\mathbf{x},\mathbf{x}).H(\mathbf{y},\mathbf{y})}}
\end{aligned} \tag{9.22}$$

are also kernels,[2] where s stands for sum, p for product, d for polynomial, g for Gaussian and n for normalized.

Exercise 9.4. Show how the kernel Eq. 9.21 can be formulated as a polynomial kernel. Show also that this kernel corresponds to $< \Phi(\mathbf{x}), \Phi(\mathbf{y}) >$ with Φ defined as in Eq. 9.20.

In Sect. 9.4, we shall expand these results to obtain kernels for structured data. Before doing so, we turn our attention to distance-based learning.

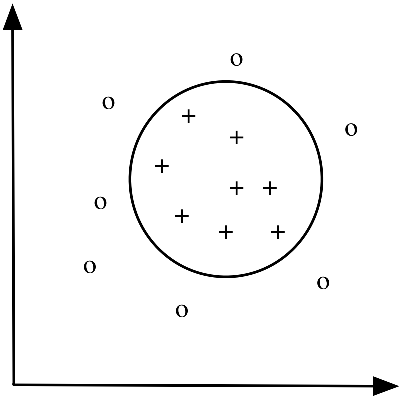


Fig. 9.2. A circular concept. The examples are marked with a "+" and the negative ones with an "o". Whereas there is no hyperplane in the original space that separates the positives from the negatives, there is one in the feature space. Mapping that hyperplane back to the original space yields the circular concept indicated

[2] For the normalized kernel, it is assumed that $K(x,x) > 0$ for all x.

9.3 Distance-Based Learning

Two of the most popular distance-based learning algorithms are the k-nearest neighbor algorithm for classification and regression and the k-means clustering algorithm. Both types of algorithms employ distance measures or metrics, which we introduce before presenting these algorithms.

9.3.1 Distance Functions

Distance measures and metrics are mathematically more easy to define than kernels, but often harder to obtain.

A distance *measure* is a function $d : \mathcal{L}_e \times \mathcal{L}_e \rightarrow \mathbb{R}$ that measures the distance between instances in e. *Distance measures* must satisfy the following requirements:

$$\begin{aligned} \text{non-negativeness: } & \forall x, y : d(x,y) \geqslant 0 && (9.23)\\ \text{reflexivity: } & \forall x, y : d(x,x) = 0 && (9.24)\\ \text{strictness: } & \forall x, y : d(x,y) = 0 \text{ only if } x = y && (9.25)\\ \text{symmetry: } & \forall x, y : d(x,y) = d(y,x) && (9.26)\end{aligned}$$

If a distance measure also satisfies the triangle inequality

$$\forall x, y, z : d(x,y) + d(y,z) \geqslant d(x,z) \tag{9.27}$$

then the distance measure is called a *metric*.

Exercise 9.5. Show that d_e as defined in Eq. 9.1 is a metric.

Using Eq. 9.22 one can define new kernels in terms of existing ones. This is also possible for distance metrics. For instance, if d_1 and d_2 are metrics defined on $\mathcal{L}_e \times \mathcal{L}_e$ and $c, k \in \mathbb{R}$, then

$$\begin{aligned} d_c(x,y) &= c d_1(x,y)\\ d_{1+2}(x,y) &= d_1(x,y) + d_2(x,y)\\ d_k(x,y) &= d_1(x,y)^k \text{ with } 0 < k \leqslant 1\\ d_n(x,y) &= \frac{d_1(x,y)}{d_1(x,y)+1} && (9.28)\end{aligned}$$

are metrics, but functions such as

$$d_p(x,y) = d_1(x,y)\ d_2(x,y) \tag{9.29}$$

are not.

Exercise 9.6. Show that d_p is not a metric.

9.3.2 The k-Nearest Neighbor Algorithm

The k-nearest neighbor algorithm applies to both classification and regression tasks, where one is given a set of examples $E = \{(e_1, f(e_1)), \cdots, (e_n, f(e_n))\}$ of an unknown target function f, a small positive integer k, and a distance measure (or metric) $d : \mathcal{L}_e \times \mathcal{L}_e \rightarrow \mathbb{R}$. The idea is to simply store all the examples, to keep the hypothesis h implicit, and to compute the predicted value $h(e)$ for an instance e by selecting the set of k nearest examples E_k to e and computing the average value for f on E_k. This prediction process is sketched in Algo. 9.1.

Algorithm 9.1 The k-nearest neighbor algorithm

```
$E_k := \{(x, f(x)) \in E | d(e, x)$ is amongst the $k$ smallest in $E\}$
Predict $h(e) := avg_{x \in E_k} f(x)$
```

In this algorithm, the function *avg* computes the average of the values of the function f in E_k. In a classification settting, *avg* corresponds to the mode, which is the most frequently occurring class in E_k, while in a regression setting, *avg* corresponds to the arithmetic average. The k-nearest neighbor algorithm is very simple to implement; it often also performs well in practice provided that the different attributes or dimensions are normalized, and that the attributes are independent of one another and relevant to the prediction task at hand; cf. standard textbooks on machine learning such as [Mitchell, 1997, Langley, 1996].

Exercise 9.7. Argue why violating one of these assumptions can give problems with the k-nearest neighbor algorithm.

Many variants of the basic k-nearest neighbor algorithm exist. One involves using all the available examples x in E but then weighting their influence in $h(e)$ using the inverse distance $1/d(e, x)$ to the example e to be classified. Other approaches compute weights that reflect the importance of the different dimensions or attributes. Further approaches devise schemes to forget some of the examples in E [Aha et al., 1991].

9.3.3 The k-Means Algorithm

Clustering is a machine learning task that has, so far, not been discussed in this book. When clustering, the learner is given a set of unclassified examples E, and the goal is to partition them into meaningful subsets called *clusters*. Examples falling into the same cluster should be similar to one another, whereas those falling into different classes should be dissimilar. This requirement is sometimes stated using the terms high *intra-cluster similarity* and low *inter-cluster similarity*. There are many approaches to clustering, but given that this chapter focuses on distances, we only present a simple clustering algorithm using distance measures.

The k-means algorithm generates k clusters in an iterative way. It starts initializing these clusters by selecting k examples at random, the initial *centroids*, and then computing the distance between each example and the k centroids. The examples are then assigned to that cluster for which the distance between the example and its centroid is minimal. In the next phase, for each cluster, the example with minimum distance to the other examples in the same cluster is taken as the new centroid, and the process iterates until it converges to a stable assignment, or the number of iterations exceeds a threshold. The algorithm is summarized in Algo. 9.2.

Algorithm 9.2 The k-means algorithm

```
Select $k$ centroids $c_k$ from $E$ at random
repeat
  Initialize all clusters $C_k$ to their centroids $\{c_k\}$
  for all examples $e \in E - \{c_1, ..., c_k\}$ do
    Let $j := \arg\min_j d(e, c_j)$
    Add $e$ to $C_j$
  end for
  for all clusters $C_j$ do
    Let $i := \arg\min_i \sum_{e \in C_j} d(e, e_i)$
    $c_j := e_i$
  end for
until convergence, or, max number of iterations reached
```

Several variants of the k-means algorithm exist. Analogously to the c-nearest neighbor algorithm there is the c-means algorithm for *soft* clustering. In soft clustering, all examples belong to all the clusters to a *certain degree.* The degree is, as in the c-nearest neighbor algorithm, inversely related to the distance to the centroid of the cluster. There is also the k-medoid algorithm, where the centroids are replaced by *medoids*, which correspond to the average of the instances in the cluster.

9.4 Kernels for Structured Data

In this section, we first introduce a general framework for devising kernels for structured data, the *convolution kernels* due to Haussler [1999], and we then show how it can be applied to different types of data: vectors and tuples, sets and multi-sets, strings, trees and atoms, and graphs. In the next section, we will take a similar approach for defining distance measures and metrics.

9.4.1 Convolution and Decomposition

In Haussler's framework for convolution kernels, $\mathcal{L}_e$ consists of structured instances e, and it is assumed that there is a *decomposition* relation R defined on $\mathcal{L}_e \times \mathcal{E}_1 \times \cdots \times \mathcal{E}_D$ such that the tuple

$$(e, e_1, \cdots, e_D) \in R \text{ if and only if } e_1, \cdots, e_D \text{ are parts of } e \tag{9.30}$$

So, the $\mathcal{E}_i$ denote spaces of parts of objects. We shall also write $(e_1, \cdots, e_D) \in R^{-1}(e)$.

The type of decomposition relation depends very much on the nature of the structured instances in $\mathcal{L}_e$. For instance, when dealing with sets, it is natural to define $R(e, x) = x \in e$; when dealing with strings, one can define $R(e, x_1, x_2)$ as concatenation, that is, $R(e, x_1, x_2)$ is true if and only if e is x_1 concatened with x_2; and when dealing with vectors in $\mathbb{R}^n$, $R(e, x_1, \cdots, x_n)$ is true if e denotes the vector $(x_1, \cdots, x_n)$. Observe that there may be multiple ways to decompose an object. For instance, when working with strings of length n, they can be decomposed in n different ways using the concatenation relation.

The basic result by Haussler employs kernels $K_d : \mathcal{E}_d \times \mathcal{E}_d \rightarrow \mathbb{R}$ defined at the part level to obtain kernels at the level of the instances $\mathcal{L}_e$ using a finite decomposition relation R. More formally, under the assumption that the K_d are kernels, Haussler shows that

$$K_{R,\oplus}(x, z) = \sum_{(x_1, \cdots, x_D) \in R^{-1}(x)} \; \sum_{(z_1, \cdots, z_D) \in R^{-1}(z)} \; \sum_{d=1}^{D} K_d(x_d, z_d) \tag{9.31}$$

$$K_{R,\otimes}(x, z) = \sum_{(x_1, \cdots, x_D) \in R^{-1}(x)} \; \sum_{(z_1, \cdots, z_D) \in R^{-1}(z)} \; \prod_{d=1}^{D} K_d(x_d, z_d) \tag{9.32}$$

are also kernels.

This is a very powerful result that can be applied to obtain kernels for sets, vectors, strings, trees and graphs, as we will show below.

The idea of defining a function on pairs of structured objects by first decomposing the objects into parts and then computing an aggregate on the pairwise parts is also applicable to obtaining distance measures. It is only that the aggregation function used will often differ, as will become clear throughout the remainder of this chapter.

9.4.2 Vectors and Tuples

The simplest data type to which decomposition applies is that of vectors and tuples. They form the basis for attribute-value learning and for the more complex relational learning settings.

Let us first reinvestigate the inner product over $\mathbb{R}^d$. Viewing the product in $\mathbb{R}$ as a kernel $K_\times$, the inner product in $\mathbb{R}^d$ can be viewed as a decomposition kernel (using Eq. 9.32):

$$< \mathbf{x}, \mathbf{y} >= \sum_i K_\times(x_i, y_i) \tag{9.33}$$

This definition can easily be adapted to define kernels over tuples in attribute-value form. The key difference is that, generally speaking, not all attributes will be numeric. To accomodate this difference, one only needs to define a kernel over a discrete attribute. This could be, for instance,

$$\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \tag{9.34}$$

Example 9.8. Reconsider the playtennis example of Table 4.1. Using δ in combination with the dot product yields:

$$< (\mathsf{sunny}, \mathsf{hot}, \mathsf{high}, \mathsf{no}), (\mathsf{sunny}, \mathsf{hot}, \mathsf{high}, \mathsf{yes}) >= 3$$

9.4.3 Sets and Multi-sets

When using sets, and membership as the decomposition relation, and a kernel K_{el} defined on elements, we obtain the convolution kernel K_{Set}:

$$K_{Set}(X, Y) = \sum_{x_i \in X} \sum_{y_j \in Y} K_{el}(x_i, y_j) \tag{9.35}$$

For instance, if $K_{el}(x, y) = \delta(x, y)$ (cf. Eq. 9.34), then

$$K_{Set-\delta}(X, Y) = |X \cap Y| \tag{9.36}$$

Of course, other kernels at the element level can be choosen as well. For sets, it is also convenient to *normalize* the kernel. This can be realized using Eq. 9.22, which yields

$$K_{SetNorm}(X, Y) = \frac{K_{Set}(X, Y)}{\sqrt{K_{Set}(X, X) K_{Set}(Y, Y)}} \tag{9.37}$$

Using the δ kernel, we obtain

$$K_{SetNorm-\delta}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X| \; |Y|}} \tag{9.38}$$

where it is assumed that the sets are not empty.

The K_{Set} kernel naturally generalizes to multi-sets. The only required change is that in the summations the membership relation succeed multiple times for those objects that occur multiple times.

Exercise 9.9. Compute $K_{MultiSet-\delta}(\{a, a, b, b, c\}, \{a, a, b, b, b, d\})$ using $K_{el} = \delta$.

9.4.4 Strings

Let us now apply convolution kernels to strings. There are two natural ways of decomposing a string into smaller strings. The first employs the substring relation, the second the subsequence relation.

A string $S : s_0 \cdots s_n$ is a *substring* of $T : t_0 \cdots t_k$ if and only if

$$\exists j : s_0 = t_j \wedge \cdots \wedge s_n = t_{j+n} \tag{9.39}$$

that is, all symbols in the string S occur in consecutive positions in the string T.

On the other hand, a string $S : s_0 \cdots s_n$ is a *subsequence* of $T : t_0 \cdots t_k$ if and only if

$$\exists j_0 < j_1 < \cdots < j_n : s_0 = t_{j_0} \wedge \cdots \wedge s_n = t_{j_n} \tag{9.40}$$

that is all symbols in the string S occur in positions in the string T in the same order, though not necessarily consecutively. $j_n - j_0 + 1$ is called the *occurring length* of the subsequence. For instance, the string *achi* is a substring of *machine* and a subsequence of *Tai-chi* of length 5.

These two notions result in alternative relationships R. Let us first consider the substring case, where the string can be decomposed into its substrings. It is convenient to consider only substrings from Σ^n, the set of all strings over Σ containing exactly n symbols, yielding:

$$R^{-1}_{substring}(s) = \{t \in \Sigma^n | t \text{ is a substring of } s\} \tag{9.41}$$

Note that it is assumed that $R^{-1}_{substring}$ returns a multi-set. In combination with a kernel such as δ, it is easy to obtain a substring kernel.

Example 9.10. Consider the strings *john* and *jon*. When choosing $n = 2$:

$$R^{-1}_{substring}(john) = \{jo, oh, hn\}$$
$$R^{-1}_{substring}(jon) = \{jo, on\}$$

Therefore, $K_{substring}(john, jon) = 1$ because there is one common substring of length 2.

It is also interesting to view this kernel as performing a kind of feature construction or propositionalization. Essentially,

$$K_{substring}(s, t) = < \Psi_{substring}(s), \Psi_{substring}(t) >, \tag{9.42}$$

which corresponds to the inner product using the feature mapping $\Psi_{substring}(s) = (\psi_{u_1}(s), \cdots, \psi_{u_k}(s))$ with $\Sigma^n = \{u_1, \cdots, u_k\}$ and $\psi_{u_i}(s)$ = number of occurrences of u_i as a substring in s. This kernel is sometimes called the *n-gram* kernel.

An interesting alternative uses the subsequence relation:

$$R^{-1}_{subseq}(s) = \{t/l | t \in \Sigma^n, t \text{ is a subsequence of } s \text{ of occurring length } l\} \tag{9.43}$$

Again, it is assumed that a multi-set is returned. The idea of the subsequence kernel is that subsequences with longer occurring lengths are penalized. This is realized using the following kernel in the convolution in Eq. 9.32:

$$K_{pen}(s/l, t/k) = \begin{cases} \lambda^{l+k} & \text{if } s = t \\ 0 & \text{otherwise} \end{cases} \tag{9.44}$$

where $0 < \lambda < 1$ is a decay factor that determines the influence of occurring length. The larger λ, the less the influence of gaps in the subsequences.

Example 9.11. Reconsider the strings *john* and *jon*. When choosing $n = 2$:

$$\begin{aligned} R^{-1}_{subseq}(john) &= \{jo/2, jh/3, jn/4, oh/2, on/3, hn/2\} \\ R^{-1}_{subseq}(jon) &= \{jo/2, jn/3, on/2\} \end{aligned}$$

Hence, $K_{subseq,pen}(john, jon) = \lambda^{2+2} + \lambda^{3+2} + \lambda^{4+3}$ for the subsequences jo, on and jn.

This kernel corresponds to the feature mapping $\Psi_{subseq}(s) = (\phi_{u_1}(s), \cdots, \phi_{u_k}(s))$ where $\phi_{u_i}(s) = \sum_{u_i/l \in R^{-1}_{subseq}(s)} \lambda^l$.

Exercise 9.12. If one takes into account all strings of length less than or equal n instead of only those of length n, does one still obtain valid substring and subsequence kernels? Why? If so, repeat the two previous exercises for this case.

The two kernels K_{subseq} and $K_{substring}$ can be computed efficiently, that is, in time $O(n|s||t|)$, where $|s|$ denotes the length of the string s; cf. Gärtner [2003].

9.4.5 Trees and Atoms

As this book is largely concerned with logical representations, which are centered around terms and atoms, this subsection concentrates on defining kernels and distances for these structures, in particular, ground atoms. Nevertheless, due to their intimate relationship to ordered trees, the resulting kernels and distances can easily be adapted for use with ordered and unordered trees.

Because ground terms are hierachically structured, there is a natural way of decomposing them. We assume that two ground terms and the type of each of the arguments are given. One can then inductively define a kernel on ground terms $K_{Term}(t, s)$:

$$= \begin{cases} k_c(t,s) & \text{if } s \text{ and } t \text{ are constants} \\ k_f(g,h) & \text{if } t = g(t_1, \cdots, t_k) \text{ and } s = h(s_1, \cdots, s_m) \\ & \quad \text{and } g \neq h \\ k_f(g,g) + \sum_i K_{Term}(t_i, s_i) & \text{if } t = g(t_1, \cdots, t_k) \text{ and } t = g(s_1, \cdots, s_k) \end{cases} \tag{9.45}$$

This definition assumes that k_c is a kernel defined on constants and k_f a kernel defined on functor names. The kernel K_{Term} is an instance of the convolution kernel defined in Eq. 9.32. It uses the decomposition function $R^{-1}(f(t_1, t_2, \cdots, t_n)) = (f, n, t_1, \cdots, t_n)$.Notice that it is also possible to employ the product convolution kernel of Eq. 9.32, which can be obtained by replacing all sums in the recursive case by products; cf. [Menchetti et al., 2005].

Example 9.13. If $k_c = k_f = \delta$, then

$$\begin{aligned} K_{term}(r(a,b,c), r(a,c,c)) &= k_f(r,r) + [K_{term}(a,a) + K_{term}(b,c) \\ &\quad + K_{term}(c,c)] \\ &= 1 + [\delta(a,a) + \delta(b,c) + \delta(c,c)] \\ &= 1 + [1 + 0 + 1] \\ &= 3 \end{aligned}$$

It is clear that K_{term} can be computed efficiently, that is, in time linearly in the size of the two terms.

9.4.6 Graph Kernels*

Throughout this chapter, we shall focus on directed graphs in which the nodes are labeled and the edges are not, though the reader should keep in mind that it is often straightforward to adapt the framework to other types of graphs. Formally, the type of graph G considered is a triple (V, E, l), where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and the labeling function $l : V \rightarrow \Sigma$ maps vertices onto their labels from an alphabet Σ. Fig. 9.3 displays two graphs of this type, where the vertices are numbered and the labels are from the alphabet $\Sigma = \{A, B, C, \cdots, Z\}$. When the context is clear, we may omit the names of the vertices.

Subgraph Isomorphism

When working with graphs, the generality relation is specified using the notion of subgraph morphism. For logical formulae, various notions of subsumption exist that posses different properties, which is also the case for subgraph morphisms. Best known is the subgraph isomorphism, which closely corresponds to OI-subsumption, introduced in Sect. 5.5.1. In a subgraph ismorphism each node of the more general graph is mapped onto a different node in the more

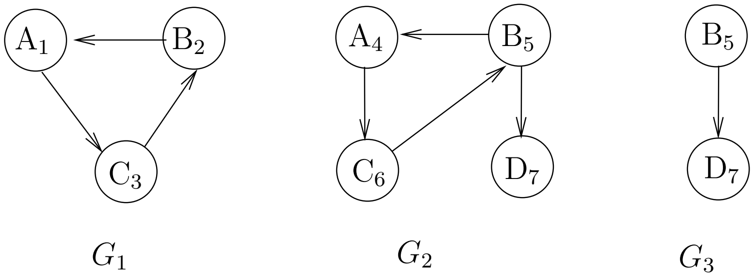


Fig. 9.3. The graphs G_1, G_2 and G_3

specific one. The nodes in the subgraph, that is, the more general graph, play the role of logical variables, and, as for OI-subsumption, two different nodes (or logical variables) must be mapped onto different nodes (or terms) in the subsumed graph. An alternative is the notion of subgraph *homeomorphism*, which is closely related to θ-subsumption, introduced in Sect. 5.4, and does allow for two nodes in the more general graph being mapped onto the same node in the more specific graph. For simplicity, we employ in this chapter a restricted form of subgraph isomorphism called *induced subgraph ismorphism.* It is restricted because it employs subgraphs that are induced by the subset relation at the level of vertices.[3] More formally, $G_s = (V_s, E_s, l_s)$ is an (induced) *subgraph* of a graph $G = (V, E, l)$ if and only if

$$V_s \subseteq V, E_s = E \cap (V_s \times V_s), \text{ and } \forall v \in V_s : l_s(v) = l(v) \tag{9.46}$$

Because the subgraph G_s is completely determined by V_s and G, we refer to G_s as the subgraph of G *induced* by V_s. The reader may want to verify that in Fig. 9.3 G_3 is the subgraph of G_2 induced by $\{5, 7\}$.

A bijective function $f : V_1 \to V_2$ is a *graph isomorphism* from $G_1 = (V_1, E_1, l_1)$ to $G_2 = (V_2, E_2, l_2)$ if and only if

$$\forall v_1 \in V_1 : l_2(f(v_1)) = l_1(v_1) \text{ and} \tag{9.47}$$

$$\forall (v, w) \in E_1 : (f(v), f(w)) \in E_2 \text{ and, vice versa} \tag{9.48}$$

$$\forall (v, w) \in E_2 : (f^{-1}(v), f^{-1}(w)) \in E_1 \tag{9.49}$$

Graph isomorphisms are important because one typically does not distinguish two graphs that are isomorphic, a convention that we will follow throughout this book. The notion is also used in the definition of the generality relation.

An injective function $f : V_1 \to V_2$ is a *subgraph isomorphism* from $G_1 = (V_1, E_1, l_1)$ to $G_2 = (V_2, E_2, l_2)$ if G_1 is an (induced) subgraph of G_2 and f is a graph isomorphism from G_1 to G_2.

[3] The more general notion of subgraph ismorphism takes into account the edges as well, but this is more complicated.

Example 9.14. There exists a subgraph isomorphism from G_1 to G_2. Consider the subgraph of G_2 induced by $\{4, 5, 6\}$. This subgraph is isomorphic to G_1 by using the function $f(1) = 4, f(2) = 5$ and $f(3) = 6$.

Exercise 9.15. * Represent the graphs of the previous example as sets of facts. Use this representation to relate the various types of graph isomorphism to the different notions of subsumption; cf. Sect. 5.9.

Graph Kernels

When developing kernels for graphs or unordered rooted trees, it is tempting to make an analogy with the string kernels. The analogy suggests decomposing the graph into its subgraphs because strings were decomposed into substrings or subsequences.

In terms of a feature mapping, this corresponds to the choice

$$\Psi_{subgraph}(G) = (\psi_{g_1}(G), \psi_{g_2}(G), \psi_{g_3}(G) \cdots) \tag{9.50}$$

for a particular (possibly infinite) enumeration of graphs $g_1, g_2, \cdots$ and the features

$$\psi_{g_i}(G) = \lambda_{|edges(g_i)|} \times |\{g \text{ subgraph of } G | g \text{ is isomorphic to } g_i\}| \tag{9.51}$$

where it is assumed that there is a sequence of decay factors $\lambda_i \in \mathbb{R}$, and the $\lambda_i > 0$ account for the possibly infinite number of graphs. The result is a valid kernel (as shown by Gärtner et al. [2003]). However, computing the kernel is NP-complete, which can be proven using a reduction from the Hamiltonian path problem, the problem of deciding whether there exists a path in a directed graph that visits all nodes exactly once; cf. Gärtner et al. [2003]. The hardness even holds when restricting the considered subgraphs to paths. Recall that a *path* in a graph is a sequence of vertices $v_0, \cdots, v_n$ in the graph such that each pair of consecutive vertices v_i, v_{i+1} is connected by an edge and no vertex occurs more than once. For instance, in G_2 of Fig. 9.3, $4, 6, 5, 7$ is a path. It is convenient to denote the paths through the string of labels $l(v_0) \cdots l(v_n)$ from Σ^*, Σ being the alphabet of labels used in the graph. For instance, the previous path can be written as $ACBD$.

Because of the computational problems with the above subgraph kernels, the kernel community has developed a number of alternative, less expressive graph kernels. One approach considers only specific types of subgraphs in the decomposition, for instance, the frequent subgraphs, or all paths up to a fixed length. A quite elegant and polynomial time computable alternative considers all (possibly infinite) walks in the graph; cf. Gärtner et al. [2003].

A *walk* in a graph is a sequence of vertices where each pair of consecutive vertices is connected but it is not required that each vertex occurs only once. As a consequence, any (finite) graph contains only a finite number of paths, but may contain an infinite number of walks. For instance, in graph G_2, the sequence $4, 6, 5, 4, 6, 5, 7$ is a walk (in label notation $ACBACBD$).

The feature mapping considered by Gärtner et al. [2003] is based on the features ψ_s with $s \in \Sigma^*$:

$$\psi_s(G) = \sqrt{\lambda_{|s|}}|\{w|w \text{ is a walk in } G \text{ encoding } s\}| \tag{9.52}$$

where the λ_k are decay factors as before. The interesting point about the resulting kernel K_{walk} is that it can be computed in polynomial time when choosing the geometric or the exponential series for the λ_i. (In the geometric series, $\lambda_i = \gamma^{-i}$; in the exponential one, $\lambda_i = \frac{\beta^i}{i!}$).

The algorithm for computing K_{walk} employs product graphs and adjacency matrices, which we now introduce. Given two labeled graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$, the *product graph* $G_{1\times 2} = G_1 \times G_2$ is defined as

$$\begin{aligned} V_{1\times 2} &= \{(v_1, v_2) \in V_1 \times V_2 | l_1(v_1) = l_2(v_2)\} \\ E_{1\times 2} &= \{((u_1, u_2), (v_1, v_2)) \in V_{1\times 2}^2 | (u_1, v_1) \in E_1 \text{ and } (u_2, v_2) \in E_2\} \\ \forall (v_1, v_2) \in V_{1\times 2}: \quad & l_{1\times 2}((v_1, v_2)) = l_1(v_1) \end{aligned} \tag{9.53}$$

The product graph corresonds to a kind of generalization of the two graphs.

Example 9.16. Fig. 9.4 contains the product graph of G_1 and G_2 of Fig. 9.3.

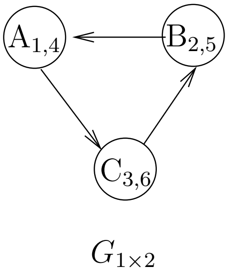


Fig. 9.4. The product graph $G_1 \times G_2$ of G_1 and G_2

Graphs can be conveniently represented using their adjacency matrices. For a particular ordering of the vertices $v_1, \cdots, v_n$ of a graph $G = (V, E, l)$, define the matrix of dimension $n \times n$ where $\mathbf{A}_{ij} = 1$ if $(v_i, v_j) \in E$, and 0 otherwise. It is well-known that the nth power of the adjacency matrix $\mathbf{A}_{ij}^n$ denotes the number of walks of length n from vertex i to vertex j; cf. [Rosen, 2007].

Example 9.17. The product graph $G_1 \times G_2$ in Fig. 9.4 can be represented by the matrix (for the order $(1, 4), (2, 5), (3, 6)$):

$$\mathbf{A}_{1\times 2} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

and the second power of the matrix is

$$\mathbf{A}^2_{1\times 2} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

The reader may want to verify that there are indeed only three walks of length 2 through the product graph and that these correspond to $(1,4)$ to $(2,5)$, $(2,5)$ to $(3,6)$ and $(3,6)$ to $(1,4)$.

Gaertner et al. show that

$$K_{walk}(G_1, G_2) = \sum_{i,j=1}^{|V_{1\times 2}|} \Big(\sum_{n}^{\infty} \lambda_n A^n_{1\times 2} \Big)_{ij} \tag{9.54}$$

where $A_{1\times 2}$ denotes the adjacency matrix of $G_1 \times G_2$. This series can be computed in cubic time when choosing the exponential and geometric series for λ_i.

Devising kernels for structured data types, such as graphs, trees, and sequences, is an active research area and several alternative kernels exist.

We now turn our attention to distances and metrics, where we identify similar principles that can be used to define such measures for structured data types.

9.5 Distances and Metrics

Although there exists a wide range of possible distance measures and metrics for structured and unstructured data, there are some common ideas that we will introduce throughout this section. The first idea is that of decomposition. Under conditions to be specified, metrics and distances for structured objects can be defined by decomposing the objects and then aggregating metrics or distances defined on these components. Various metrics to be presented, such as the Hausdorff and the matching distances on sets, work in this way. The second idea, related to generalization, is based on the observation that many distances try to identify common parts amongst these instances. A recent result by De Raedt and Ramon [2008], introduced in Sect. 9.5.1, shows that many distance metrics can be related to the size of the minimally general generalizations of the two instances. The larger the common part or generaliation, the smaller the distance. The distance between the two instances can be defined as the cost of the operations needed to specialize the minimally general generalizations to the instances. This is sometimes referred to as an *edit-distance*. The third observation is that some distances do not rely on generalizations of the two instances but rather on a correspondence, a so-called match or matching, of the parts of one instance to those of the second one. Thus, very often, distances are related to generalization and matchings. In this section, we shall first study the relationship between metrics and generalization, and then introduce a wide variety of metrics for vectors, sets, strings, atoms and graphs.

9.5.1 Generalization and Metrics

We consider partially ordered pattern or hypothesis spaces $(\mathcal{L}_e, \preceq)$, where we write, as in Chapter 5, $s \preceq g$ (or $s \prec g$) to denote that s is more specific than (or strictly more specific than) g. Throughout this section, it will be assumed that the relation $\preceq$ is partially ordered, that is, it is *reflexive*, *anti-symmetric* and *transitive*. Recall from Chapter 5 that not all generalization relations satisfy these requirements. We shall also assume

1. that there is a function *size* $|\_| : \mathcal{L}_e \rightarrow \mathbb{R}$ that is *anti-monotonic* w.r.t. $\preceq$, that is,

$$\forall g, s \in \mathcal{L}_e : s \preceq g \rightarrow |s| \geqslant |g| \tag{9.55}$$

   and *strictly order preserving*

$$\forall g, s \in \mathcal{L}_e : s \prec g \rightarrow |s| > |g| \tag{9.56}$$

2. that the *minimally general generalizations* and *maximally general specizializations*, introduced in Eq. 3.26, of two patterns are defined and yield at least one element:

$$mgg(x, y) = min\{l \in \mathcal{L}_e | x \preceq l \text{ and } y \preceq l\} \tag{9.57}$$
$$mgs(x, y) = min\{l \in \mathcal{L}_e | l \preceq x \text{ and } l \preceq s\} \tag{9.58}$$

   We also define:

$$|mgg(x, y)| = max_{m \in mgg(x,y)} |m| \tag{9.59}$$
$$|mgs(x, y)| = min_{m \in mgs(x,y)} |m| \tag{9.60}$$

We can then define the *generalization* distance $d_{\preceq}(x, y)$ on elements $x, y \in \mathcal{L}_e$:

$$d_{\preceq}(x, y) = |x| + |y| - 2|mgg(x, y)| \tag{9.61}$$

That is, to go from x to y, one should go from x to $mgg(x, y)$ and then to y. Therefore, the distance $d_{\preceq}$ can be interpreted as an *edit-distance*, measuring the cost of performing operations on x to turn it into y, or vice versa. The costs are here interpreted as the difference in size between the $mgg(x, y)$ and x or y.

An interesting property, called the *diamond inequality*, for this type of distance function states that

$$\forall x, y : d_{\preceq}(x, y) \leqslant d_{\preceq}(x, mgs(x, y)) + d_{\preceq}(mgs(x, y), y) \tag{9.62}$$

or, equivalently,

$$\forall x, y : 2|x| + 2|y| \leqslant 2|mgg(x, y)| + 2|mgs(x, y)|. \tag{9.63}$$

When a distance function satisfies the diamond equation, the distance from x to y via $mgg(x, y)$ is shorter than that via $mgs(x, y)$. This is illustrated in Fig. 9.5. De Raedt and Ramon [2008] show that the distance $d_{\preceq}$ is a metric if it satisfies the diamond inequality (and $(\mathcal{L}_e, \preceq)$ is a partial order and the size $|\_|$ satisfies the above stated requirements). Metrics obtained in this way are called *generalization distances.*

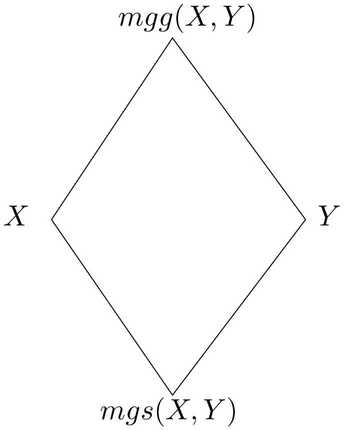


Fig. 9.5. The diamond inequality. Adapted from [De Raedt and Ramon, 2008]

This is a fairly interesting result, because it connects the key notion from logical learning (generality) with the key notion of distance-based learning (metrics). Its use will be illustrated below, when developing metrics for vectors, sets, sequences, trees and graphs.

9.5.2 Vectors and Tuples

There are two ways to obtain a distance metric on vectors or tuples. The first applies the idea of decomposition. For (symbolic) attribute-value representations, it is convenient to define

$$d_{att}(\mathbf{x}, \mathbf{y}) = \sum_i d_i(x_i, y_i) \tag{9.64}$$

The distance d_{att} will be a metric provided all the functions d_i are metrics as well. A convenient choice for the function d_i is $\overline{\delta}$, the distance corresponding to the kernel δ from Eq. 9.34:

$$\overline{\delta}(x,y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases} \tag{9.65}$$

The second way of obtaining the same distance is to use the natural generalization relation on attribute-value representations and then define a generalization distance. Using the notation of logical atoms to represent attribute-value tuples, the subsumption relation on logical atoms studied in Sect. 5.3 as the generalization relation, and the number of constants in a description as size yields the same metric d_{att}.

Example 9.18. Consider the examples

$$\mathsf{playtennis(sunny, hot, high, no)}$$
$$\mathsf{playtennis(sunny, mild, low, no)}$$

and their generalization

$$\mathsf{playtennis(sunny, X, Y, no)}$$

Thus the distance according to $d_{\preceq}$ is $2 + 2 - 2 = 2$, which is the same as that when using the d_{att} with $\overline{\delta}$.

Exercise 9.19. Prove that d_{att} is a metric when all of the d_i are metrics.

Exercise 9.20. Show that $d_{\preceq}$ satisfies the diamond inequality in this case.

When working with tuples on continuous attributes, it is convenient to use the natural distance $d_a(x,y) = |x - y|$ on $\mathbb{R}$. Decomposing the vectors along the different dimensions and adding the corresponding distances yields the distance d_{man}, known in the literature as the *Manhattan* or *cityblock* distance:

$$d_{man}(\mathbf{x},\mathbf{y}) = \sum_i d_a(x_i, y_i) \tag{9.66}$$

9.5.3 Sets

Even though sets are – like vectors – relatively simple data structures, they are much more challenging from a distance point of view. There actually exists a wide range of possible distance measures and metrics for sets. It is instructive to develop a number of key representatives carefully and to relate them to notions of generalization and matching, which will be useful for more complex data structures.

A Simple Metric for Sets

Perhaps the simplest metric for sets is d_{Set}:

$$d_{Set}(X, Y) = |X \setminus Y| + |Y \setminus X| \tag{9.67}$$

A normalized version can be obtained by dividing by $|X \cup Y|$.

This metric can be reinterpreted as another instance of the generalization distance (using the generalization relation for sets, which was extensively studied in Chapter 3, and the natural size measures on sets). The *lgg* of two sets is their intersection, and the *glb* their union. Therefore,

$$\begin{aligned} d_{\preceq}(X, Y) &= |X| + |Y| - 2|lgg(X, Y)| \\ &= |X| - |X \cap Y| + |Y| - |X \cap Y| \\ &= |X - Y| + |Y - X| \\ &= d_{Set}(X, Y) \end{aligned} \tag{9.68}$$

The Hausdorff Metric

The distance d_{Set} is to some extent the natural metric analogue to the $K_{Set-\delta}$ kernel (cf. Eq. 9.36). We may now try to express d_{Set} using decomposition and a metric d_{el} defined on elements of the set. The natural candidate for d_{el} is the distance $\overline{\delta}$ corresponding to δ. It is, however, not easy to employ decomposition as this results in the computation of an aggregate over all *pairs* of elements. However, a reformulation that splits up the contribution over the different sets works for $\overline{\delta}$, though not in general.

$$d_{Set-d_{el}}(X, Y) = \sum_{x_i \in X} \left(\min_{y_j \in Y} d_{el}(x_i, y_j) \right) + \sum_{y_j \in Y} \left(\min_{x_i \in X} d_{el}(x_i, y_j) \right) \tag{9.69}$$

The expression $\min_{y_j \in Y} d_{el}(x_i, y_j))$ captures the idea that the distance $d(x_i, Y)$ from x_i to the set Y is given by the distance of x_i to the closest point on Y. This formulation tries to match elements of one set to elements of the other sets. The reader may want to verify that $d_{Set-\overline{\delta}} = d_{Set}$, that is, applying decomposition with the $\overline{\delta}$ distance at the level of elements results in the distance d_{Set} for sets specified in Eq. 9.67.

Exercise 9.21. Show that, in general, the distance $d_{Set-d_{el}}$ is not a metric even when d_{el} is. Hint: use $d_a(x, y) = |x - y|$ defined on $\mathbb{R}$ as d_{el}.

A refinement of the formulation in Eq. 9.69 does work. It is perhaps the most famous metric for sets, the *Hausdorff* metric. It follows a similar pattern:

$$d_{HD-d_{el}}(X, Y) = \max \left(\max_{x_i \in X} \left(\min_{y_j \in Y} d_{el}(x_i, y_j) \right), \max_{y_j \in Y} \left(\min_{x_i \in X} d_{el}(x_i, y_j) \right) \right) \tag{9.70}$$

The Hausdorff metric is based on the previous idea that the distance $d(x, Y)$ between an instance x and a set of instances Y is defined as $\min_{y \in Y} d_{el}(x, y)$. But then, the distance between two sets X and Y is defined as the maximum distance of a point in X to the set Y, that is, $\max_{x \in X} \min_{y \in Y} d_{el}(x, y)$. Unfortunately this is not a metric, because the symmetry requirement does not hold. Symmetry can be restored by taking the maximum of the distance of X to Y and that of Y to X, and this is exactly what the Hausdorff distance does. The Hausdorff metric is a metric whenever the underlying distance d_{el} is a metric.

Example 9.22. (from Ramon [2002]) Let $\mathcal{L}_e = \mathbb{R}^2$ and consider the Manhattan distance d_{man}. Let

$$X = \{(0, 0), (1, 0), (0, 1)\}$$

and

$$Y = \{(2, 2), (1, 3), (3, 3)\}$$

One can then easily verify that

$$\begin{aligned} d((0, 0), Y) &= 3 \\ d((1, 0), Y) &= 2 \\ d((0, 1), Y) &= 2 \\ d((2, 2), X) &= 3 \\ d((1, 2), X) &= 2 \\ d((3, 3), X) &= 5 \end{aligned}$$

Therefore,

$$d_{HD_{d_{man}}}(X, Y) = 5$$

Even though the Hausdorff metric is a metric, it does not capture much information about the two sets as it is completely determined by the distance of the most distant elements of the sets to the nearest neighbor in the other set.

Exercise 9.23. Show that the function

$$d_{min}(X, Y) = \min_{x_i \in X, y_j \in Y} d_{el}(x_i, y_j)$$

defined on sets is not necessarily a metric when d_{el} is.

Matching Distances

These drawbacks actually motivate the introduction of a different notion of matching between two sets. The previous two distances associate elements y in one set to the closest element in the other set X using $\min_{x \in X} d_{el}(x, y)$. This type of approach allows one element in one set to match with multiple elements in the other set, which may be undesirable. Many distances therefore employ matchings, which associate one element in a set to at most one other element. Formally, a *matching* m between two sets X and Y is a relation $m \subseteq X \times Y$ such that every element of X and of Y is associated with at most one other element of the other set. Furthermore, if $|m| = min(|X|, |Y|)$, then the matching is called *maximal.*

Example 9.24. Consider the sets $X = \{a, b, c\}$ and $Y = \{d, e\}$. Then one matching is $\{(a, d), (c, e)\}$. It is also maximal.

Given a matching m, define the distance:

$$d(m, X, Y) = \sum_{(x,y) \in m} d_{el}(x, y) + \frac{|X - m^{-1}(Y)| + |Y - m(X)|}{2} \times M \quad (9.71)$$

where M is a large constant, larger than the largest possble distance according to d_{el} between two possible instances. So, $M > \max_{x,y} d_{el}(x, y)$.

This can be used to define a matching distance for sets:

$$d_{Set-match}(X, Y) = \min_{m \in matching(X,Y)} d(m, X, Y) \quad (9.72)$$

Example 9.25. Consider the sets $X = \{a, b, c\}$ and $Y = \{a, e\}$, the matching $\{(a, a), (c, e)\}$, the distance $\overline{\delta}$ and set $M = 4$ then

$$d(m, X, Y) = \overline{\delta}(a, a) + \overline{\delta}(c, e) + \frac{2 + 0}{2} \times 4 = 3$$

The best matching is also m. and therefore $d_{Set-match} = 2$.

This distance can be related to the generalization distance introduced before. The key difference is that we now employ a *matching*, which is identifying not only common parts but also parts that are close to one another. The matching distance can be written as:

$$d(m, X, Y) = c\big(m(X, Y)\big) + c\big(m(X, Y) \rhd X\big) + c\big(m(X, Y) \rhd Y\big) \quad (9.73)$$

where the matching part is denoted as $m(X, Y)$, a kind of “approximative” mgg. The function c represents an edit cost, that is the cost $c(m(X, Y))$ the matching m needs to turn X into Y, or the cost $c(m(X, Y) \rhd X)$ for turning $m(X, Y)$ into X. The formula for $d(m, X, Y)$ generalizes that for the generalization distance of Eq. 9.61, because for the generalization distance, m would be taken as the maximal size mgg, $c(m(X, Y)) = 0$, and $c(m(X, Y) \rhd X) = |X| - |mgg(X, Y)|$. Thus a non-zero cost can now be associated with the “matching” part of X and Y.

Computing Matching Distances for Sets*

The distance $d_{Set-match}$ can be computed in polynomial time using flow networks; cf. Ramon [2002], Ramon and Bruynooghe [2001]. A flow network is shown in Fig. 9.6. The network is essentially a directed acyclic graph, with a designated start node s and terminal node t, and, furthermore, the edges in the network are labeled as a tuple $(cap, cost) : flow$, where the capacity *cap* denotes the possible flow through the edge, and the *cost* per unit of flow that needs to be paid for flow through the edge. It is convenient to view a flow network as an abstract model of a pipeline system. The edges then correspond to the different pipes in the system, and the nodes to points where different pipes come together. The *maximum flow* problem through the network models the problem of transporting as much flow as possible from s to t. The flow $flow$ is a function from the edges in the network to $\mathbb{N}$ or $\mathbb{R}$. The inflow into a node n is the sum of the flows on the edges leading to that node, and the outflow is the sum of the flows on edges starting from that node. A flow network satisfies some natural constraints. One constraint states that the inflow is equal to the outflow for all nodes (except s and t) in the network. Another one that the flow on each edge can be at most equal to the capacity.

The *maximum flow problem* is concerned with finding

$$\begin{aligned} maxflow &= \arg \max_{f \in Flow} \sum_x f(s,x) \qquad (9.74) \\ &= \arg \max_{f \in Flow} \sum_y f(y,t) \qquad (9.75) \end{aligned}$$

A simple flow network with corresponding maximum flow is shown in Fig. 9.6. Typically, one is not only interested in the maximum flow, but also in the maximum flow that has the mimimum cost. This is known as the *minimum cost maximum flow problem.* It consists of finding the maximum flow f for which

$$\sum_{(x,y) \in edges} f(x,y)\ c(x,y)$$

is minimal, where $c(x,y)$ denotes the cost for flow from x to y. The flow illustrated in Fig. 9.6 has minimum cost. There are polynomial algorithms for computing the minimum cost maximum flow [Cormen et al., 2001].

There are now two interesting applications of the maximum flow minimum cost problem to computing macthing distances between sets; cf. Ramon and Bruynooghe [2001], Ramon [2002] The first computes the distance $d_{Set-match}$ above using the flow network specified in Fig. 9.7:

- there is a node x_i for each element of the set X, as well as an extra *sink* node x_- for X,
- there is a node y_i for each element of the set Y, as well as an extra *sink* node y_- for Y,

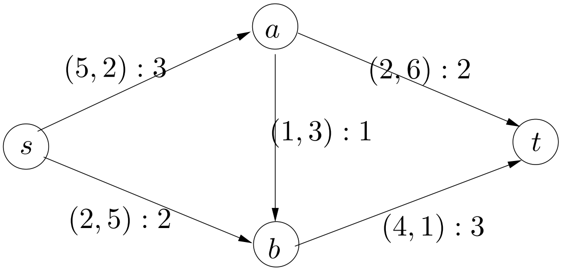


Fig. 9.6. The minimum cost maximum flow problem. The labels $(cap, cost) : flow$ denote the optimal $flow$ through the edge, the capacity cap of the edge, and the $cost$ per unit of flow through the edge

- there are edges from s to the x_i with label $(1, 0)$,
- there is an edge from s to x_- with label $(N, 0)$ where N is a constant such that $N \geqslant max(|X|, |Y|)$,
- there are edges from the y to t with label $(1, 0)$,
- there is an edge from y_- to t with label $(N, 0)$,
- there are edges from x_i to the y_i labeled $(1, d(x_i, y_i))$,
- there are edges from the x_i to y_- and from x_- to the y_i, with labels $(\infty, M/2)$, where M is the constant defined in Eq. 9.72.

Furthermore, only integers are allowed as flows. The minimum cost maximum flow then encodes the optimal matching. It essentially matches elements for which $flow(x_i, y_i) = 1$.

Jan Ramon has introduced also a variant of the distance $d_{Set-match}$. This is motivated by the fact that $d_{Set-match}$ is often dominated by the sizes of the sets. Certainly, when the sizes of the two sets are far apart, the largest contribution will come from the unmatched parts. The variant works with *weighted* sets; these are sets S where each element carries a particular weight $w_S(x)$. The idea is that the weight denotes the importance of the element in the set. The resulting distance measure can be computed using the flow network depicted in Fig. 9.7, but with the edges from s to x_i labeled by $(w_X(x_i), 0)$, and those from y_i to t by $(w_Y(y_i), 0)$. Furthermore, it is no longer required that the flow be an integer function. This formulation allows an element to match with multiple elements in the other set, and the flow will indicate the importance of the individual matches.

9.5.4 Strings

Some of the ideas for sets carry over to strings. Let us first look at distances between strings of fixed length (or size) n, that is strings in Σ^n. Let us denote the two strings $s_1 \cdots s_n$ and $t_1 \cdots t_n$. Applying decomposition leads to the Hamming distance:

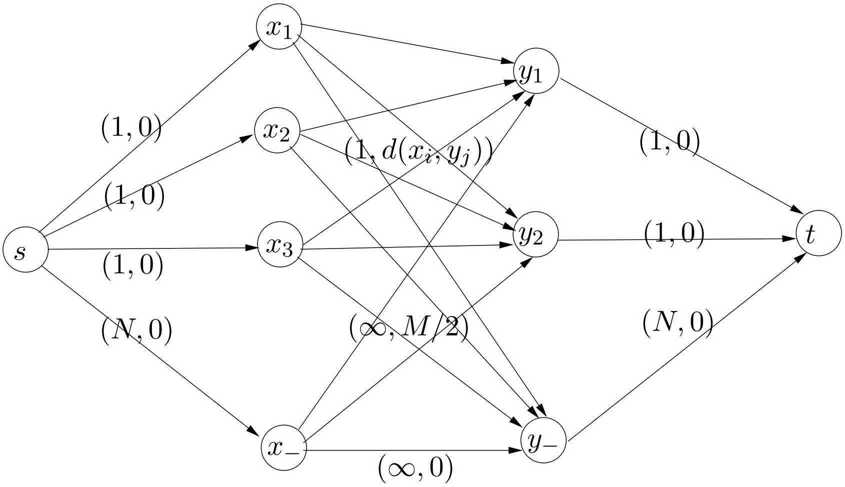


Fig. 9.7. The minimum cost maximum flow problem. The labels $(cap, cost)$ denote the capacities and the costs per flow of the edges. Adapted from [Ramon and Bruynooghe, 2001]

$$d_{Hamming}(s,t) = \sum_i \overline{\delta}(s_i,t_i) \tag{9.76}$$

Applying the generalization distance to strings leads also to a Hamming distance (up to a factor 2).

Example 9.26. Consider the strings `abcdefgh` and `xbcdyfgh`. Their *lgg* would be `-bcd-fgh` of size 6, and hence the generalization distance between these two strings is 4, whereas the Hamming distance between them is 2.

More popular distances for strings (certainly in the field of bioinformatics) are edit distances, which also produce *alignments*. They do not usually require the strings to be of equal length. The best known example of such a distance is the Levenshtein distance [1966], which formed the basis for many routinely applied techniques in bioinformatics applications, such as the famous Blast algorithm [Altschul et al., 1990]. More formally, the edit distance between two strings is defined as the minimum number of operations needed to turn one string into the other. The allowed operations are the insertion, deletion and substitution of a symbol in Σ.

Example 9.27. For instance, the string `artificial` (English) can be turned into `artificieel` (Dutch) using two operations. Inserting an `e` and substituting an `a` by an `e`. If one assumes that all operations have unit cost, the Levenshtein distance $d_{lev}(\texttt{artificieel}, \texttt{artificial}) = 2$. The parts of the two strings that match are typically depicted as an alignment. For our example, there are two possible alignments:

```
artificieel     artificieel
|||||||| +|     ||||||||+ |
artifici-al     artificia-l
```

From these alignments, `artificil` is a minimally general generalization, which is called the *least common subsequence* in the literature on string matching. The matched part is indicated by the vertical bars. This edit-distance is a generalization distance, as shown by De Raedt and Ramon [2008]. Like the Hamming distance, taking the length of the string as size yields an edit distance with cost 2 for each operation.

The Levenshtein distance can be computed in polynomial time using a simple application of dynamic programming [Cormen et al., 2001]. For two strings $s_1 \cdots s_n$ and $t_1 \cdots t_m$, it employs an $(n+1) \times (m+1)$ matrix M; which is defined as follows (assuming unit costs for the edit operations):

$$
\begin{aligned}
M(0,0) &= 0 \\
M(i,0) &= i \text{ for } (n \geqslant i \geqslant 1) \\
M(0,j) &= j \text{ for } (m \geqslant j \geqslant 1) \\
M(i,j) &= \min \begin{cases} M(i-1,j-1) + \overline{\delta}(s_i,t_j) & \\ M(i-1,j) + 1 & \text{insert for } i,j \geqslant 1 \\ M(i,j-1) + 1 & \text{delete} \end{cases}
\end{aligned}
\tag{9.77}
$$

An example computation is depicted in Table 9.1.

Table 9.1. Computing the Levenshtein distance

		a	r	t	i	f	i	c	i	a	l
	0	1	2	3	4	5	6	7	8	9	10
a	1	0	1	2	3	4	5	6	7	8	9
r	2	1	0	1	2	3	4	5	6	7	8
t	3	2	1	0	1	2	3	4	5	6	7
i	4	3	2	1	0	1	2	3	4	5	6
f	5	4	3	2	1	0	1	2	3	4	5
i	6	5	4	3	2	1	0	1	2	3	4
c	7	6	5	4	3	2	1	0	1	2	3
i	8	7	6	5	4	3	2	1	0	1	2
e	9	8	7	6	5	4	3	2	1	1	2
e	10	9	8	7	6	5	4	3	2	2	2
l	11	10	9	8	7	6	5	4	3	3	2

The best aligment can be obtained by storing in each cell $M(i,j)$ of the matrix a backward pointer to the previous best cell, that is, that points to that cell which yielded the minimum value in the recursive case. When the algorithm terminates, the backwards pointers can be followed from the cell $M(n+1, m+1)$, which contains the minimum edit-distance.

Exercise 9.28. Compute the Levenshtein distance and the least common subsequence of `machine learning` and `machinelles lernen`.

9.5.5 Atoms and Trees

Shan-Hwei Nienhuys-Cheng [1997] introduced the following metric for ground atoms or terms $d_{term}(t, s)$

$$d_{term}(t,s) = \begin{cases} \overline{\delta}(t,s) & \text{if } t \text{ and } s \text{ are constants} \\ \overline{\delta}(g,h) & \text{if } t = g(t_1, \cdots, t_k) \text{ and } s = h(s_1, \cdots, s_m) \\ & \quad \text{and } g \neq h \\ \frac{1}{2k}\sum_i d_{term}(t_i, s_i) & \text{if } t = g(t_1, \cdots, t_k) \text{ and } s = g(s_1, \cdots, s_k) \end{cases} \tag{9.78}$$

One can arrive at this distance in various possible ways. First, following the idea of decomposition, there is a clear analogy with the kernel K_{term} in Eq. 9.45. The key difference other than the use of $\overline{\delta}$ instead of δ lies in the factor $\frac{1}{2k}$, which is needed to obtain a metric that satisfies the triangle inequality. Second, the distance can be viewed as a matching or edit-distance, cf. Eq. 9.73, where the cost of the matching is 0 and the edit cost for extending the common part by inserting arguments in the term depends on the position and depth of the term as indicated by the factor $\frac{1}{2k}$.

From a logical perspective, there is one drawback of the distance d_{term} (and also of the kernel K_{term}): they take into account neither variables nor unification.

Example 9.29.

$$d_{term}(r(a,b,d), r(a,c,c)) = \frac{2}{6}$$

and

$$d_{term}(r(a,b,b), r(a,c,c)) = \frac{2}{6}$$

This is counterintuive as the similarity between $r(a,b,b)$ and $r(a,c,c)$ is larger than that between $r(a,b,d)$ and $r(a,c,c)$. This is clear when looking at the *lgg*s. The *lgg* of the second pair of terms is more specific than that of the first one. Therefore one would expect their distance to be smaller.

Various researchers such as Hutchinson [1997] and Ramon [2002] have investigated distance metrics to alleviate this problem. To the best of the author's knowledge this problem has not yet been addressed using kernels. The approaches of Hutchinson [1997] and Ramon [2002] rely on the use of the *lgg* operation that was introduced in Chapter 5. More specifically, by using as matching operation the *lgg* and then applying the matching distance of Eq. 9.73, one obtains:

$$d_{lgg}(t,s) = c\big(lgg(t,s) \rhd t\big) + c\big(lgg(t,s) \rhd s\big) \tag{9.79}$$

So, the distance d_{lgg} measures how far the terms are from their lgg, and $c(lgg(t,s) \rhd t)$ indicates the cost for specializing the lgg to t. This can be related to the size of the substitution θ needed so that $lgg(t,s)\theta = t$, as the following example illustrates.

Example 9.30. Applying this idea to the instances in the previous example yields

$$d_{lgg}(r(a,b,d),r(a,c,c)) = c\big(r(a,X,Y) \rhd sr(a,b,d)\big)+c\big(r(a,X,Y) \rhd r(a,c,c)\big)$$

and

$$d_{lgg}(r(a,b,b),r(a,c,c)) = c\big(r(a,X,X) \rhd r(a,b,b)\big)+c\big(r(a,X,X) \rhd r(a,c,c)\big)$$

Under any reasonable definition of the cost c, the latter expression will be smaller than the former. For instance, Hutchinson uses as c a measure on the size of the substitutions; more specifically, he assigns a weight to each functor and constant symbol, and then takes the sum of the symbols occurring on the right-hand side of the substitutions. In the example, this yields $c(r(a,X,X) \rhd r(a,b,b)) = size(\{X/a\}) = w_b$, and $c(r(a,X,Y) \rhd r(a,b,d)) = size(\{X/b,Y/d\}) = w_b + w_c$.

The distances studied by Hutchinson [1997] and Ramon [2002] capture some interesting logical intuitions but still exhibit some problems (for instance, how to measure the distance between $p(X,Y,Z)$ and $p(W,W,W)$). Furthermore, they are also quite involved, especially when applied to clauses rather than terms or atoms, which explains why we do not discuss them in more detail here.

9.5.6 Graphs

To develop a metric on graphs, it is useful to look for a notion of mimimally general generalization, or alignment, and to apply the generalization distance. For graphs, a natural notion is given by the maximal common subgraph.

Formally, a *maximal common subgraph* $m(G_1, G_2)$ of two graphs G_1 and G_2 is a graph G such that 1) there exist subgraph isomorphisms from G to G_1 and from G to G_2 and 2) there exists no graph containing more nodes than G that satisfies 1). Like kernels, we employ the restricted notion of induced subgraph ismorphism here. The notion of a maximal common subgraph is illustrated in Fig. 9.8.

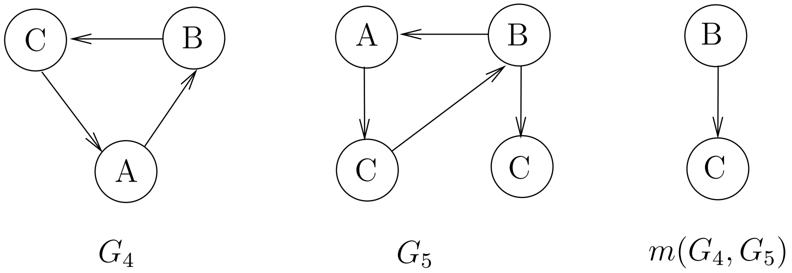


Fig. 9.8. The maximal common subgraph $m(G_4, G_5)$ of G_4 and G_5

The computation of a maximal common subgraph is an NP-complete problem and the maximal common subgraph of two graphs is not necessarily unique. Notice that a maximal common subgraph is uniquely characterized by and can therefore be represented by the (maximal) subgraph isomorphism f from G_1 to G_2 that maps vertices in G_1 to vertices in G_2. Such a maximal subgraph isomorphism is computed using a backtracking algorithm due to [McGregor, 1982], of which a variant along the lines of [Bunke et al., 2002] is summarized in Algo. 9.3.

The algorithm repeatedly adds a pair of nodes $(n_1, n_2) \in V_1 \times V_2$ to the function f, and when doing so it ensures that the resulting mapping is a feasible subgraph isomorphism by testing that the mapping f is injective and that for any two tuples (n_1, m_1) and $(n_2, m_2) \in f$: $(n_1, n_2) \in E_1$ if and only if $(m_1, m_2) \in E_2$. If the cardinality of the resulting mapping f_m is larger than that of previously considered subgraph ismorphisms, then the maximal subgraph isomorphism and corresponding size are updated. Finally, if the function f_m can still be expanded because there are still unmatched and untried vertices in V_1, and one cannot prune these refinements (because the size of f_m plus the number of such vertices is larger than $maxsize$), the procedure is called recursively.

The maximal common subgraph can be used as a minimally general generalization. A natural size measure on graphs is the number of vertices they contain. It is possible to show that the size and generality relation satisfy the necessary requirements for generalization distances; cf. [De Raedt and Ramon, 2008]. Therefore, the distance

$$d_{graph}(G_1, G_2) = |G_1| + |G_2| - 2|mcs(G_1, G_2)| \tag{9.80}$$

is a metric. Furtheremore, its normalized form

$$d_{graph-nrom}(G_1, G_2) = 1 - \frac{|mcs(G_1, G_2)|}{\max(|G_1|, |G_2|)} \tag{9.81}$$

corresponds to a well-known metric introduced by Bunke and Shearer [1998]. This distance can also be related to some type of edit-distance because the

$d_{graph-norm}(G_1, G_2)$ captures the proportion of common vertices. Bunke and Shearer argue that edit distances for graphs are – depending on the choice of cost function – not always metrics.

Example 9.31. Using these notions, $d_{graph}(G_4, G_5) = 3 + 4 - 2 \times 2 = 3$ and $d_{graph-norm}(G_4, G_5) = 1 - \frac{2}{4} = 0.5$.

Algorithm 9.3 The function $mcs(f)$ that computes a maximal subgraph isomorphism f_m of two graphs $G_i = (V_i, E_i, l_i)$ and is called with the empty relation. It uses the global parameters *maxsize*, which represents the cardinality of the relation f_m and which is initialized to 0, and *maxisomorphism*, which represents the maximal subgraph isomorphism and which is initialized to $\emptyset$

```
while there is a next pair $(n_1, n_2) \in (V_1 \times V_2)$ with $l_1(v_1) = l_2(v_2)$ do
  if feasible($f \cup \{(n_1, n_2)\}$) then
    $f_m := f \cup \{(n_1, n_2)\}$
    if $|f_m| > maxsize$ then
      $maxsize := size(f_m)$
      $maxisomophism := f_m$
    end if
    if nopruning($f_m$) and expansionpossible($f_m$) then
      call $mcs(f_m)$
    else
      backtrack on $f_m$
    end if
  end if
end while
return $f_m$
```

Exercise 9.32. Apply Algo. 9.3 to compute the maximal common subgraph of G_4 and G_5 as in Fig. 9.8.

Exercise 9.33. * Discuss the relationship between the concept of maximal common subgraph and maximally general generalization under *OI*-subsumption. (Hint: represent graphs as sets of facts.)

9.6 Relational Kernels and Distances

Whereas kernels and distances have been developed and studied for a wide variety of structured data types, such as vectors, sets, strings, trees and graphs, kernels and distances that work directly on logical and relational representations, say, clausal logic, have proven to be more challenging to develop.

Therefore, several researchers have looked into heuristic approaches to this problem.

One popular heuristic approach, due to Bisson [1992a,b], and refined by Emde and Wettschereck [1996] and Kirsten et al. [2001], has been succesfully used in a series of distance-based learning methods targeting both classification and clustering. As in the learning from entailment setting, examples (ground facts) as well as a background theory are given. The computation of a distance measure proceeds by computing for each example a tree, and then applying a distance on trees to the corresponding examples. The resulting distance is not a metric. The idea is illustrated in Ex. 9.34 due to Kirsten et al. [2001].

Example 9.34. Let us assume that the instance positive(set1, 125, personal) is given together with the background theory:

$$\begin{array}{l}
\mathsf{cheese(set1, camembert, 150, france)} \leftarrow \\
\mathsf{vineyard(gallo, famous, large, usa)} \leftarrow \\
\mathsf{vineyard(mouton, famous, small, france)} \leftarrow \\
\mathsf{wine(set1, gallo, 1995, 0.5)} \leftarrow \\
\mathsf{wine(set1, mouton, 1988, 0.75)} \leftarrow
\end{array}$$

It describes a particular basket from a gourmet shop. The instance can be turned into a tree, of which it forms the root. Subtrees are then constructed by following the foreign links on the arguments of the instance. For instance, the argument set1 occurs in three further facts, and hence these facts can be turned into children of set1. Furthermore, these facts in turn contain arguments, such as gallo, camembert and mouton, that occur in further facts, and hence can be used at the next level in the tree. The resulting tree is shown in Fig. 9.6. The process of following foreign links is typically terminated when a particular depth is reached. Given two such trees, corresponding to two instances, standard tree kernels and distance measures based on the idea of decomposition can be used.

A closely related approach is taken by Passerini et al. [2006], who provide special *visitor* predicates in addition to the background theory and the instances. The visitor predicates are then called on the instances, and the resulting proof trees are computed and passed on to a kernel. In this way, the visitor predicates encode the features that will be used by the kernel and the way that the instances will be traversed.

Transforming instances and background theory into trees forms an interesting alternative to traditional propositionalization. Rather than targeting a purely flat representation, an intermediate representation is generated that still possesses structure but is less complex than the relational one.

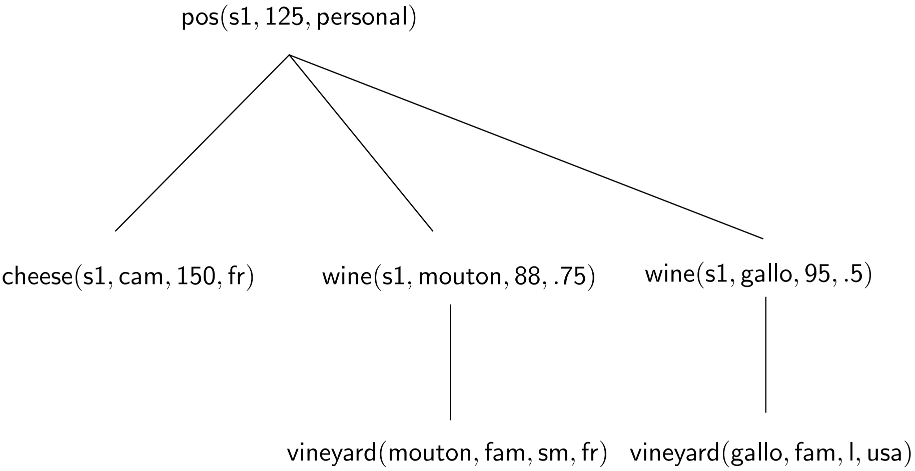

pos(s1, 125, personal)
cheese(s1, cam, 150, fr)
wine(s1, mouton, 88, .75)
wine(s1, gallo, 95, .5)
vineyard(mouton, fam, sm, fr)
vineyard(gallo, fam, l, usa)

Fig. 9.9. A tree computed from an instance and the background theory. Reproduced with permission from [Kirsten et al., 2001]

9.7 Conclusions

In this chapter we have studied how logical and relational representations can be used together with support vector machines and instance-based learning techniques. After a short introduction to these classes of machine learning techniques, we have studied how distance metrics and kernels can be defined. For kernels, convolution and decomposition can be used to define kernels for complex data structures in terms of kernels for simple data structures. For distance metrics, we have investigated the relation of edit-distances to the generality relation, and we have argued that, under certain conditions, distances can be generated using a size measure and the minimimally general generalization operation. Variants of this idea, based on the notion of matching, have also been introduced. These principles have then been applied to some well known data structures, such as sets, strings, trees and graphs.

9.8 Bibliographical and Historical Notes

Many contempprary machine learning and data mining techniques employ kernel or distances. Kernels provide a basis for the popular support vector machines (see [Cristianini and Shawe-Taylor, 2000] for a gentle introduction) and distances for case-based reasoning [Aamodt and Plaza, 1994].

The introduction to kernels in Sect. 9.2 follows the introductory chapter of [Schölkopf and Smola, 2002]; and the overview of kernels for structured data follows the exposition in the overview article of [Gärtner, 2003]. A good and rather complete overview of distances and metrics for structured data is contained in [Ramon, 2002], which also forms the basis for the present overview, though some new materials and insights from [De Raedt and Ramon, 2008] have been incorporated.

Within logic learning, the first distance measure for relational data was contributed by Bisson [1992b] for use in clustering, and later employed in the RIBL system of Emde and Wettschereck [1996]. RIBL is a relational k-nearest neighbour algorithm. Further contributions to distance-based learning were made by Kirsten and Wrobel [2000] in the context of clustering (k-means and k-medoid) and by Horvath et al. [2001] for instance-based learning. However, the distances employed in this line of work were not metrics. The systems based on this type of distance performed quite well in practice, but at the same time they motivated a more theoretical stream of work that focussed on developing metrics for relational representations, in particular the work of Ramon [2002] and Nienhuys-Cheng [1997, 1998].

Due to the importance of structured data, there is a lot of interest in developing kernels for different types of data structures. A seminal contribution in this regard is the notion of a convolution kernel due to Haussler [1999]. Since then, many kernels have been developed for a variety of data structures, including multi-instance learning [Gärtner et al., 2002], graphs [Kashima et al., 2003, Gärtner et al., 2003] and hypergraphs [Wachman and Khardon, 2007] and terms in higher-order logic [Gärtner et al., 2004]. To the best of the author's knowledge, there have only been a few attempts to integrate kernel methods with logic; cf. Muggleton et al. [2005], Passerini et al. [2006], Landwehr et al. [2006]. Some recent contributions are contained in an ongoing series of workshops on *Mining and Learning with Graphs* [Frasconi et al., 2007].

Various exciting applications of distance and kernels for structured, relational data exist. Well known are the applications to structure activity relationship prediction [Ralaivola et al., 2005], analysis of NMR spectra [Džeroski et al., 1998], and classical music expression analysis [Tobudic and Widmer, 2005].

10

Computational Aspects of Logical and Relational Learning

Logical and relational learning are known to be computationally more expensive than traditional propositional approaches. This chapter investigates computational aspects of logical and relational learning. More specifically, Sect. 10.1 investigates these issues from a pragmatic perspective. It starts by empirically investigating the computational cost of θ-subsumption, which lies at the heart of logical and relational learning, as it is the most common coverage relation. Afterward, implementation and optimization issues of logical and relational learning systems are discussed. In Sect. 10.2, logical and relational learning are investigated from a theoretical point of view, and, in particular, various computational learning theory settings, and positive as well as negative results, are presented.

10.1 Efficiency of Relational Learning

There is a trade-off in computer science between efficiency and expressiveness. The more expressive a representation language, the more computationally expensive it is to use it. This universal wisdom, of course, also applies to logic and theorem proving, as well as to logical and relational learning. In this section, we first identify some computational problems encountered by typical logical or relational learning systems, analyze them, and then present some typical solutions.

There are a large number of factors that influence the computational resources a relational learning system needs to tackle a learning problem, including the data set, the background knowledge, the hypothesis space (determined by the bias and the alphabet), and the search strategy. In previous chapters, we have already addressed the effects of bias, heuristics, search strategy and operators; in the present section, we focus on one of the key steps in any relational learning problem: the coverage test. Testing whether a hypothesis covers a particular example is one of the key factors responsible for the computational complexity of relational learning systems: it is computationally expensive, typically NP-hard or even undecidable in the general case, and repeated a large number of times in any relational learning system. So, if one can optimize the coverage test, one can also improve the overall performance of the relational learning system.

10.1.1 Coverage as θ-Subsumption

In general, when working with definite clause logic (allowing for functors, recursion and background knowledge), the typical coverage test ($B \cup H \models e$, where H and B are sets of definite clauses and e is a definite clause) is undecidable. From a practical perspective, it often suffices to consider a purely relational setting, in which H defines a single predicate and neither are there functors nor is there recursion. In this case, coverage testing corresponds to θ-subsumption. Indeed, to see this, consider the Bongard problem in Ex. 4.11 and Fig. 4.4 as a typical illustration of this setting.

Example 10.1. Consider the two basic encodings (corresponding to Ex. 4.11 and 4.27 respectively). Using the first representation, the example e can be encoded as a clause, for instance, $\mathsf{pos} \leftarrow \mathsf{circle(c)}, \mathsf{triangle(t)}, \mathsf{in(t, c)}$ and the hypothesis would consist of clauses such as $\mathsf{pos} \leftarrow \mathsf{circle(X)}, \mathsf{triangle(T)}, \mathsf{in(T, C)}$. Clearly, the hypothesis covers the example if and only if there is a clause $c \in H$ that θ-subsumes e. Using the second representation, identifiers are added to the facts and the resulting example is encoded as $\mathsf{pos(e2)}$ and the facts $\mathsf{in(e2, t1, c1)}, \mathsf{circle(e2, c1)}, \mathsf{triangle(e2, t2)}$ are part of the (extensional) background theory B. To test whether a hypothesis H defining $\mathsf{pos/1}$ covers the example, consider each clause $c \in H$ in turn, and test whether $head(c)$ unifies with the example (yielding the substitution θ), and if it does, test whether $body(c)\theta$ succeeds in the background theory. For instance, in our example, $body(c)\theta$ is $\leftarrow \mathsf{circle(e2, X)}, \mathsf{triangle(e2, T)}, \mathsf{in(e2, T, C)}$, and it succeeds on our background theory. If the background theory is extensional, evaluating the query corresponds again to performing a θ-subsumption test; cf. also Ex. 5.17.

So, in the simplest relational learning setting, coverage testing corresponds to θ-subsumption. In more complex relational settings, involving intensional background knowledge, functors, or recursion, coverage testing even becomes computationally harder. θ-subsumption is a known NP-complete problem, which explains why relational learning is computationally much more expensive than learning within propositional representations. At the same time, because θ-subsumption is central to relational learning, it is worthwhile to empirically analyze it and to optimize θ-subsumption tests in relational learning systems.

10.1.2 θ-Subsumption Empirically

Over the past decade, several NP-complete problems, such as 3-SAT (determining whether a set of propositional clauses containing at most three literals is satisfiable), have been empirically investigated. For θ-subsumption, this work was carried out by Giordana and Saitta [2000], Botta et al. [2003]. In these empirical investigations, many problem instances are generated at random and are characterized along particular dimensions. For θ-subsumption, one such experiment is summarized in Figs. 10.1 and 10.2. It concerns testing whether a query q succeeds (or θ-subsumes) an extensional database B of facts. In the experiment, a number of simplifying assumptions are made: all predicates are binary and contain N facts, the queries contain each predicate at most once, the queries are *connected* (that is, each literal in the query contains at least one variable that occurs also in another literal[1]), and there is only a single type. The following parameters are used to characterize the different problem instances:

- the number n of variables in q,
- the number m of predicate symbols in q,
- the number L of constants in B, and
- the number N of atoms for predicate symbols in q.

The coverage test employed used a simple variant of backtracking (cf. [Giordana and Saitta, 2000]). Using more advanced techniques, some of which will be discussed below, it is sometimes possible to get better runtimes, though the peaks and the forms of the graphs are still very similar.

In Fig. 10.1 the probability P_{sol} that a random query succeeds in a random database is plotted as a function of (m, L) for a fixed $(n = 10, N = 1, 00)$ pair, whereas Fig. 10.2 shows the average running time of a randomly selected covers test plotted in the same dimensions. The figures exhibit the typical *phase-transition* behavior. This means that problem instances fall into three main categories:

- the *under-constrained* problems, where P_{sol} is close to 1, where there are typically many solutions that can be found quickly by a solver,
- the *over-constrained* problems, where P_{sol} is close to 0, where there are typically no solutions and solvers rapidly discover this,
- the problems on the *phase-transition*, where P_{sol} is close to 0.5, where there is typically either no solution or only a few solutions, and solvers have to consider many combinations before finding the solution or being able to conclude that no solution exists.

Even though θ-subsumption is NP-complete, many problems still turn out to be easy to solve. This is not contradictory, as NP-completeness is a worst-case result and there do exist problems that are very hard to solve, in particular, those that lie on the phase-transition.

[1] If the query would not be connected, it could easily be optimized; cf. also below.

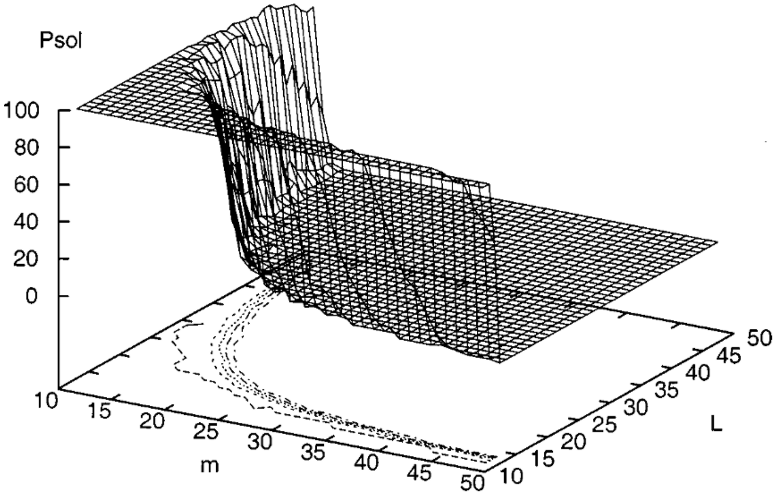


Fig. 10.1. (Reprinted with permission from [Giordana and Saitta, 2000], page 223). The probability P_{sol} that a random query succeeds in a random database, averaged over 1000 pairs (q, B) for each (m, L) point. Also, $n = 10$ and $N = 100$. On the horizontal plane, the contour plots points corresponding to P_{sol} values in the interval (0.1,0.9)

10.1.3 Optimizing the Learner for θ-subsumption

Because θ-subsumption and coverage testing are computationally expensive and have to be performed many times throughout the learning process, it is essential to optimize these types of tests in logical and relational learners. From the author's experience, the time spent by a logical learning system on such tests accounts typically for about 90% of the overall running time. So, by optimizing relational learning systems with regard to θ-subsumption and coverage testing, a lot of time can be saved. There are many ways to realize this.

First, one can try to minimize the number of coverage tests as much as possible. One way of realizing this associates with each hypothesis a list of identifiers of covered examples. When searching from general to specific, the coverage of a newly generated candidate hypothesis s can be determined by testing the coverage of the examples associated with the more general hypothesis g that s was generated from. A variant of this scheme can be used when searching for frequent queries if one is not interested in the exact frequencies

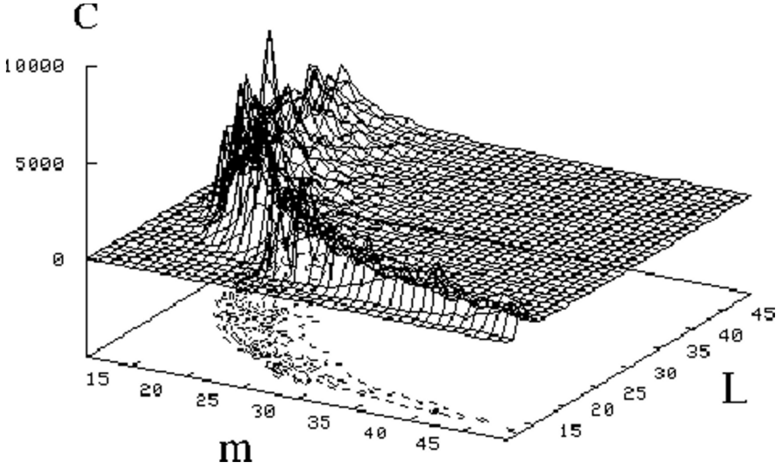


Fig. 10.2. (Reprinted with permission from [Giordana and Saitta, 2000], page 225.) The computational cost for a single coverage test with a Monte Carlo method (in seconds on a Sparc Enterprise 450).

of the queries. Instead of keeping track of all covered examples, it suffices to keep track of the first m covered examples, where m corresponds to the frequency threshold. Upon refining a query, one first checks the coverage with regard to the m examples on the list, and only when needed considers further examples. Alternatively, the number of coverage tests can be reduced using *sampling*. The idea is that rather than computing the evaluation functions used throughout the search exactly, we estimate the number of covered examples is estimated by sampling from the data set. Various schemes for sampling have been proposed in the data mining literature, and can be applied in a relational setting as well; see, for instance, [Toivonen, 1996, Srinivasan, 1999].

Second, one can try to make the coverage and θ-subsumption tests simpler. Using the experimental setting mentioned above, the computational complexity of θ-subsumption grows as a function of the parameters n, N, m and L. So, by reducing the size of the queries and background knowledge, we improve the performance of the learning system. This insight implies that for the Bongard problem discussed in Ex. 10.1, the first representation is more efficient than the second one, because in general it is beneficial to encode each example separately, rather than to merge their facts into a larger database. Separate encodings of the examples are natural within the learning from interpretations settings and within the setting where each example is completely encoded in a single clause (as in the first representation of Ex. 10.1). A further advantage of separately encoding the examples is related to memory management. When working with a huge data set, rather than having to run coverage tests on the overall database, one can retrieve the example from the database and run the coverage test on a much smaller database. This possibility is often combined with the idea of *inverting the loops* in data mining systems in order to minimize the number of passes through the database. Indeed, many traditional machine learning systems possess a subroutine that operates as follows: for each candidate refinement h, for each example e, test whether e is covered by h. Using this scheme, the number of passes through the database is equal to the number of refinements considered by the subroutine. When inverting the two loops, one obtains: for each example e, for each hypothesis h, test whether e is covered by h. As a consequence, one needs only a single pass through the database, which is much more efficient according to database principles.

Third, one can optimize the θ-subsumption procedure itself. In Chapter 2, the standard SLD-resolution procedure employed in Prolog for executing queries on a database was introduced. It essentially employs backtracking to answer queries. Backtracking is not necessarily the most efficient way of answering queries on a relational database.

Example 10.2. To illustrate this point, consider the (somewhat artificial) query

$$\leftarrow \mathsf{rank(X), suit(Y), X = f, Y = s}$$

and the simple card database shown below:

$$\begin{array}{ll}
\mathsf{suit(d)} \leftarrow & \mathsf{suit(c)} \leftarrow \\
\mathsf{suit(h)} \leftarrow & \mathsf{suit(s)} \leftarrow \\
\mathsf{rank(a)} \leftarrow & \mathsf{rank(10)} \leftarrow \\
\mathsf{rank(7)} \leftarrow & \mathsf{rank(k)} \leftarrow \\
\mathsf{rank(8)} \leftarrow & \mathsf{rank(q)} \leftarrow \\
\mathsf{rank(9)} \leftarrow & \mathsf{rank(f)} \leftarrow
\end{array}$$

Using backtracking, the standard Prolog engine will generate 32 substitutions (that is, all cards) before finding the unique solution. On the other hand, by simply reordering the literals as in the query

$$\leftarrow \mathsf{X = f, Y = s, rank(X), suit(Y)}$$

only a single substitution needs to be generated.

Exercise 10.3. Discuss why Prolog generates all 32 substitutions for the above example.

The example motivates some worthwhile optimizations to perform before executing the queries. One such optimization splits the query into maximally connected sub-queries. Recall that a connected query is one where all literals share variables with at least one other literal. In our example, there are two maximally connected sub-queries:

$$\leftarrow \mathsf{rank}(\mathsf{X}), \mathsf{X} = \mathsf{f}$$
$$\leftarrow \mathsf{suit}(\mathsf{Y}), \mathsf{Y} = \mathsf{s}.$$

These sub-queries can be executed independently of one another and the resulting answer substitutions can be combined. Using this optimization on the original query only eight substitutions for X and four for Y are considered. This optimization can also be performed iteratively. Given a connected query $\leftarrow l_1, \cdots, l_n$, one can first solve l_1 yielding substitution θ_1, and then answer the remaining query $\leftarrow l_2\theta_1, \cdots, l_n\theta_1$. Even though the original query $\leftarrow l_1, \cdots, l_n$ may be connected, instantiating l_1 may cause the remaining query $\leftarrow l_2\theta_1, \cdots, l_n\theta_1$ to be no be longer connected. If this is the case, it may be worthwhile to divide the remaining query again. Another possible optimization concerns the automatic reordering of the literals in the query according to the growing number of possible answers or tuples. For our example query, the literal for rank has 8, for suit, 4, and for each occurrence of =, 1 possible answers. Using this optimization, the original query is rewritten into

$$\leftarrow \mathsf{X} = \mathsf{f}, \mathsf{Y} = \mathsf{s}, \mathsf{rank}(\mathsf{X}), \mathsf{suit}(\mathsf{Y})$$

which directly yields a solution without backtracking. The above two optimizations can be combined by first dividing the query into sub-queries and then reordering the literals in the resulting sub-queries. Optimizations along this line (and corresponding Prolog code for meta-interpreters) are discussed extensively by [Costa et al., 2003b]. In the author's experience, using such optimizations (by, for instance, incorporating some of the readily available meta-interpreters by [Costa et al., 2003b]) can yield speedups of the overall learning system of a factor 5 to 10.

Rather than optimizing the way Prolog executes queries, queries can be executed in an alternative way. A first alternative employs a relational or deductive database management system instead of a Prolog implementation to determine which examples are covered by the hypotheses. In contrast to typical Prolog engines, database management systems are optimized to deal with large databases, and hence, when the data set mined is large, the database system may produce results when Prolog implementations run into problems due to the size of the database. However, naively coupling a relational learning system to a traditional database management system that targets data sets that reside on disk is not necessarily a solution as it may result in a significant overhead. According to the author's experience, using a naive coupling with such a database management system for carrying out coverage tests may slow down the system by a factor of 20 for molecular databases. One reason for this is that database systems are optimized to return all answers to a particular query, whereas a Prolog engine is optimized to return the first few answers only. To decide whether a hypothesis covers a particular example, only one answer is needed; cf. also below. At present, the question of how to efficiently couple a relational learner to a database system remains open. One promising direction may be to use in-memory databases and another possible approach is sketched in Yin et al. [2006].

A second alternative is to view θ-subsumption as a constraint satisfaction problem:[2]

Example 10.4. Reconsider the Bongard problems of Ex. 10.1. Consider the example database

$$\mathsf{circle(c1), circle(c2), triangle(t1), triangle(t2), in(t1, c2)}$$

and the query

$$\leftarrow \mathsf{triangle(X), triangle(Y), in(X, Y).}$$

This query has two variables (X and Y). Their types correspond to the domains, that is, they specify the values they can take. Assume that there is only one type, $dom(\mathsf{X}) = \{\mathsf{t1}, \mathsf{t2}, \mathsf{c1}, \mathsf{c2}\} = dom(\mathsf{Y})$. The unary predicates then specify constraints on a single variable, whereas the binary predicates specify constraints on pairs of variables.

The above example demonstrates that θ-subsumption problems can be phrased within a constraint satisfaction framework, and hence that standard constraint satisfaction principles and solvers can be applied on θ-subsumption. This view has been explored by various researchers such as Botta et al. [2003], Maloberti and Sebag [2004], and both stochastic as well as deterministic constraint satisfaction solvers have been successfully applied to θ-subsumption.

Thirdly, rather than optimizing the way individual θ-subsumption or coverage tests are performed, one can try to optimize the overall mining process. A key observation in this regard is that many queries and hypotheses that are generated during the search are closely related to one another.

Example 10.5. Consider the query $\leftarrow \mathsf{in(X, Y)}$ and some of its refinements:

$$\begin{array}{l} \leftarrow \mathsf{in(X, Y), triangle(X)} \\ \leftarrow \mathsf{in(X, Y), circle(X)} \\ \leftarrow \mathsf{in(X, Y), triangle(X), red(Y)} \\ \leftarrow \mathsf{in(X, Y), circle(X), red(Y)} \end{array}$$

When evaluating these queries independently of one another, a lot of redundant computations are repeated. There are several ways of tackling this problem. A first possibility is to simply store the intermediate results for later use. This is for instance done in Quinlan's Foil [Quinlan, 1990], where all answer substitutions to the more general query would be stored, and then used as the starting point for evaluating its refinements. Whereas this approach is possible for greedy search methods, storing these intermediate results for complete search methods seems prohibitive because of the memory requirements. A second possibility, which forms the basis of the so-called *query packs* [Blockeel et al., 2002], is to rewrite the query in such a way that common parts of the query are shared.

[2] The reader unfamiliar with constraint satisfaction problems may want to consult [Russell and Norvig, 2004] for an excellent introduction.

Example 10.6. The previous set of queries can be written using a query pack:

$$\leftarrow \mathsf{in}(\mathsf{X},\mathsf{Y}), (\mathsf{true\ or\ }(\mathsf{triangle}(\mathsf{X}), (\mathsf{true\ or\ red}(\mathsf{Y}))) \mathsf{\ or\ } (\mathsf{circle}(\mathsf{X}), (\mathsf{true\ or\ red}(\mathsf{Y}))))$$

The or closely corresponds to Prolog's or but has a slightly different semantics.

Using the query packs, the common parts of the queries are executed less often than if the queries are executed individually. This is clear when looking, for instance, at the in(X, Y) atom. This atom is only called once, and the substitutions for which it succeeds are propagated to the next levels. The query pack approach is implemented within a special-purpose Prolog engine called ILProlog [Blockeel et al., 2002], which is optimized to count the number of examples covered by the different branches in query packs.

A final optimization is relevant when the background knowledge is intensional, that is, contains the definition of view predicates. For instance, when considering molecular application as illustrated in Ex. 4.22, there may be definitions of functional groups, such as ring types. The computation of which ring structures are present in a particular molecule can be non-trivial (as this is essentially a test for subgraph isomorphism). Therefore, it often pays off to precompute, that is, to *materialize*, such intensional predicates if memory allows for this. Alternatively, a technique called *tabling* can be employed. Tabling does not precompute the predicates, but rather memorizes those computations it has already performed. For instance, in the molecular example, it might have computed that there is a benzene ring in molecule 225 and memorized that fact on the fly for later use. Tabling is related to dynamic programming and is especially important in the context of probabilistic logic learning; cf. Chapter 8.

As the reader has probably noticed, there are many possible ways for improving the performance of logical and relational learning systems. Which approach is to be preferred often depends on the particular type of data set considered. Relevant questions are whether the data set is purely extensional or intensional, whether it contains functors, how large it is, etc. Improving the performance of logical and relational learning systems is bound to remain an active area of research in the near future as it is one of the key challenges of working with expressive representations.

10.2 Computational Learning Theory*

Whereas the previous was concerned with an empirical perspective and possible ways of improving the implementation of relational learning systems, this section takes a more theoretical perspective. More specifically, *computational learning theory* investigates classes of learning problems that are efficiently learnable. There are two issues here. First, what is the meaning of *efficient*? And second, what do we mean by *learnable*?

The first question has the standard answer in computer science, where efficient corresponds to polynomial. Two forms of efficiency are considered in computational learning theory: *sample complexity* and *computational complexity.* The sample complexity expresses the number of examples needed before a high-quality solution is obtained, whereas the computational complexity refers to the time and memory requirements of particular learning problems or algorithms.

The second question is somewhat harder to answer, because there exist many different notions of learnable within the computational learning theory literature. They all have to do with a notion of convergence to a desirable hypothesis given evidence (in the form of examples or answers to particular queries; cf. Sect. 7.3.2). A full discussion of all possibilities is outside the scope of this book,[3] though we present some key notions in the following subsection.

10.2.1 Notions of Learnability

In Chapter 7, we encountered some notions of convergence. First, there was the Horn algorithm by Angluin et al. [1992] that *exactly identifies* theories. Exact identification requires that the learner halt after processing a finite number of examples and queries and output a theory that is (logically) equivalent to the target theory. In the case of Horn, there were further guarantees concerning the complexity of the algorithm. Indeed, the number of queries as well as the computation time required were both polynomial in the size parameters of the target theory.

Second, there is the Model Inference System algorithm by Shapiro [1983], which *identifies theories in the limit.* Systems that identify theories in the limit are presented with a potentially infinite sequence of examples $e_1, ..., e_n, ...$ of the target theory T,[4] and have to output a sequence $T_1, ..., T_n, ...$ of theories such that T_i is consistent with the first i examples $e_1, ..., e_i$. A system *identifies a theory in the limit* if and only if, for all possible theories and all possible sequences of examples (in which each example eventually occurs), there is a number i such that T_i is equivalent to the target theory T and $\forall j > i : T_j = T_i$. So, systems that identify interpretations in the limit converge in a finite number of steps upon a theory that is correct. The key difference with exact identification is the system is not required to know when the point of convergence occurs.

Thirdly, there is the more recent and also more popular *PAC-learning* (*probably approximately correct* learning) setting introduced by Valiant [1984] on which we will focus in this section. Whereas the Model Inference System and the Horn algorithm employed membership and equivalence queries

[3] Actually, computational learning theory is an active research field in itself to which several textbooks have been devoted; for instance, [Natarajan, 1991, Kearns and Vazirani, 1994]. The reader may also want to consult [Mitchell, 1997] for a gentle introduction.

[4] In MIS's case the theory corresponds to an *intended interpretation*; cf. Sect. 7.3.2.

on an oracle, the PAC-learning setting models batch learning, in which the learner is presented a data set and has to output a hypothesis without further interaction with a user or oracle. At the same time, it does not require the learner to converge on a correct theory, but rather on one that is approximately correct. To this end, it is assumed that there is an unknown probability distribution $\mathcal{D}$ on the examples in $\mathcal{L}_e$, and that the examples in the data set are drawn at random according to this distribution. A hypothesis h is then *approximately correct* with regard to a target theory t if and only if

$$P_{\mathcal{D}}(\{e \in \mathcal{L}_e | \mathbf{c}(h,e) \neq \mathbf{c}(t,e)\}) < \epsilon \tag{10.1}$$

where $\mathbf{c}(h,e)$ denotes the coverage relation. To be approximately correct, the probability that a randomly selected example (according to $\mathcal{D}$) is classified differently by the learned hypothesis h and the target theory t should be smaller than a parameter ϵ.

Informally, a learner is said to learn an approximately correct hypothesis if it outputs an approximately correct hypothesis $h \in \mathcal{L}_h$ with high probability, that is, with probability larger than $1 - \delta$ for a parameter δ, and it outputs a hypothesis satisfying Eq. 10.1. As it is unreasonable to expect a learner to output an approximately correct hypothesis regardless of what the data set and the target theory are, the notion of PAC-learning also takes into account the complexity of the learning task. More formally, the size n_t of the target theory $t \in \mathcal{L}_h$ and of the (largest) examples n_e in the data set are taken into account, as are the parameters ϵ and δ. For $\mathcal{L}_h$ to be learnable, there must exist a polynomial function $m(n_t, n_e, 1/\epsilon, 1/\delta)$ such that the learner outputs an approximately correct hypothesis with high probability if the data set (drawn at random according to $\mathcal{D}$) is at least of size $m(n_t, n_e, 1/\epsilon, 1/\delta)$. Thus the learner is allowed to see more examples as the learning task (as measured by n_t and n_e) becomes more complex, or the guarantees it gives (in terms of ϵ and δ) become tighter. The final requirement for PAC-learning is that the resulting algorithm is efficient, that is, it must run in polynomial time in the parameters $n_t, n_e, 1/\epsilon, 1/\delta$ of the problem.

To summarize, a language of theories $\mathcal{L}_h$ is *PAC-learnable* from examples in $\mathcal{L}_e$ if and only if there exists a polynomial algorithm that for all theories $t \in \mathcal{L}_h$, for all probability distributions on $\mathcal{L}_e$, and for all $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$ outputs with high probability $(1 - \delta)$ a hypothesis $h \in \mathcal{L}_h$ that is approximately correct when receiving a data set that contains at least $m(n_t, n_e, 1/\epsilon, 1/\delta)$ examples.

There exist often interesting relationships among these models of learnability. For instance, [Angluin, 1987] showed that each language that can be (exactly) identified using equivalence queries is also PAC-learnable. The key idea is that each equivalence query can be replaced by drawing a set of examples and checking whether it contains a counter-example.

10.2.2 Positive Results

Despite the expressiveness of logic, several positive PAC-learning results exist.

A first class of results is obtained using *propositionalization.* For instance, the first PAC-learning result for inductive logic programming, due to Džeroski et al. [1992], concerns the most popular inductive logic programming setting, learning from entailment, as employed by systems such as FOIL [Quinlan, 1990]. Recall that in this setting examples are true and false ground facts concerning a single target predicate, and the background theory is extensional, that is, it consists of a set of ground facts. Džeroski et al. [1992] consider the class of ij-determinate clauses, introduced in the GOLEM system by Muggleton and Feng [1992]. The determinacy restriction allows one to propositionalize the learning problem using the table-based approach sketched in Sect. 4.12.1 and to guarantee that for each example there is only a single row in the resulting table. This was realized by allowing the substitutions to only follow **(n:1)** and **(1:1)** relations. More formally, a clause $h \leftarrow b_1, \ldots, b_n$ is *determinate* with regard to the background theory B if and only if after grounding $h\theta$ for all $1 \leqslant i \leqslant n$ there is at most one substitution θ_i such that

$$B \models b_1\theta\theta_1, b_2\theta\theta_1\theta_2, \ldots b_i\theta\theta_1 \ldots \theta_i \text{ and the expression is ground} \quad (10.2)$$

Example 10.7. For instance, the clause

$$\mathsf{grandfather(X,Y)} \leftarrow \mathsf{father(X,Z), father(Z,Y)}$$

is not determinate under the usual interpretation where $\mathsf{father(X,Y)}$ is true if X is the father of Y because one father X can have multiple children Z. Therefore, depending on the background theory, the requirement does not hold for the first literal (that is, for $i = 1$). On the other hand, by reordering the clause as

$$\mathsf{grandfather(X,Y)} \leftarrow \mathsf{father(Z,Y), father(X,Z)}$$

the clause is determinate, because any given person Y has exactly one father, who in turn has one father.

In addition to the determinacy restriction, Džeroski et al. [1992] (and Muggleton and Feng [1992] in their GOLEM system) consider two parameters i and j to bound the depth of the dependencies among terms in the substitutions and the arity of predicates (which in turn determine the size of the resulting table). More essential than these two complexity parameters are that the PAC-learning results for ij-determinate clauses then follow from the PAC-learnability of the problems in the table format. Džeroski et al. [1992] essentially apply the learnability result for k-DNF (that is, propositional formulae in disjunctive normal form, where each conjunction contains at most k literals). This can be realized using the notion of a prediction-preserving reducibility (cf. below).

The propositionalization approach has also been applied to the learning from interpretations setting for jk-clausal theories [De Raedt and Džeroski, 1994]. This result directly upgrades the original result by Valiant [1984] for k-CNF formulae to clausal logic. In Valliant's approach a specific-to-general algorithm is applied, which starts from the most specific k-CNF formula (consisting of the conjunction of all propositional clauses involving at most k literals), and which repeatedly generalizes the current hypothesis when a positive example is encountered that is not a model for the current hypothesis. The generalization process simply deletes all clauses from the current hypothesis that violate the positive example. This idea is carried over in jk-clausal theories, where k again limits the number of literals in a clause and j determines the maximum size of the atoms. For fixed k and j, one can then evaluate whether a finite Herbrand interpretation satisfies a jk-clause. The result can again be viewed as a propositionalization approach, but now in the query-based setting (cf. Sect. 4.12.2). Indeed, each possible jk-clause corresponds to a feature and yields the value true for an example if the example satisfies it, and false otherwise. The parameters j and k allow the table to be constructed in polynomial time. The result then follows from a corresponding PAC-learning result for monomials (or item-sets); see Ex. 10.15.

It has been argued that the learning from interpretations setting may well yield more positive results than the learning from entailment [Cohen and Page, 1995, De Raedt, 1997]. The reason is that interpretations contain more information and are also larger than the facts employed in learning from entailment. This does not only make the learning task easier but also allows the learning system more processing time.

Exercise 10.8. Illustrate Valliant's k-CNF algorithm on a simple example.

Exercise 10.9. Construct a simple example of the propositionalization approach taken in the learning from interpretations setting.

Second, there exist a number of exciting but unfortunately also rather involved results that do not rely on a propositionalization approach. Several of these are actually upgrades of the famous HORN algorithm of Angluin et al. [1992] presented in Sect. 7.4.1, or the variant given by Frazier and Pitt [1993] for learning from entailment. Some interesting positive results are given by Reddy and Tadepalli [1997, 1998], Arias and Khardon [2000], Khardon [1999]. The last approach by Roni Khardon upgrades HORN towards learning function-free Horn theories from interpretations and was implemented in the LOGAN-H system [Khardon, 2000]. Whereas the original version of this algorithm employs membership and equivalence queries, the batch variant implemented in LOGAN-H emulates these examples by drawing samples from the data.

10.2.3 Negative Results

Most PAC-learning results for relational learning are negative. Cohen and Page [1995] distinguish essentially three proof techniques for obtaining negative PAC-learning results. We closely follow the exposition by Cohen and Page [1995] to explain these.

First, if the problem of finding a consistent hypothesis (that is, a hypothesis that covers all positive examples and no negative example) cannot be solved in polynomial time, then the problem is not PAC-learnable. Let us illustrate this using a result due to Kietz [1993], Kietz and Džeroski [1994], derived from [Haussler, 1989]. Kietz showed that the well-known SAT problem is polynomially reducible to the consistency problem for the language $\mathcal{L}^{nd}_{12}$. In this problem, all examples and hypotheses are *single* clauses, the predicates in the body are of maximum arity 2, the depth of terms in the clause is at most 1[5] and the determinacy restriction is not imposed. In addition, the coverage relation is θ-subsumption. Because SAT is an NP-complete problem, it follows that the consistency problem is NP-complete as well, and therefore this language is not PAC-learnable. This reduction is presented in the example below. It can be skipped (as well as the two exercises following it) without loss of continuity.

Example 10.10. ** (From Kietz and Džeroski [1994].) The reduction from SAT to the consistency problem for $\mathcal{L}^{nd}_{12}$ can be defined as follows. In SAT, one is given a set $V = \{v_1, \cdots, v_n\}$ of boolean variables and a set of clauses $C = \{C_1, \cdots, C_m\}$ over V, and the question is whether there exists a truth assignment to V that satisfies the clauses in C.

The SAT problem is reduced to the consistency problem as follows. The positive examples are (for all $1 \leqslant i \leqslant n$) of the form

$$\begin{array}{l} h(c_i) \leftarrow \\ \quad p(c_i, c_{i1}), p(c_i, c_{i2}), \\ \quad t_1(c_{i1}), \cdots, t_{i-1}(c_{i1}), t_{i+1}(c_{i1}), \cdots, t_n(c_{i+1}), f_1(c_{i1}), \cdots, f_n(c_{i1}), \\ \quad t_1(c_{i2}), \cdots, t_n(c_{i2}), f_1(c_{i2}), \cdots f_{i-1}(c_{i2}), f_{i+1}(c_{i2}), \cdots, f_n(c_{i2}) \end{array}$$

and the negative example is

$$\begin{array}{l} h(d) \leftarrow \\ \qquad \{p(d, d_j), t_i(d_j) | 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m \text{ and } v_i \not\in C_j\} \cup \\ \qquad \{p(d, d_j), f_i(d_j) | 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m \text{ and } \neg v_i \not\in C_j\} \end{array}$$

In this reduction, the predicates t_i (or f_i) denote the truth-values of the variables v_i.

[5] The depth of a term in a clause is defined as the minimum length of its linking chains. A term in a clause is linked with a linking chain of length 0 if it occurs in the head of the clause, and with a linking chain of length $d + 1$ if another term in the same literal is linked with a linking chain of length d in the clause [Kietz, 1993].

For instance, for $V = \{v_1, v_2\}$ and $C = \{\{v_1\}, \{v_2\}\}$, the following positive examples are constructed:

$$h(c_1) \leftarrow p(c_1, c_{11}), t_2(c_{11}), f_1(c_{11}), f_2(c_{11}), p(c_1, c_{12}), t_1(c_{12}), t_2(c_{12}), f_2(c_{12})$$

$$h(c_2) \leftarrow p(c_2, c_{21}), t_1(c_{21}), f_1(c_{21}), f_2(c_{21}), p(c_2, c_{22}), t_1(c_{22}), t_2(c_{22}), f_1(c_{12})$$

as well as the negative

$$h(d) \leftarrow p(d, d_1), t_2(d_1), f_1(d_1), f_2(d_1), p(d, d_2), t_1(d_2), f_1(d_2), f_2(d_2)$$

A solution for this learning task is now given by

$$h(X) \leftarrow p(X, X_1), t_1(X_1), t_2(X_1)$$

which corresponds to the truth assignment $v_1 = true = v_2$.

Exercise 10.11. Show how to compute a solution to the learning task of the previous example by computing the *lgg* of the positive examples. Show that it does not subsume the negative. Show also that for $C = \{\{v_1\}, \{\neg v_1\}\}$, the resulting learning problem does not have a solution.

Exercise 10.12. ** Show that the reduction sketched in the previous example is indeed a polynomial reduction. (The solution to this exercise and the previous one can be found in [Kietz and Džeroski, 1994].)

The above example shows that a particular inductive logic programming problem is not PAC-learnable. The weakness of this result (and of that of any PAC-learning result based on the hardness of the consistency problem) is that it only provides limited insight into the learning task. Indeed, even though it was shown that learning a single clause in $\mathcal{L}_{12}^{nd}$ is not tractable, it might well be the case that other, more expressive languages would be PAC-learnable. As such, the above result does not exclude the possibility that learning sets of clauses in $\mathcal{L}_{12}^{nd}$ is PAC-learnable.

A second proof technique for obtaining negative PAC-learning results relies on the seminal work by Schapire [1990]. This result essentially states that a language cannot be PAC-learnable (under certain complexity theory assumptions) if there exists a concept in the language for which the covers test cannot be evaluated in polynomial time. This is sometimes referred to as *evaluation hardness*. More formally, it is required that there exist a hypothesis $h \in \mathcal{L}$ such that testing whether h covers an example e cannot be performed in time polynomial in e and h. The result by Schapire is quite intuitive. Indeed, learning systems typically need to (repeatedly) test whether a candidate hypothesis covers particular examples, and therefore the learning task is expected to be harder than testing coverage. Even though the result by Schapire is intuitively clear, its proof is quite involved. Nevertheless, this result together with the complexity of typical coverage tests (such as θ-subsumption) indicates that there is little hope of obtaining positive PAC-learning results for large classes of inductive logic programming problems. The proof technique of evaluation hardness is illustrated in the following example, which can be safely skipped by the casual reader less interested in formal details. It concerns the language $\mathcal{L}_{13}^{nd}$, the language containing clauses of maximum depth 1, and predicates in the body of clauses of maximum arity 3.

Example 10.13. ** (From [Cohen, 1995]; cf. also [Cohen and Page, 1995]). To show evaluation hardness for $\mathcal{L}_{13}^{nd}$, a reduction from 3-SAT is used. More precisely, it is shown that a 3-CNF formula $C = \{C_1, \cdots, C_n\}$ with the $C_i = \{l_{i1}, l_{i2}, l_{i3}\}$ over the variables $\{x_1, \cdots, x_n\}$ can be encoded as an example e and that there exists a hypothesis $h \in \mathcal{L}_{13}^{nd}$ that covers e exactly when C is satisfiable. The example e is obtained from C as (where $m_{ij} = k$ if $l_{ij} = x_k$, $m_{ij} = -k$ if $l_{ij} = \neg x_k$):

$$
\begin{aligned}
& satisfiable(m_{11}, m_{12}, m_{13}, \cdots, m_{n1}, m_{n2}, m_{n3}) \leftarrow \\
& \quad sat(1,0,0), sat(0,1,0), sat(0,0,1), \\
& \quad sat(1,1,0), sat(0,1,1), sat(1,0,1), \\
& \quad sat(1,1,1), bool(0), bool(1), \\
& \quad \cup\{link_k(M,V,X) \mid \text{with } M \in \{-n, \cdots, -1, 1, \cdots, n\}, \\
& \qquad V \in \{0,1\}, X \in \{0,1\} \text{ and} \\
& \qquad (M = k \wedge X = V) \text{ or } (M = -k \wedge X = \neg V) \text{ or } (M \neq k \wedge M \neq -k)\}
\end{aligned}
$$

Additionally, the clause against which to test coverage (using θ-subsumption) is specified as:

$$
\begin{aligned}
& satisfiable(M_{11}, M_{12}, M_{13}, M_{21}, M_{22}, M_{23}, \cdots, M_{n1}, M_{n2}, M_{n3}) \leftarrow \\
& \qquad \wedge_{k=1}^{n} boolean(X_k) \\
& \qquad \wedge_{i=1}^{n} \wedge_{j=1}^{3} boolean(V_{ij}) \\
& \qquad \wedge_{i=1}^{n} \wedge_{j=1}^{3} \wedge_{k=1}^{n} link_k(M_{ij}, V_{ij}, X_k) \\
& \qquad \wedge_{i=1}^{n} sat(V_{i1}, V_{i2}, V_{i3})
\end{aligned}
$$

The idea is that the X_i and V_{ij} non-deterministically represent boolean variables. The X_i represent the values for x_i and the V_{ij} those of the corresponding literals l_{ij}. Furthermore, the literals for $link_k$ ensure that the V_{ij} have values that are consistent with the values of X_i; more specifically, if $l_{ij} = x_k$ then $V_{ij} = X_k$, and when $l_{ij} = \neg x_k$, V_{ij} and X_k should have complementary values. The *sat* predicate then determines when a clause evaluates to true (that is, when one of its literals is true).

Exercise 10.14. * Show how the 3-CNF

$$\{\{\neg x_1, \neg x_2, \neg x_3\}, \{\neg x_1, \neg x_2, x_3\}, \{\neg x_1, x_2, \neg x_3\}\}$$

is represented by this reduction.

This result seems, at first sight, similar to the one concerning the consistency problem. However, evaluation hardness provides a much more useful result. The reason is that any superset of the language $\mathcal{L}_{13}^{nd}$ will necessarily also contain a concept that is hard to evaluate, and therefore any superset of this language cannot be PAC-learnable either.

A third proof technique for obtaining PAC-learning results, called *prediction-preserving reducibility*, directly reduces one PAC-learning problem to another. The idea is very similar to that of other reductions employed within theoretical computer science, such as reductions from one NP-complete problem (for instance, SAT) to another (for instance, the consistency problem for $\mathcal{L}^{nd}_{12}$). This proof technique employs a slight variant of PAC-learnability, the so-called *PAC-predictability.* The key difference between PAC-predictability and PAC-learnability of a language $\mathcal{L}$ is that in the case of PAC-predictability it is not required that the output hypothesis belong to the language $\mathcal{L}$ of hypotheses. Instead, it is required that the output hypothesis evaluate in polynomial time (in the size of its input).

Consider the two hypotheses languages $\mathcal{L}_{h1}$ over $\mathcal{L}_{e1}$ and $\mathcal{L}_{h2}$ over $\mathcal{L}_{e2}$, and suppose that there exist two functions $f_e : \mathcal{L}_{e1} \rightarrow \mathcal{L}_{e2}$ and $f_h : \mathcal{L}_{h1} \rightarrow \mathcal{L}_{h2}$ with the following properties:

$$\begin{array}{l} e \in H \text{ if and only if } f_e(e) \in f_h(H) \\ size(f_h(H)) \text{ is polynomial in } size(H) \\ f_e(e) \text{ can be computed in polynomial time} \end{array} \tag{10.3}$$

Then we say that predicting $\mathcal{L}_{h1}$ reduces to predicting $\mathcal{L}_{h2}$, notation $\mathcal{L}_{h1} \trianglelefteq \mathcal{L}_{h2}$ [Pitt and Warmuth, 1990]. The first condition states that membership be preserved, the second that the size of hypotheses be preserved within a polynomial factor, and the last that the instance mapping f_e run in polynomial time.

If $\mathcal{L}_{h1} \trianglelefteq \mathcal{L}_{h2}$, one can use a learning algorithm for $\mathcal{L}_{h2}$ to learn concepts in $\mathcal{L}_{h1}$ by first mapping the data set E_1 (represented in $\mathcal{L}_{e1}$) to the data set $E_2 = \{f_e(e) | e \in E_1\}$ (represented in $\mathcal{L}_{e2}$), and then employing the learning algorithm for $\mathcal{L}_{h2}$ to learn a hypothesis H_2. The hypothesis H_2 can then be used for predicting the class of (unseen) examples $e \in \mathcal{L}_{e1}$ by testing whether $f_e(e)$ is covered by H_2. Pitt and Warmuth [1990] have shown that if $\mathcal{L}_{h1} \trianglelefteq \mathcal{L}_{h2}$ and $\mathcal{L}_{h2}$ is PAC-predictable, then $\mathcal{L}_{h1}$ is PAC-predictable. Prediction-preserving reducibilities can now be used in two directions. First, to show that $\mathcal{L}_{h1}$ is PAC-predictable, it suffices to find a PAC-predictable language $\mathcal{L}_{h2}$ and a prediction-preserving reduction such that $\mathcal{L}_{h1} \trianglelefteq \mathcal{L}_{h2}$. By taking the contra-position of the theorem by Pitt and Warmuth [1990], one can also show that a language $\mathcal{L}_{h1}$ is not PAC-predictable (and therefore not PAC-learnable) if it reduces to a language $\mathcal{L}_{h2}$ that is not PAC-predictable. PAC-reducibility is, hence, a powerful tool to obtain an understanding of the relative difficulty of learning problems.

Exercise 10.15. * Prove the positive result for jk-clausal theories and learning from interpretations using a prediction-preserving reduction to the set of monomials. Assume that monomials (conjunctions of boolean attributes) are PAC-learnable.

Exercise 10.16. ** Specify a prediction-preserving reduction from r-term DNF to the language $\mathcal{L}^{f}_{1}$. R-term DNF formulae are boolean formulae of the form $T_1 \vee \cdots \vee T_r$, where each of the T_i is a conjunction of literals over n boolean variables. Furthermore, $\mathcal{L}_1^f$ consists of single-clause hypotheses that contain at most one free or existential variable, that is, at most one variable appears in the body but not in the head of the clause. (See Cohen [1995], Cohen and Page [1995] for a solution.)

10.3 Conclusions

Throughout this chapter, we have focused on computational aspects of logical and relational learning. First, we addressed implementation issues, where we emphasized the need for efficient coverage testing. This lead us to investigate the efficiency of θ-subsumption testing in the phase-transition framework, and discuss various possible optimizations, such as sampling, use of interpretations, and query-packs. Second, we investigated logical and relational learning from a theoretical computer science perspective, that is, we discussed the convergence and complexity of logical and relational learning systems. Various frameworks for learnability were considered, such as identification in the limit and probably approximately correct learning. Although most results are negative, we also presented some positive results for simple settings.

10.4 Historical and Bibliographic Notes

An excellent survey on scaling and efficiency issues in relational learning is given by Blockeel and Sebag [2003]. The first section of this chapter is actually partly based on this work. The prominent role of θ-subsumption in inductive logic programming has motivated various researchers to look at algorithms for testing θ-subsumption (cf. [Kietz and Lübbe, 1994, Scheffer et al., 1997, Maloberti and Sebag, 2004]), and subsequently the effect for learning was analyzed empirically within the phase-transition framework by Giordana and Saitta [2000], Botta et al. [2003]. Various optimizations within Prolog were developed by Blockeel et al. [2002], Costa et al. [2003b], and, recently, some work has started looking at optimizations from a database perspective [Yin et al., 2004]. Optimizations for learning from interpretations were suggested by Blockeel et al. [1999].

There is also very rich literature on various frameworks for learning theory. Surveys of results in inductive inference, including identification in the limit and exact identification, can be found in [Angluin and Smith, 1983, Biermann, 1986], and of the PAC-learning framework in [Kearns and Vazirani, 1994, Natarajan, 1991]. More recently, there has been quite some attention given to statistical learning theory, lying at the basis of the developments around support vector machines and kernel methods; see Chapter 9 for more details. The dissertation of Shapiro [1983] contains a seminal result on the identification of Prolog programs using queries. A shorter version can be found in [Shapiro, 1991]. The famous Horn algorithm and its variants for exact identification of Horn theories from examples are described in [Angluin et al., 1992, Frazier and Pitt, 1993]. The role of queries for learning is investigated in [Angluin, 1987, 2004].

The first results concerning PAC-learning of logical and relational learning are due to Haussler [1989] and, in an inductive logic programming context, to Džeroski et al. [1992]. Early negative results are also due to Kietz [1993]; see also [Kietz and Džeroski, 1994]. An excellent introduction to computational learning theory for logical and relational learning is given by Cohen and Page [1995]. Various other interesting results can be found in [Cohen, 1995, De Raedt and Džeroski, 1994, Reddy and Tadepalli, 1997, 1998, Arias and Khardon, 2000, Khardon, 1999]. Worth mentioning is also the work on the polynomial learnability of elementary formal systems by Miyano et al. [2000]. Elementary formal systems employ a definite-clause-like syntax to specify formal languages and manipulate strings. Further results can be found in the proceedings of the annual conferences on Computational Learning Theory and Algorithmic Learning Theory.

11

Lessons Learned

Whereas the previous chapters introduced different aspects of logical and relational learning, the present chapter constitutes an attempt to summarize some of the main lessons learned that may be important for future developments in machine learning and artificial intelligence research. Taken together these lessons put logical and relational learning in a new perspective.

11.1 A Hierarchy of Representations

There exists a hierarchy of representational languages that can be used for logical and relational learning. The principles of logical and relational learning apply to all of these languages.

A multitude of different languages exist that can be used to represent structured data. As argued extensively in Chapter 4, they form a natural hierarchy. The hierarchy includes boolean, attribute-value, multi-instance, relational and logic programming representations as well as representations for sequences, trees and graphs. This means that, for instance, attribute-value representations are a special case of multi-instance learning representations, which in turn specialize relational representations. This observation, although straightforward, is important because learning problems and techniques developed for more expressive representations directly apply to the less expressive representations. As one illustration, relational learning systems can, in principle, be applied to multi-instance learning problems. The observation also holds for the theoretical results that exist about logical and relational learning, such as those concerning the generality relation; see below.

The multitude of different representations also points to the difficulty and desirability of selecting the right level of representation for the specific problem at hand. Selecting a too general representation increases the computational complexity, while selecting a too specific one may affect the quality of the learned knowledge. Therefore, it is important to get the representation right even if this means that a new representation or system has to be designed. This can be realized, when needed, by upgrading or downgrading.

11.2 From Upgrading to Downgrading

Logical and relational learning techniques can be obtained by upgrading techniques for less expressive representations; conversely, techniques for less expressive representations can be obtained by downgrading logical and relational learning techniques.

Most of the work so far in logical and relational learning has *upgraded* various propositional learning tasks, representations and systems to use logical and relational representations; see also Chapter 6. The upgrading approach has been successful and has resulted in many interesting new systems and representations, such as the rule-learner Foil [Quinlan, 1990], the logical decision tree inducer Tilde [Blockeel and De Raedt, 1998], the frequent query miner Warmr [Dehaspe and Toivonen, 2001] and even the relational instance-based learner Ribl [Emde and Wettschereck, 1996] and the Probabilistic Relational Models [Getoor et al., 2001a]. The upgraded frameworks are typically very expressive and allow one to emulate the original setting and system. For instance, Bayesian nets are an instance of probabilistic relational models and Quinlan's [1993a] well-known decision tree learner C4.5 is a special case of Tilde. At the same time, because of their expressiveness, they can work at different levels of representation, also intermediate ones. For instance, graphs and networks can easily be represented using relational logic, and hence logical and relational learning systems are applicable to graph- and network-based representations.

Expressiveness comes at a computational cost, which explains why logical and relational learning are usually less efficient than, and do not scale as well as more specialized machine learning techniques. Nevertheless, even in such cases, logical and relational learning can give interesting results. First, logical and relational learning systems can and should be used as a starting point and as a baseline for evaluating more specific techniques. Second, logical and relational learning systems can be *downgraded*, that is, specialized, tailored and optimized for use with less expressive representations; there are many opportunities for doing this. One productive line of research, which, arguably, downgrades logical and relational learning, is that on mining and learning in graphs.

11.3 Propositionalization and Aggregation

Propositionalization and aggregation can be used to transform logical and relational learning problems into less expressive representations.

Logical and relational learning has also studied how the richer representations can be transformed into simpler representations. In this regard, logical and relational learning has contributed several *propositionalization* techniques (cf. Sect. 4.12) that transform structured machine learning and data mining problems into a simpler format, typically a feature-vector or an attribute-value representation, though also more complex intermediate representations (such as multi-instance representations) are possible. The resulting problems can directly be input into the (more) standard machine learning and data mining algorithms that employ flat representations such as support-vector and kernel methods, decision tree learners, etc. Two types of techniques can be distinguished: *static* propositionalization, which first maps the logical or relational learning problem onto the simpler format, and then invokes learners on the simpler representations, and *dynamic* approaches, which incrementally construct a set of good features by coupling the propositionalization step with the learning step.

Aggregation (cf. Sect. 4.13) can play an important role in propositionalization as it summarizes the information about multiple values into a single value.

Propositionalization and aggregation have to be exercised with care because there is a risk of losing information, though propositionalization and aggregation techniques have proven to be effective for many classification problems.

11.4 Learning Tasks

Logical and relational learning, in principle, apply to any machine learning and data mining task.

Whereas initial work on logical and relational learning has focused on the task of learning a set of rules from positive and negative examples, it is clear that these representations are generally applicable, and can be used for solving other tasks. Today, logical and relational learning techniques exist that induce decision trees [Blockeel and De Raedt, 1998], realize clustering and instance-based learning [Kirsten et al., 2001], discover association rules in relational databases [Dehaspe and Toivonen, 2001], revise theories [Wrobel, 1996], employ kernels and support vector machines [Passerini et al., 2006], define graphical models [Getoor and Taskar, 2007, De Raedt et al., 2008], and learn from reinforcement [Džeroski et al., 2001].

11.5 Operators and Generality

Logical and relational learning has contributed a rich variety of frameworks and operators for generality that apply also to other representations.

The theory of logical and relational learning has contributed a rich variety of frameworks for reasoning about the generality of hypotheses, cf. Chapter 5. When using hypotheses in the form of logical formulae, the generality relation coincides with that of logical entailment. Typically, a hypothesis G is said to be more general than a hypothesis S if G entails S, that is, if $G \models S$. Applying a deductive inference operator leads to specializations, and applying inverted deductive operators, that is, inductive operators, leads to generalizations. A multitude of operators for generalization and specialization have been devised and are theoretically well-understood. Different operators exist that depend on the form of the hypotheses (single or multiple clause), the presence or absence of a background theory, the type of inference rule (deductive or inductive), and the search strategy applied (heuristic or complete search).

Many of the frameworks for generality can also be downgraded for use with more specialized representations, such as for instance graphs. The two most important frameworks for deciding whether one clause is more general than another one are θ-subsumption [Plotkin, 1970] and OI-subsumption [Esposito et al., 1996]. Specialized to graphs, these definitions correspond to the well-known notions of subgraph-isomorphism and -homeomorphism. As a consequence, it is easy (not to say straightforward) to adapt many of the results and operators of the subsumption frameworks to those of the graph-morphism ones. This can be used to obtain methods and algorithms for enumerating graphs with different properties. At the same time, some variants of the subsumption framework that take into account background knowledge in the form of sets of clauses or rules might be adapted towards the graph mining setting, potentially leading to a new class of graph mining systems.

11.6 Unification and Variables

Logical formulae can act as templates, realize parameter tying, and provide memory for use in various hybrid representations.

A key difference between propositional logic and relational and first-order logic lies in their use of variables and unification. Unification is a very powerful tool for machine learning and data mining in at least two respects. First, logical expressions that contain variables can be used as *templates* that make abstraction of specific instances. Knowledge-based model construction, as discussed in Chapter 8, is essentially the application of this idea to generate graphical models. Consider, for instance, Markov logic [Richardson and Domingos, 2006], in which a set S of weighted logical formulae of the form $w : f$ is used to construct a Markov network. This is realized by generating from each ground instance $f\theta$ of a formula $w : f$ some local fragment of the Markov network. On the one hand, the templates provide a general and compact description that allow us to deal with multiple extensions, and on the other hand it also allows for *parameter tying*, which facilitates learning. Second, variables and unification can also be used to propagate information. This is not only useful when performing deduction in logic, but also in the above sketched knowledge-based model construction approach. For instance, in a logical Markov model context, abstract state transitions such as $p(X) \rightarrow q(X)$, where $p(X)$ is an abstract state, can be instantiated to grounded states, that is, $p(c1)$. The abstract state transition can then be instantiated to $p(c1) \rightarrow q(c1)$ denoting that a transition to state $q(c1)$ occurs (perhaps with a particular probability). The value $c1$ is propagated from one state to the next and realizes a kind of memory in the Markov model. This mechanism by itself is important as it adds expressive power as shown, for instance, in Logical HMMs [Kersting et al., 2006]. Even though logical Markov models and Markov logic use elements of logic, they are not a purely logical representation because they merge graphical models and logic. As a consequence, logic is no longer used as a *target* language but rather as a *means* for realizing interesting intermediate representations.

11.7 Three Learning Settings

The nature of the examples can be taken into account in logical and relational learning by selecting the right learning setting.

The distinction between the model-theoretic and proof-theoretic perspective in logic has been used to define three settings for logical and relational learning that are applicable to different types of data; cf. Chapter 4. In the first setting, *learning from interpretations*, an example is a logical interpretation I, that is, a state description or possible world, and an example is covered by a hypothesis H (that is, a logical formula) if I is a model for H. In the second setting, *learning from entailment*, an example corresponds to an observation about the truth or falsity of a formula F. A hypothesis H then covers the formula F if F is entailed by the hypothesis, that is, $H \models F$. In the final setting, *learning from proofs*, an example is a proof (or a trace) and an example P is covered by a hypothesis H if P is a possible proof in the hypothesis H. Interpretations are the natural type of examples used in, for instance, Bayesian networks and item-set mining; observations in the form of true and false formulae are typical of scientific knowledge discovery problems, and proofs and traces are very natural when learning tree-bank grammars and Markov models. The settings provide different types of clues about the underlying target theory, and can be ordered according to their difficulty. Proofs carry the most information, as they directly encode (instantiated) rules of the unknown target theory; interpretations provide full information about a specific example; whereas formulae summarize or aggregate information about multiple states (or interpretations). Therefore, learning from proofs is easier than learning from interpretations, which in turn is easier than learning from entailment; cf. [De Raedt, 1997].

11.8 Knowledge and Background Knowledge

Using logical and relational learning new knowledge can be generated, and, in principle, any type of background knowledge can be specified and used in the learning process.

Throughout its history logical and relational learning has always stressed the importance of knowledge. Logical and relational learning techniques can generate new knowledge that is understandable and that provides new insights into the applcation domain. Several examples, such as those concerning structure activity relationship prediction, were discussed throughout this book. At the same time, logical and relational learning techniques can also incorporate background knowledge in the learning process. It enables this by exploiting the underlying representation formalism, which typically supports the definition of *intentional* or *view* predicates or relations, which provide additional information about the domain of discourse. These predicates can be used as any other predicate or relation in the learning process. This is a simple but powerful mechanism because, when using logic programs, essentially any "programmable" form of background theory can be specified. In this way Craven and Slattery [2001] have even encoded learning algorithms as background predicates. Furthermore, because hypotheses are encoded in the same language as the background theory, already learned hypotheses can be added to the background theory realizing closed-loop learning.

11.9 Applications

Logical and relational learning is applicable to a wide variety of problem domains.

Even though the book has stressed machine learning and data mining principles rather than applications, the reader should be aware that logical and relational learning are applicable to a wide variety of problem areas. Indeed, some well-known applications of logical and relational learning include natural language learning [Cussens and Džeroski, 2000], in bio- and chemo-informatics [King et al., 2004, Page and Craven, 2003], drug design [King et al., 1992], qualitative reasoning [Bratko et al., 1992], music analysis [Tobudic and Widmer, 2005], activity recognition [Liao et al., 2005], robotics [Kersting et al., 2007, Limketkai et al., 2005], intelligent assistants [Myers et al., 2007], ecological modeling [Džeroski et al., 1994], text and web mining [Craven and Slattery, 2001], user modeling [Jacobs and Blockeel, 2001], game playing [Ramon et al., 2001], validation and verification [Cohen, 1994b, De Raedt et al., 1991]. Overviews of some of these applications can be found in [Bratko and Muggleton, 1995, Page and Craven, 2003, Bratko and Džeroski, 1995, Džeroski and Bratko, 1996, Džeroski, 2001, Cussens and Džeroski, 2000].

References

A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

H. Adé and M. Denecker. Abductive inductive logic programming. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, 1995.

H. Adé, L. De Raedt, and M. Bruynooghe. Declarative Bias for Specific-to-General ILP Systems. *Machine Learning*, 20(1/2):119 – 154, 1995.

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

D. Aha, D. Kibler, and M. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

J. Allen. *Natural Language Understanding.* Benjamin/Cummings Publishing Company, 1987.

S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

C. R. Anderson, P. Domingos, and D. S. Weld. Relational Markov models and their application to adaptive web navigation. In D. Hand, D. Keim, O. R. Zaïne, and R. Goebel, editors, *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, pages 143–152. ACM Press, 2002.

D. Angluin. Queries revisited. *Theoretical Computer Science*, 313(2):175–194, 2004.

D. Angluin. Queries and concept-learning. *Machine Learning*, 2:319–342, 1987.

D. Angluin and C. H. Smith. A survey of inductive inference: theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of Horn clauses. *Machine Learning*, 9:147–162, 1992.

M. Arias and R. Khardon. Learning closed Horn expressions. *Information and Computation*, 178(1):214–240, 2002.

M. Arias and R. Khardon. A new algorithm for learning range restricted Horn expressions (extended abstract). In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 21–39. Springer, 2000.

F. Bacchus. Lp, a logic for representing and reasoning with statistical knowledge. *Computational Intelligence*, 6:209–231, 1990.

L. Badea. A refinement operator for theories. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 1–14. Springer, 2001.

P. Baldi and S. Brunak. *Bioinformatics: the Machine Learning Approach.* The MIT Press, 1998.

P. Baldi, P. Frasconi, and P. Smyth. *Modeling the Internet and the Web: Probabilistic Methods and Algorithms.* John Wiley, 2003.

F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. *SIGMOD Record*, 15:16–51, 1986.

R. B. Banerji. A language for the description of the concepts. *General Systems*, 5:117–127, 1964.

Y. Bengio and P. Frasconi. An input output HMM architecture. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 427–434. MIT Press, 1994.

F. Bergadano and A. Giordana. Guiding induction with domain theories. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 3, pages 474–492. Morgan Kaufmann, 1990.

F. Bergadano and D. Gunetti, editors. *Inductive Logic Programming: from Machine Learning to Software Engineering.* MIT Press, 1995.

F. Bergadano, A. Giordana, and L Saitta. Concept acquisition in noisy environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10:555–578, 1988.

I. Bhattacharya, L. Getoor, and L. Licamele. Query-time entity resolution. In T. Eliassi-Rad, L. H. Ungar, M. Craven, and D. Gunopulos, editors, *KDD*, pages 529–534. ACM, 2006. ISBN 1-59593-339-5.

A. Biermann. Fundamental mechanisms in machine learning and inductive inference. In W. Bibel and P. Jorrand, editors, *Fundamentals of Artificial Intelligence*. Springer, 1986.

A. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C(21):592–597, 1972.

A. Biermann, G. Guiho, and Y. Kodratoff, editors. *Automatic Program Construction Techniques.* Macmillan, 1984.

C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

G. Bisson. Learning in FOL with a similarity measure. In *Proceedings of the 10th National Conference on Artificial Intelligence.* AAAI Press, 1992a.

G. Bisson. Conceptual clustering in a first-order logic representation. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 458–462. John Wiley, 1992b.

H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–84. Springer, 1997.

H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

H. Blockeel and M. Sebag. Scalability and efficiency in multi-relational data mining. *SIGKDD Explorations*, 5(1):17–30, 2003.

H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1):59–93, 1999.

H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

M. Bongard. *Pattern Recognition.* Spartan Books, 1970.

H. Boström and P. Idestam-Almquist. Induction of logic programs by example-guided unfolding. *Journal of Logic Programming*, 40(2-3):159–183, 1999.

M. Botta, A. Giordana, L. Saitta, and M. Sebag. Relational learning as search in a critical region. *Journal of Machine Learning Research*, 4:431–463, 2003.

C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDP's. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 690–700. Morgan Kauffmann, 2001.

I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1990. 2nd Edition.

I. Bratko and S. Džeroski. Engineering applications of ILP. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):313–333, 1995.

I. Bratko and S. Muggleton. Applications of inductive logic programming. *Communications of the ACM*, 38(11):65–70, 1995.

I. Bratko, S. Muggleton, and A. Varšek. Learning qualitative models of dynamic systems. In S. Muggleton, editor, *Inductive Logic Programming*, pages 437–452. Academic Press, 1992.

J. S. Breese, R. P. Goldman, and M. P. Wellman. Introduction to the special section on knowledge-based construction of probabilistic and decision models. *Cybernetics*, 24(11):1577–1579, 1994.

L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees.* Chapman and Hall, 1984.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

B. G. Buchanan and T. M. Mitchell. Model-directed learning of production rules. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems.* Academic Press, 1978.

H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3):255 – 259, 1998.

H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In T. Caelli, A. Amin, R. P. W. Duin, M. S. Kamel, and D. de Ridder, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, volume 2396 of *Lecture Notes in Computer Science*, pages 123–132. Springer, 2002.

W. Buntine. Induction of Horn-clauses: Methods and the plausible generalization algorithm. *International Journal of Man-Machine Studies*, 26:499–520, 1987.

W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

W. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.

S. Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data.* Morgan Kaufmann, 2002.

E. Charniak. Tree-bank grammars. In *Proceedings of the 13th National Conference on Artificial Intelligence*, volume 2, pages 1031–1036. AAAI Press, 1996.

P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In Y. Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 151–163. Springer, 1991.

P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.

B. L. Cohen and C. Sammut. Object recognition and concept learning with CONFUCIUS. *Pattern Recognition Journal*, 15(4):309– 316, 1982.

W. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994a.

W. W. Cohen. Recovering Software Specifications with ILP. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 142–148, 1994b.

W. W. Cohen. PAC-learning non-recursive Prolog clauses. *Artificial Intelligence*, 79:1–38, 1995.

W. W. Cohen and D. Page. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Computing*, 13, 1995.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

V. Santos Costa, D. Page, M. Qazi, and J. Cussens. CLP(BN): Constraint logic programming for probabilistic knowledge. In C. Meek and U. Kjærulff, editors, *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*, pages 517–524. Morgan Kaufmann, 2003a.

V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003b.

R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems.* Springer, 1999.

M. Craven and S. Slattery. Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, 43(1/2):97–119, 2001.

N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines.* Cambridge University Press, 2000.

J. Cussens. Stochastic logic programs: Sampling, inference and applications. In C. Boutilier and M. Goldszmidt, editors, *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 115–122. Morgan Kaufmann, 2000.

J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.

J. Cussens and S. Džeroski, editors. *Learning Language in Logic*, volume 1925 of *Lecture Notes in Computer Science.* Springer, 2000.

E. Dantsin. Probabilistic logic programs and their semantics. In A. Voronkov, editor, *Proceeings 1st Russian Conference on Logic Programming*, volume 592 of *Lecture Notes in Computer Science*. Springer, 1992.

L. De Raedt. The inductive logic programming project. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 1996.

L. De Raedt. Attribute-value learning versus inductive logic programming: The missing links (extended abstract). In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 1998.

L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95: 187–201, 1997.

L. De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach.* Academic Press, 1992.

L. De Raedt and M. Bruynooghe. A unifying framework for concept-learning algorithms. *The Knowledge Engineering Review*, 7(3):251–269, 1992a.

L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992b.

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

L. De Raedt and M. Bruynooghe. Interactive theory revision. In R. S. Michalski and G. Tecuci, editors, *Machine Learning: A Multistrategy Approach*, volume 4. Morgan Kaufmann, 1994.

L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

L. De Raedt and K. Kersting. Probabilistic inductive logic programming. In *Proceedings of the 15th International Conference on Algorithmic Learning Theory*, number 3244 in Lecture Notes in Computer Science, pages 19–36. Springer, 2004.

L. De Raedt and K. Kersting. Probabilistic logic learning. *SIGKDD Explorations*, 5(1):31–48, 2003.

L. De Raedt and S. Kramer. The level-wise version space algorithm and its application to molecular fragment finding. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 853–862. Morgan Kaufmann, 2001.

L. De Raedt and J. Ramon. On the generality relation and distance metrics. submitted, 2008.

L. De Raedt and J. Ramon. Condensed representations for inductive logic programming. In D. Dubois, A. Welty C., and M.-A. Williams, editors, *Proceedings of the 9th International Conference on Principles and Practice of Knowledge Representation*, AAAI Press, pages 438–446, 2004.

L. De Raedt, G. Sablon, and M. Bruynooghe. Using interactive concept-learning for knowledge base validation and verification. In M. Ayel and J.P. Laurent, editors, *Validation, Verification and Testing of Knowledge Based Systems*, pages 177–190. John Wiley, 1991.

L. De Raedt, P. Idestam-Almquist, and G. Sablon. Theta-subsumption for structural matching. In *Proceedings of the 9th European Conference on Machine Learning*, volume 1224 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 1997.

L. De Raedt, H. Blockeel, L. Dehaspe, and W. Van Laer. Three companions for data mining in first order logic. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 105–139. Springer, 2001.

L. De Raedt, T. G. Dietterich, L. Getoor, and S. Muggleton, editors. *Probabilistic, Logical and Relational Learning - Towards a Synthesis*, volume 05051 of *Dagstuhl Seminar Proceedings*, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

L. De Raedt, K. Kersting, and S. Torge. Towards learning stochastic logic programs from proof-banks. In *Proceedings of the 20th National Conference* *on Artificial Intelligence of Artificial Intelligence Conference*, pages 752–757, 2005.

L. De Raedt, T. G. Dietterich, L. Getoor, K. Kersting, and S. Muggleton, editors. *Probabilistic, Logical and Relational Learning - Towards a Further Synthesis*, volume 07161 of *Dagstuhl Seminar Proceedings*, 2007a. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic Prolog and its application in link discovery. In M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2462–2467, 2007b.

L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors. *Probabilistic Inductive Logic Programming — Theory and Applications*, volume 4911 of *Lecture Notes in Artificial Intelligence*. Springer, 2008.

R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 125–132. Springer, 1997.

L. Dehaspe and H. Toivonen. Discovery of relational association rules. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 189–212. Springer, 2001.

L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Proceedings of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press, 1998.

M. Denecker and D. De Schreye. SLDNFA: An abductive procedure for normal abductive logic programs. In K. R. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 686–700. MIT Press, 1992.

T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89 (1-2):31–71, 1997.

K. Driessens, J. Ramon, and H. Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree algorithm. In L. De Raedt and P. Flach, editors, *Proceedings of the 12th European Conference on Machine Learning*, volume 2167 of *Lecture Notes in Artificial Intelligence*, pages 97–108. Springer, 2001.

K. Driessens, R. Givan, and P. Tadepalli, editors. *Proceedings of the ICML Workshop on Relational Reinforcement Learning*, 2004.

K. Driessens, A. Fern, and M. van Otterlo, editors. *Proceedings of the ICML Workshop on Rich Representations for Reinforcement Learning*, 2005.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

S. Džeroski. Relational data mining applications: An overview. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 339–364. Springer, 2001.

S. Džeroski and H. Blockeel, editors. *Proceedings of the 3rd SIGKDD Workshop on Multi-Relational Data Mining*, 2004.

S. Džeroski and H. Blockeel, editors. *Proceedings of the 4th SIGKDD Workshop on Multi-Relational Data Mining*, 2005.

S. Džeroski and I. Bratko. Applications of inductive logic programming. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 65–81. IOS Press, 1996.

S. Džeroski, S. Muggleton, and S. Russell. PAC-learnability of determinate logic programs. In *Proceedings of the 5th ACM Workshop on Computational Learning Theory*, pages 128–135, 1992.

S. Džeroski, L. Dehaspe, B. Ruck, and W. Walley. Classification of river water quality data using machine learning. In *Proceedings of the 5th International Conference on the Development and Application of Computer Techniques to Environmental Studies*, 1994.

S. Džeroski, S. Schulze-Kremer, K. R. Heidtke, K. Siems, D. Wettschereck, and H. Blockeel. Diterpene structure elucidation from 13C NMR spectra with inductive logic programming. *Applied Artificial Intelligence*, 12(5): 363–383, 1998.

S. Džeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43(1/2):5–52, 2001.

S. Džeroski, L. De Raedt, and S. Wrobel, editors. *Proceedings of the 1st SIGKDD Workshop on Multi-Relational Data Mining*, 2002.

S. Džeroski, L. De Raedt, and S. Wrobel, editors. *Proceedings of the 2nd SIGKDD Workshop on Multi-Relational Data Mining*, 2003.

S. Džeroski and N. Lavrač, editors. *Relational Data Mining.* Springer, 2001.

A. Eisele. Towards probabilistic extensions of contraint-based grammars. In J. Dörne, editor, *Computational Aspects of Constraint-Based Linguistics Decription-II.* DYNA-2 deliverable R1.2.B, 1994.

R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, 2nd edition, 1989.

W. Emde and D. Wettschereck. Relational instance-based learning. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning*, pages 122–130. Morgan Kaufmann, 1996.

W. Emde, C. U. Habel, and C. R. Rollinger. The discovery of the equator or concept driven learning. In A. Bundy, editor, *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 455–458. Morgan Kaufmann, 1983.

F. Esposito, A. Laterza, D. Malerba, and G. Semeraro. Refinement of Datalog programs. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programing*, pages 73–94, 1996.

A. Fern, R. Givan, and J. M. Siskind. Specific-to-general learning for temporal events with application to learning event definitions from video. *Journal of Artificial Intelligence Research*, 17:379–449, 2002.

A. Fern, S. W. Yoon, and R. Givan. Reinforcement learning in relational domains: A policy-language approach. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning.* MIT Press, 2007.

R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

P. Flach. Predicate invention in inductive data engineering. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence, pages 83–94. Springer, 1993.

P. Flach. *Simply Logical - Intelligent Reasoning by Example.* John Wiley, 1994.

P. Flach and A. C. Kakas, editors. *Abduction and Induction: Essays on Their Relation and Integration.* Kluwer Academic Press, 2000.

P. Flach and N. Lachiche. Confirmation-guided discovery of first-order rules with Tertius. *Machine Learning*, 42(1/2):61–95, 2001.

P. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

P. Flach, C. Giraud-Carrier, and J. W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 185–194. Springer, 1998.

P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: achievements and prospects. *Journal of Logic Programming*, 4(1):141–195, 1999.

P. Frasconi, K. Kersting, and K. Tsuda, editors. *Proceedings of the 5th International Workshop on Mining and Learning in Graphs*, 2007.

M. Frazier and L. Pitt. Learning from entailment. In *Proceedings of the 9th International Conference on Machine Learning.* Morgan Kaufmann, 1993.

N. Friedman. The Bayesian structural EM algorithm. In G. F. Cooper and S. Moral, editors, *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*, pages 129–138. Morgan Kaufmann, 1998.

N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In T. Dean, editor, *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence International Joint Conference on Artificial Intelligence*, pages 1300–1309, Stockholm, Sweden, 1999. Morgan Kaufmann.

J. Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.

H. Gallaire, J. Minker, and J. M. Nicolas. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16:153–185, 1984.

J. G. Ganascia and Y. Kodratoff. Improving the generalization step in learning. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 215–241. Morgan Kaufmann, 1986.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, California, 1979.

G. C. Garriga, R. Khardon, and L. De Raedt. On mining closed sets in multi-relational data. In M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 804–809, 2007.

T. Gärtner. A survey of kernels for structured data. *SIGKDD Explorations*, 5(1):49–58, 2003.

T. Gärtner, P. Flach, A. Kowalczyk, and A. Smola. Multi-instance kernels. In C. Sammut and A. G. Hoffmann, editors, *Proceedings of the 19th International Conference on Machine Learning*, pages 179–186. Morgan Kaufmann, 2002.

T. Gärtner, K. Driessens, and J. Ramon. Graph kernels and Gaussian processes for relational reinforcement learning. In T. Horváth and A. Yamamoto, editors, *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Artificial Intelligence*, pages 146–163. Springer, 2003.

T. Gärtner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In B. Schölkopf and M. K. Warmuth, editors, *Proceeedings 16th Annual Conference on Computational Learning Theory and 7th Kernel Workshop*, volume 2777 of *Lecture Notes in Computer Science*, pages 129–143. Springer Verlag, 2003.

T. Gärtner, J. W. Lloyd, and P. Flach. Kernels and distances for structured data. *Machine Learning*, 57(3):205–232, 2004.

P. Geibel and F. Wysotzki. A logical framework for graph theoretical decision tree learning. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 173–180. Springer, 1997.

M. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

L. Getoor. Link mining: A new data mining challenge. *SIGKDD Explorations*, 5(1):84 – 89, 2003.

L. Getoor. *Learning Statistical Models from Relational Data.* PhD thesis, Stanford University, 2001.

L. Getoor and C. P. Dielh, editors. Special Issue on Link Mining. *SIGKDD Explorations,* 7(2), 2005.

L. Getoor and D. Jensen, editors. *Working Notes of the IJCAI-2003 Workshop on Learning Statistical Models from Relational Data (SRL-00)*, 2003.

L. Getoor and D. Jensen, editors. *Working Notes of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data (SRL-03)*, 2000.

L. Getoor and B. Taskar, editors. *An Introduction to Statistical Relational Learning.* MIT Press, 2007.

L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 307–335. Springer, 2001a.

L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of relational structure. In C. E. Brodley and A. Danyluk, editors, *Proceedings of the 18th International Conference on Machine Learning*, pages 170–177. Morgan Kaufmann, 2001b.

L. Getoor, J. Rhee, D. Koller, and P. Small. Understanding tuberculosis epidemiology using probabilistic relational models. *Journal of Artificial Intelligence in Medicine*, 30:233–256, 2004.

D. A. Gillies. *Artificial Intelligence and Scientific Method.* Oxford University Press, 1996.

A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000.

P. Haddawy. Generating Bayesian networks from probabilistic logic knowledge bases. In R. López de Mántaras and D. Poole, editors, *Proceedings of the 10th Annual Conference on Uncertainty in Artificial Intelligence*, pages 262–269. Morgan Kaufmann, 1994.

C. Hartshorne and P. Weiss, editors. *Collected papers of Charles Sanders Peirce.* Harvard University Press, 1965.

D. Haussler. Learning conjunctive concepts in structural domains. *Machine Learning*, 4:7–40, 1989.

D. Haussler. Convolution kernels on discrete structures. Technical Report 99-10, UCSC-CRL, 1999.

F. Hayes-Roth and J. McDermott. An interference matching technique for inducing abstractions. *Communications of the ACM*, 21:401–410, 1978.

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.

C. Helma, editor. *Predictive Toxicology.* CRC Press, 2005.

H. Hirsh. *Incremental Version-Space Merging: A General Framework for Concept Learning.* Kluwer Academic Publishers, 1990.

C. J. Hogger. *Essentials of Logic Programming.* Oxford University Press, 1990.

T. Horvath, S. Wrobel, and U. Bohnebeck. Relational instance-based learning with lists and terms. *Machine Learning*, 43(1/2):53–80, 2001.

A. Hutchinson. Metrics on terms and clauses. In M. van Someren and G. Widmer, editors, *Proceedings of the 9th European Conference on Machine Learning*, volume 1224 of *Lecture Notes in Artificial Intelligence*, pages 138–145. Springer, 1997.

P. Idestam-Almquist. *Generalization of clauses.* PhD thesis, Stockholm University, Department of Computer and Systems Sciences, 1993.

A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50:321–354, 2003.

N. Jacobs and H. Blockeel. From shell logs to shell scripts. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 80–90. Springer, 2001.

M. Jaeger. Model-theoretic expressivity analysis. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming — Theory and Applications*, volume 4911 of *Lecture Notes in Artificial Intelligence*. Springer, 2008.

M. Jaeger. Parameter learning for relational Bayesian networks. In Z. Ghahramani, editor, *ICML*, volume 227, pages 369–376. ACM, 2007.

M. Jaeger. Relational Bayesian networks. In D. Geiger and P. P. Shenoy, editors, *Proceedings of the 13th Annual Conference on Uncertainty in Artificial Intelligence*, pages 266–273. Morgan Kaufmann, 1997.

D. Jensen and J. Neville. Linkage and autocorrelation cause feature selection bias in relational learning. In C. Sammut and A. G. Hoffmann, editors, *Proceedings of the 19th International Conference on Machine Learning*, pages 259–266. Morgan Kaufmann, 2002.

F. V. Jensen. *Bayesian Networks and Decision Graphs.* Springer, 2001.

W. S. Jevons. *The Principles of Science: a Treatise on Logic and Scientific Method.* Macmillan, 1874.

A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.

H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In T. Fawcett and N. Mishra, editors, *Proceedings of the 20th International Machine Learning Conference*, pages 321–328. AAAI Press, 2003.

M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

K. Kersting and L. De Raedt. Bayesian logic programming: theory and tool. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning.* MIT Press, 2007.

K. Kersting and L. De Raedt. Bayesian logic programs. Technical Report 151, University of Freiburg, Institute for Computer Science, April 2001.

K. Kersting and L. De Raedt. Logical Markov Decision Programs. In L. Getoor and D. Jensen, editors, *Working Notes of the IJCAI-2003 Workshop on Learning Statistical Models from Relational Data (SRL-03)*, 2003.

K. Kersting, M. van Otterlo, and L. De Raedt. Bellman goes relational. In *Proceedings of the 21st International Conference on Machine learning.* ACM, 2004.

K. Kersting, L. De Raedt, and T. Raiko. Logical hidden Markov models. *Journal of Artificial Intelligence Research*, 25:425–456, 2006.

K. Kersting, C. Plagemann, A. Cocora, W. Burgard, and L. De Raedt. Learning to transfer optimal navigation policies. *Advanced Robotics*, 21(13):1565–1582, 2007.

R. Khardon. Learning function free Horn expressions. *Machine Learning*, 37 (3):141–275, 1999.

R. Khardon. Learning Horn expressions with Logan-H. In P. Langley, editor, *Proceedings of the 17th International Conference on Machine Learning.* Morgan Kaufmann, 2000.

R. Khardon and D. Roth. Learning to reason. *Journal of the ACM*, 44(5): 697–725, 1997.

J.-U. Kietz. Some lower bounds for the computational complexity of inductive logic programming. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 115–124. Springer, 1993.

J.-U. Kietz and S. Džeroski. Inductive logic programming and learnability. *SIGART Bulletin*, 5(1):22–32, 1994.

J.-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 97–106. Gesellschaft für Mathematik und Datenverarbeitung, 1994.

R. D. King and A. Srinivasan. Prediction of rodent carcinogenicity bioassays from molecular structure using inductive logic programming. *Environmental Health Perspectives*, 104(5):1031–1040, 1996.

R. D. King, S. Muggleton, R. A. Lewis, and M. J .E. Sternberg. Drug design by machine learning: the use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, 89 (23):1322–11326, 1992.

R. D. King, M. J. E. Sternberg, and A. Srinivasan. Relating chemical activity to structure: an examination of ILP successes. *New Generation Computing*, 13, 1995.

R. D. King, K .E. Whelan, F. M. Jones, P. Reiser, C. H. Bryant, S. Muggleton, D.B. Kell, and S. Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427:247–252, 2004.

M. Kirsten and S. Wrobel. Extending k-means clustering to first-order representations. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 112–129. Springer, 2000.

M. Kirsten, S. Wrobel, and T. Horvath. Distance based approaches to relational learning and clustering. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 213–232. Springer, 2001.

A. J. Knobbe, M. de Haas, and A. Siebes. Propositionalisation and aggregates. In L. De Raedt and A. Siebes, editors, *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 277–288. Springer, 2001.

A. J. Knobbe, A. Siebes, and B. Marseille. Involving aggregate functions in multi-relational search. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 1–1. Springer, 2002.

S. Kok and P. Domingos. Learning the structure of Markov logic networks. In *Proceedings of the 22nd International Machine Learning Conference*, pages 441–448, 2005.

D. Koller. Probabilistic relational models. In S. Džeroski and P.A. Flach, editors, *Proceedings of 9th International Workshop on Inductive Logic Programming*, volume 1634 of *LNAI*, pages 3–13. Springer, 1999.

D. Koller and A. Pfeffer. Object-oriented Bayesian networks. In D. Geiger and P. P. Shenoy, editors, *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 302–313, Providence, Rhode Island, USA, 1997. Morgan Kaufmann.

R. A. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

S. Kramer. Structural regression trees. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 1996.

S. Kramer, N. Lavrač, and P. Flach. Propositionalization approaches to relational data mining. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 262–291. Springer, 2001.

M.-A. Krogel and S. Wrobel. Transformation-based learning using multirelational aggregation. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, pages 142–155. Springer, 2001.

M. Kuramochi and G. Karypis. Frequent subgraph discovery. In N. Cercone, T.Y. Lin, and X. Wu, editors, *Proceedings of the 1st IEEE International Conference on Data Mining*, pages 313–320, 2001.

J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In C. E. Brodley and A. P. Danyluk, editors, *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289. Morgan Kaufmann, 2001.

L. V. S. Lakshmanan, N. Leone, R. B. Ross, and V. S. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Transactions Database Systems*, 22(3):419–469, 1997.

N. Landwehr, A. Passerini, L. De Raedt, and P. Frasconi. kFoil: Learning simple relational kernels. In *Proceedings of the 21st National Conference on Artificial Intelligence.* AAAI Press, 2006.

N. Landwehr, K. Kersting, and L. De Raedt. Integrating naïve Bayes and Foil. *Journal of Machine Learning Research*, 8:481–507, 2007.

P. Langley. *Elements of Machine Learning.* Morgan Kaufmann, 1996.

P. Langley, H. Simon, G. Bradshaw, and J. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Processes.* MIT Press, 1987.

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

N. Lavrač, S. Džeroski, and M. Grobelnik. Learning non-recursive 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.* Springer, 1991.

S. D. Lee. *Constrained Mining of Patterns in Large Databases.* PhD thesis, Albert-Ludwigs-University, 2006.

S. D. Lee and L. De Raedt. Constraint based mining of first-order sequences in SeqLog. In R. Meo, P. L. Lanzi, and M. Klemettinen, editors, *Database Support for Data Mining Applications*, volume 2682 of *Lecture Notes in Computer Science*, pages 154–173. Springer, 2004.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 1966.

L. Liao, D. Fox, and H. A. Kautz. Location-based activity recognition using relational Markov networks. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 773–778, 2005.

B. Limketkai, L. Liao, and D. Fox. Relational object maps for mobile robots. In L. P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1471–1476, 2005.

J. W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data.* Springer, 2003.

J. W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd edition, 1987.

J. Maloberti and M. Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.

H. Mannila and K.-J. Raiha. *The Design of Relational Databases.* Addison-Wesley, 1992.

H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

C. H. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing.* The MIT Press, 1999.

R. Manthey and F. Bry. A hyperresolution-based proof procedure and its implementation in Prolog. In *11th German Workshop on Artificial Intelligence*, pages 221–230, 1987.

R. Manthey and F. Bry. Satchmo: a theorem prover implemented in Prolog. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 415–434. Springer, 1988.

J. Marcinkowski and L. Pacholski. Undecidability of the Horn-clause implication problem. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 354–362. IEEE Computer Society Press, 1992.

J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software – Practice and Experience*, 12(1):23–34, 1982.

S. Menchetti, F. Costa, and P. Frasconi. Weighted decomposition kernels. In L. De Raedt and S. Wrobel, editors, *Proceedings of the 22nd International Machine Learning Conference*, pages 585–592. ACM Press, 2005.

R. S. Michalski. A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 1. Morgan Kaufmann, 1983.

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

T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18(3):217–242, 2000.

R. J. Mooney. Learning for semantic interpretation: Scaling up without dumbing down. In J. Cussens and S. Džeroski, editors, *Learning Language in Logic*, volume 1925 of *Lecture Notes in Computer Science*, pages 57–66. Springer, 2000.

R. J. Mooney and M. E. Califf. Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, 3:1–24, 1995.

K. Morik, S. Wrobel, J.-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications.* Academic Press, 1993.

S. Morishita and J. Sese. Traversing itemset lattice with statistical metric pruning. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 226–236. ACM Press, 2000.

S. Muggleton. Inverting implication. In S. Muggleton, editor, *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, Report ICOT TM-1182, pages 19–39, 1992a.

S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3-4):245–286, 1995.

S. Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 287–292. Morgan Kaufmann, 1987.

S. Muggleton. Inductive logic programming. *New Generation Computing*, 8 (4):295–317, 1991.

S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, 1992b.

S. Muggleton. Learning structure and parameters of stochastic logic programs. In S. Matwin and C. Sammut, editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 198–206. Springer, 2003.

S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 1996.

S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–351. Morgan Kaufmann, 1988.

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

S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

S. Muggleton, R. D. King, and M. J .E. Sternberg. Protein secondary structure prediction using logic. *Protein Engineering*, 7:647–657, 1992.

S. Muggleton, H. Lodhi, A. Amini, and M. J. E. Sternberg. Support vector inductive logic programming. In A. G. Hoffmann, H. Motoda, and T. Scheffer, editors, *Proceedings of the 8th International Conference on Discovery Science*, volume 3735 of *Lecture Notes in Computer Science*, pages 163–175. Springer Verlag, 2005.

K. Myers, P. Berry, J. Blythe, K. Conley, M. Gervasio, D. McGuinness, D. Morley, A. Pfeffer, M. Pollack, and M. Tambe. An intelligent personal assistant for task and time management. *AI Magazine*, 28(2):47–61, 2007.

B. K. Natarajan. *Machine Learning: A Theoretical Approach.* Morgan Kaufmann, 1991.

C. Nédellec, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*, pages 82–103. IOS Press, 1996.

J. Neville, D. Jensen, L. Friedland, and M. Hay. Learning relational probability trees. In L. Getoor, T. E. Senator, P. Domingos, and C. Faloutsos, editors, *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 625–630, 2003.

L. Ngo and P. Haddawy. Probabilistic logic programming and Bayesian networks. In *Proceedings of the Asian Computing Science Conference 1995*, Pathumthai, Thailand, December 1995.

S.-H. Nienhuys-Cheng. Distances and limits on Herbrand interpretations. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 250–260. Springer, 1998.

S.-H. Nienhuys-Cheng. Distance between Herbrand interpretations: A measure for approximations to a target concept. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence.* Springer, 1997.

S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer, 1997.

S. Nijssen and J. Kok. Faster association rules for multiple relations. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 891–896. Morgan Kaufmann, 2001.

S. Nijssen and J. Kok. Efficient query discovery in Farmer. In N. Lavrač, D. Gamberger, H. Blockeel, and L. Todorovski, editors, *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 2838 of *Lecture Notes in Artificial Intelligence*, pages 350–362. Springer, 2003.

S. Nijssen and J. Kok. A quickstart in frequent structure mining can make a difference. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 647–652, 2004.

N. J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–87, 1986.

N. J. Nilsson. *The Mathematical Foundations of Learning Machines*. Morgan Kaufmann, 1990.

R. A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

D. Page and M. Craven. Biological applications of multi-relational data mining. *SIGKDD Explorations*, 5(1):69–79, 2003.

A. Passerini, P. Frasconi, and L. De Raedt. Kernels on Prolog proof trees: Statistical learning in the ILP setting. *Journal of Machine Learning Research*, 7:207–342, 2006.

M. J. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

C. Perlich and F. Provost. Aggregation-based feature invention and relational concept classes. In L. Getoor, T. E. Senator, P. Domingos, and C. Faloutsos, editors, *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 167–176. ACM Press, 2003.

A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

A. Pfeffer. *Probabilistic Reasoning for Complex Systems*. PhD thesis, Stanford University, 2000.

L. Pitt and M. Warmuth. Prediction-preserving reducibility. *Journal of Computer and System Science*, 41:430–467, 1990.

G. D. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.

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

D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47, 1988.

D. Poole. Logic programming, abduction and probability. *New Generation Computing*, 11:377–400, 1993a.

D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

D. Poole. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 985–991. Morgan Kaufman, 2003.

D. Poole. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64:81–129, 1993b.

A. Popescul and L. H. Ungar. Feature generation and selection in multi-relational statistical learning. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

D. Prescher. A tutorial on the expectation-maximization algorithm including maximum-likelihood estimation and EM training of probabilistic context-free grammars. Working notes of 15th European Summer School in Logic, Language and Information, 2003.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

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

J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993a.

J. R. Quinlan. Foil: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 3–20. Springer, 1993b.

J. R. Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, 1996.

L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi. Graph kernels for chemical informatics. *Neural Networks*, 18(8):1093–1110, 2005.

J. Ramon. *Clustering and Instance-Based Learning in First-Order Logic*. PhD thesis, Katholieke Universiteit Leuven, 2002.

J. Ramon and M. Bruynooghe. A polynomial time computable metric between point sets. *Acta Informatica*, 37(10):765–780, 2001.

J. Ramon, T. Francis, and H. Blockeel. Learning a Tsume-Go heuristic with Tilde. In T. Marsland and I. Frank, editors, *Proceedings of the 2nd International Conference on Computer and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 151–169. Springer, October 2001.

C. Reddy and P. Tadepalli. Learning Horn definitions with equivalence and membership queries. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 243–255. Springer, 1997.

C. Reddy and P. Tadepalli. Learning first-order acyclic Horn programs from entailment. In D. Page, editor, *Proceedings of the 8th International Con-*

ference on Inductive Logic Programming, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 23–37. Springer, 1998.

J. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, pages 135–152. Edinburgh University Press, 1970.

B. Richards and R. J. Mooney. Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, 19:95–131, 1995.

M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

K. Rosen. *Discrete Mathematics and its Applications.* McGraw-Hill, 6th edition, 2007.

C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14(2):219–232, 1994.

S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach.* Prentice Hall, 2nd edition, 2004.

C. Sammut. The origins of inductive logic programming: A prehistoric tale. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 127–148. J. Stefan Institute, 1993.

C. Sammut and R. B. 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 167–192. Morgan Kaufmann, 1986.

S. Sanner and C. Boutilier. Approximate linear programming for first-order MDPs. In *Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence*, pages 509–517, 2005.

S. Sanner and K. Kersting. Symbolic dynamic programming. In C. Sammut, editor, *Encyclopedia of Machine Learning.* Springer, 2009, to appear.

T. Sato. A statistical learning method for logic programs with distribution semantics. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 715–729. MIT Press, 1995.

T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

T. Sato and Y. Kameya. Prism: A symbolic-statistical modeling language. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1330–1339. Morgan Kaufmann, 1997.

T. Sato, Y. Kameya, and N.-F. Zhou. Generative modeling with failure in prism. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 847–852, 2005.

R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5: 197–227, 1990.

T. Scheffer, R. Herbrich, and F. Wysotzki. Efficient theta-subsumption based on graph algorithms. In S. Muggleton, editor, *Proceedings of the 6th Inter-*

national Inductive Logic Programming Workshop, volume 1314 of *Lecture Notes in Computer Science*, pages 212–228. Springer, 1997.

B. Schölkopf and A. Smola. *Learning with Kernels.* MIT Press, 2002.

M. P. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann, 1989.

E. Y. Shapiro. Inductive inference of theories from facts. In J. L. Lassez and G. D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 199–255. MIT Press, 1991.

E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

P. Singla and P. Domingos. Discriminative training of Markov logic networks. In M. M. Veloso and S. Kambhampati, editors, *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 868–873. AAAI Press / MIT Press, 2005.

A. Srinivasan. A study of two sampling methods for analyzing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123, 1999.

A. Srinivasan. *The Aleph Manual*, 2007. URL `http://www2.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph_toc.html`.

A. Srinivasan and R. D. King. Feature construction with inductive logic programming: a study of quantitative predictions of biological activity aided by structural attributes. *Data Mining and Knowledge Discovery*, 3(1):37–57, 1999a.

A. Srinivasan and R. D. King. Using inductive logic programming to construct Structure-Activity Relationships. In G. C. Gini and A. R. Katrizsky, editors, *Proceedings of the AAAI Spring Symposium on Predictive Toxicology of Chemicals: Experiences and Impact of AI Tools*, pages 64–73. AAAI Press, 1999b.

A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1/2):277–299, 1996.

R. E. Stepp and R. S. Michalski. Conceptual clustering of structured objects: A goal-oriented approach. *Artificial Intelligence*, 28(1):43–69, 1986.

L. Sterling and E. Y. Shapiro. *The Art of Prolog.* MIT Press, 1986.

M. E. Stickel. A Prolog technology theorem prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

P. D. Summers. A methodology for Lisp program construction from examples. *Journal of the ACM*, 24(1):161–175, 1977.

R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

B. Taskar, E. Segal, and D. Koller. Probabilistic clustering in relational data. In B. Nebel, editor, *Proceedings 17th International Joint Conference on Artificial Intelligence*, pages 870–87, Seattle, Washington, USA, 2001. Morgan Kaufmann.

G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

A. Tobudic and G. Widmer. Relational IBL in classical music. *Machine Learning*, 64(1-3):5–24, 2005.

H. Toivonen. Sampling large databases for association rules. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of 22th International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufmann, 1996.

L. Valiant. A theory of the learnable. *Communications of the ACM*, 27: 1134–1142, 1984.

A. Van Assche, C. Vens, H. Blockeel, and S. Džeroski. First-order random forests: Learning relational classifiers with complex aggregates. *Machine Learning*, 64(1-3):149–182, 2006.

P. R. J. van der Laag and S.-H. Nienhuys-Cheng. Existence and nonexistence of complete refinement operators. In F. Bergadano and L. De Raedt, editors, *Proceedings of the 7th European Conference on Machine Learning*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 307–322. Springer, 1994.

W. Van Laer and L. De Raedt. How to upgrade propositional learners to first order logic: A case study. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 235–261. Springer, 2001.

M. van Otterlo. *The Logic of Adaptive Behavior*. PhD thesis, University of Twente, 2008.

J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *Proceedings of the 20th International Conference on Logic Programming*, pages 431–445, 2004.

J. Vennekens, M. Denecker, and M. Bruynooghe. Representing causal information about a probabilistic process. In M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, editors, *Proceedings of the 10th European Conference on Logics in Artificial Intelligence*, volume 4160 of *LNCS*, pages 452–464. Springer, 2006.

C. Vens, J. Ramon, and H. Blockeel. Refining aggregate conditions in relational learning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Proceedings of the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 4213 of *Lecture Notes in Computer Science*, pages 383–394. Springer, 2006.

S. A. Vere. Induction of concepts in the predicate calculus. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 282–287. Morgan Kaufmann, 1975.

G. Wachman and R. Khardon. Learning from interpretations: A rooted kernel for ordered hypergraphs. In *Proceedings of the 24th International Conference on Machine Learning*, 2007.

C. Wang, S. Joshi, and R. Khardon. First-order decision diagrams for relational MDPs. *Journal of Artificial Intelligence Research*, 31:431–472, 2008.

T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explorations*, 5(1):59–68, 2003.

T. Washio, J. Kok, and L. De Raedt, editors. *Advanced in Mining Graphs, Trees and Sequences.* IOS Press, 2005.

L. Watanabe and L. Rendell. Learning structural decision trees from examples. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 770–776. Morgan Kaufmann, 1991.

I. Weber. Discovery of first-order regularities in a relational database using offline candidate determination. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 288–295. Springer, 1997.

R. Wirth. Learning by failure to prove. In D. Sleeman, editor, *Proceedings of the 3rd European Working Session on Learning.* Pitman, 1988.

S. Wrobel. First-order theory refinement. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*, pages 14–33. IOS Press, 1996.

A. Yamamoto. An inference method for the complete inverse of relative subsumption. *New Generation Computing*, 17(1):99–117, 1999a.

A. Yamamoto. Revising the logical foundations of inductive logic programming systems with ground reduced program. *New Generation Computing*, 17(1):119–127, 1999b.

X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2nd IEEE International Conference on Data Mining*, pages 721–724, 2002.

X. Yin, J. Han, J. Yang, and P. S. Yu. Crossmine: Efficient classification across multiple database relations. In *Proceedings of the 20th International Conference on Data Engineering*, pages 399–411. IEEE Computer Society, 2004.

X. Yin, J. Han, J. Yang, and P. S. Yu. Efficient classification across multiple database relations: A crossmine approach. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):770–783, 2006.

J. M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 1050–1055. AAAI Press/MIT Press, 1996.

J.-D. Zucker and J.-G. Ganascia. Learning structurally indeterminate clauses. In D. Page, editor, *Proceedings of the 8th International Workshop on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Computer Science*, pages 235–244. Springer, 1998.

Author Index

Index