Learning Comprehensible Theories from Structured Data

Kee Siong Ng

A thesis submitted for the degree of
Doctor of Philosophy of
The Australian National University

April 2005
(Revised February 2006)

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Kee Siong Ng
17 April 2005

To my parents
Lim Lian Hua and Ng Yong Swee
For their love, dedication, and courage.

Acknowledgements

People don't have to like or support you,
so you always have to say thank you.
Ruben Studdard

I would like to thank

my long-time supervisor John Lloyd for guidance and assistance of *every* kind going back five years. It's hard to imagine one can have a better supervisor.

my advisors Alex Smola and Gunnar Rätsch for exposing me to statistical machine learning through discussions and the weekly machine learning reading group.

my friend and ex-officemate Evan Greensmith for technical inputs of various kinds.

my colleagues (some of them ex-colleagues) Tony Bowers, Joshua Cole, Matthew Gray, Eric McCreath, Mathi Nagarajan, Vineet Nair and Xiaobing Wu for helpful discussions and valuable collaborations.

Michelle for making life so easy for everyone in the department.

my hosts during my travel to Europe Peter Flach, Stephen Muggleton, John Shawe-Taylor, James Cussens, and Luc De Raedt. I learnt a lot from that trip.

Dr Yin-tak Woo of the US Environmental Protection Agency and Dr Nick Dixon and Professor Rod Rickards of the Research School of Chemistry at the ANU for expert advice on predictive toxicology. Jean Jiayu Wen and Yu Di also helped in my (futile) endeavour to understand a little more of biology and chemistry.

The Australian National University and the Smart Internet Cooperative Research Centre for their generous scholarships.

my friends F.H. Huang (Dajie), Linda, Thararat, Yee Tuan, Agnes, Cheng Soon, Doug, Edward and Annie, Evan, Kerry, Kristy, Wongas (and Fiona), and Xiaobing for walks, talks, cakes, (Belgian) chocolates, fish, rice puddings in gula Melaka, sushi, bo-bo cha-cha, delicious curry, etc.

Ginny for being you.

my family for love and unconditional support over the last 27 years; my brother deserves special thanks.

It has been a most rewarding and exciting three years!

Abstract

This thesis is concerned with the problem of learning comprehensible theories from structured data and covers primarily classification and regression learning. The basic knowledge representation language is set around a polymorphically-typed, higher-order logic. The general setup is closely related to the learning from propositionalized knowledge and learning from interpretations settings in Inductive Logic Programming. Individuals (also called instances) are represented as terms in the logic. A grammar-like construct called a predicate rewrite system is used to define features in the form of predicates that individuals may or may not satisfy. For learning, decision-tree algorithms of various kinds are adopted.

The scope of the thesis spans both theory and practice. On the theoretical side, I study in this thesis

1. the representational power of different function classes and relationships between them;
2. the sample complexity of some commonly-used predicate classes, particularly those involving sets and multisets;
3. the computational complexity of various optimization problems associated with learning and algorithms for solving them; and
4. the (efficient) learnability of different function classes in the PAC and agnostic PAC models.

On the practical side, the usefulness of the learning system developed is demontrated with applications in two important domains: bioinformatics and intelligent agents. Specifically, the following are covered in this thesis:

1. a solution to a benchmark multiple-instance learning problem and some useful lessons that can be drawn from it;
2. a successful attempt on a knowledge discovery problem in predictive toxicology, one that can serve as another proof-of-concept that *real* chemical knowledge can be obtained using symbolic learning;
3. a reworking of an exercise in relational reinforcement learning and some new insights and techniques we learned for this interesting problem; and
4. a general approach for personalizing user agents that takes full advantage of symbolic learning.

Contents

Chapter 1

Introduction

1.1 The General Problem

Learning from examples is an important topic in the study of machine intelligence. We consider the general problem of *learning comprehensible theories from structured data* in this thesis. The three key phrases here are 'learning', 'structured data', and 'comprehensible theories'. We now elaborate on each of these in turn.

Learning The basic learning problem is standard. The learner receives randomly drawn training examples of the form (x, y), $x \in \mathcal{X}$, $y \in \mathcal{Y}$, where $\mathcal{X}$ is some set called the *individual* space, and $\mathcal{Y}$ is either a finite set or (a bounded interval of) the real line $\mathbb{R}$. There could be an underlying target function f relating each x and y by $y = f(x)$, and the examples are generated according to an unknown probability distribution on $\mathcal{X}$. More generally, we do not presuppose the existence of a target function and simply assume that examples are generated according to an unknown joint probability distribution on $\mathcal{X} \times \mathcal{Y}$. In both cases, the learning task is to find a hypothesis h from a class of functions $\mathcal{H}$ called the hypothesis space that generalizes well to unseen examples. That is to say, h is our theory of the underlying process that generates the training examples.

Structured Data In a typical formulation of the learning problem, $\mathcal{X}$ is a (strict) subset of $\mathbb{R}^n$ for some $n > 0$. This representation is inconvenient for certain domains like bioinformatics where the individuals at work are complex entities with rich internal structures. To widen the applicability of machine learning to these areas, we need to move to a richer instance space $\mathcal{X}$, one capable of modelling a wide range of structured data, and consider the problem of learning in that setting.

Comprehensible Theories Besides being able to deal with structured data, we would also like the theories we produce to be, whenever possible, comprehensible. This is a worthy goal to aim for. If we expect our theory to be useful enough to be assimilated into the body of human knowledge, then it must have explanatory power and be amenable to some form of scrutiny, and comprehensibility helps in that regard.

The general problem is, by popular agreement, an important one. There are different ways to tackle the problem. We next outline an approach based on the use of computational logic.

1.2 A Symbolic Approach

In [130], an elegant logical framework is proposed for the problem of learning comprehensible theories from structured data. The framework is based on a polymorphically-typed, higher-order logic and forms the backbone of the present thesis. The main features of the formalism are as follows. (More details can be found in Chapter 2.)

1. A special class of terms in the logic called *basic terms* is used for data modelling. A rich catalogue of data types is supported this way. They include integers, floating-point numbers, data constants, tuples, lists, trees, graphs, sets, multi-sets, and composite types that can be built up from these more basic types.

2. A class of functions called transformations provides a convenient way to define and incorporate domain knowledge into learning. Transformations can be composed to form (complex) predicates on individuals, and these form part of the hypothesis space in applications.

   Predicate classes relevant to an application can be compactly defined and efficiently enumerated using a construct called predicate rewrite systems. Essentially, a predicate rewrite system is a grammar for constructing predicates from transformations.

3. For the purpose of learning, the standard top-down tree-induction algorithm was 'upgraded', in the sense of [194], to handle basic terms and higher-order predicates. Decision-tree learning was adopted because it is one of only a few learning algorithms that can produce comprehensible rules.

The knowledge representation aspects of the framework are well-understood. The learning aspects, as documented in [130], however, could do with more development. The aim of this thesis is to fill some of these gaps in our overall understanding.

1.3 Two Scientific Questions

We have outlined the general problem and the symbolic approach taken. We now state two specific scientific questions that drive the development of the present work.

1. What is the nature of learning with an expressive language such is provided by the formalism adopted here?
2. Is the symbolic approach to learning propounded here relevant and applicable?

The first question is interesting from a theoretical perspective. A lot is known about the processes and nature of learning in the restricted setting where individuals are simple feature vectors; see, for an introduction, [3] and [103]. What is fundamentally different when we move on to a richer setting and avail ourselves of a more expressive language for representing individuals and hypothesis spaces? Is the resulting learning problem harder or easier? In what way? Why? These are but some of the questions that we will try to answer in this thesis.

The second question is important from a practical perspective, and it is one we ask to keep ourselves honest. To demonstrate relevance, we will concentrate on two application domains of scientific significance: bioinformatics and agents. In the context of bioinformatics, we will show that symbolic learning can help in the discovery of real biochemical knowledge. In the context of agents, we will show that symbolic learning can be used, in a complementary way with other components in an agent architecture, to incorporate adaptability into the behaviour of autonomous and intelligent agents.

> If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.
>
> *Donald E. Knuth* [111]

1.4 Thesis Organization

The thesis is organized as follows. In Chapter 2, we summarize the essential aspects of the knowledge representation formalism introduced in [130] and describe a few simple tools for manipulating predicate rewrite systems.

Chapter 3 is concerned with the basic problem of learning classification rules from structured data. In it, we study different decision-tree learning algorithms and analyze their properties from a few standard learning-theoretic viewpoints. The material presented in this chapter sheds light on the nature of learning with a rich language.

In Chapter 4, we extend, in a straightforward manner, the basic tree-learning algorithm to regression. In Chapter 5, we introduce two incremental induction algorithms, one for classification and one for regression. Each one is, in a sense to be made precise later, lossless with respect to its corresponding batch learning algorithm. These extensions broaden the applicability of the present work.

Chapter 6 contains descriptions of four applications, two in bioinformatics, two in agents. Two of them are 'toy problems'[1], two non-trivial problems.

Chapter 7 compares the general approach presented in this thesis with related work in the literature. Both qualitative and quantitative analyses are given.

We conclude in Chapter 8 with an evaluation of what has been achieved in this thesis and state some possible future work.

1.5 The Alkemy Software

As part of development of the work reported here, a prototype system called ALKEMY was implemented and made available for download at

[1] Toy problems can be relevant; see [110].

`http://rsise.anu.edu.au/~kee/Alkemy`.

The system is a fairly complex piece of software ($\sim$ 20,000 lines of `C++` code). It was written as a literate program [154] using Noweb [165]. All the algorithms discussed in this thesis are implemented. The system comes complete with a tutorial.

I like to work in a variety of fields
in order to spread my mistakes more thinly.
Victor Klee (1999)

Chapter 2

Knowledge Representation

Language is the armory of the human mind, and at once
contains the trophies of its past and the weapons of its future conquests.
Samuel Taylor Coleridge

2.1 Introduction

The representation language adopted in this thesis is the one proposed in [130]. The basic setting is a typed, higher-order logic based on Alonzo Church's simple theory of types [41] with several extensions, most notably the support of polymorphism in the type system. The form of the language is similar to that of a standard functional programming language like Haskell. In fact, the formalism grew out of research into a functional logic programming language called Escher [128], [129].

In this chapter, we review the fundamental elements of that formalism, starting with the representation of individuals in Section 2.2. A rich catalogue of data types is provided for that purpose. They include integers, floating-point numbers, characters, strings, booleans, data constructors, tuples, sets, multisets, lists, trees, graphs and composite types that can be built up from these more basic types. We will take a brief look at how these are made available through the concept of basic terms.

Besides individuals, we also need a language to express features about individuals. Here, features are identified with properties that individuals may or may not have, in other words, predicates. A class of predicates that can be built up from more basic functions called transformations suitable for use here is introduced in Section 2.3. Section 2.4 then describes a mechanism based on term rewriting that can be used to define a collection of predicates relevant to a particular application.

We end the chapter with a listing of some common data types and transformations in Section 2.5. They are used at different places throughout the thesis, and are collected here for easy referencing.

Before moving on, we give a few pointers to where information beyond what is given in this chapter can be obtained. With the exception of material in §2.4.3, the specifics of every concept presented in this chapter can be found in [130]. Discussions on the merits of the formalism and comparisons with existing first-order and propositional representations can be found in Sect. 7.1, [31], [83] and [76].

2.2 Representation of Individuals

The formal basis for the representation of individuals is provided by the concept of a *basic term*. Essentially, one first defines the concept of a term in higher-order logic. A suitably rich subset is then identified for data modelling. We give a brief outline of its development here.

We assume there is given a set of type constructors of various arities. The *types* of the logic are expressions that can be built up from the set of type constructors and a set of parameters (type variables) using the symbols $\rightarrow$ and $\times$. The former is used to construct function types, the latter, product types. Parameters are usually denoted using the symbols $a, b, c, \ldots$.

Example 2.2.1. Standard type constructors (of arity 0) include Ω (the type of the booleans), Nat (the type of natural numbers), Int (the type of integers), $Float$ (the type of floating-point numbers), $Char$ (the type of characters) and $String$. ◀

Example 2.2.2. $List$ is a type constructor that is used to provide list types. If α is a type, then $List\ \alpha$ is the type of lists whose elements have type α. ◀

There is also a set of constants of various types. Included in the set are 1 (true) and 0 (false), each of type Ω. One can distinguish between two kinds of constants, *data constructors* and *functions*. In a knowledge representation context, data constructors are used to represent individuals. Functions have definitions, data constructors do not. A *signature* is the declared type of a constant. If a constant C has signature α, we denote this by $C : \alpha$.

Example 2.2.3. The constant [] is the empty list constructor with signature $List\ a$, where a is a parameter. The constant $\# : a \rightarrow List\ a \rightarrow List\ a$ is a list constructor. It takes two arguments, an element of type α and a list of type $List\ \alpha$, and produces a new list of type $List\ \alpha$. (Here, α is unified with the parameter a.) ◀

The *terms* of the logic are the terms of the typed λ-calculus, formed in the usual way by application, abstraction, and tupling from the set of constants and a set of variables. The definition of *basic terms* – a strict subset of terms – has three parts: the first part gives those basic terms that have a data constructor at the top level (integers, characters, lists, trees, etc); the second part gives certain abstractions that include sets and multisets; the third part gives tuples. The exact definition is in [130].

We now describe a simple problem to give a feel of the data modeling language. It will be used as a running example to illustrate the different concepts we will encounter in this chapter.

Example 2.2.4. Consider the East-West challenge proposed by Michalski. (See, for example, [140].) Given ten trains and the directions they are traveling in (Figure 2.1), the problem is to come up with a rule that can differentiate between those heading east and those heading west. To solve the problem, we first need a way of representing the trains. How might we do that? Recognizing the fact that the order in which the carriages appear in a train might be important, we can model a train as a list of

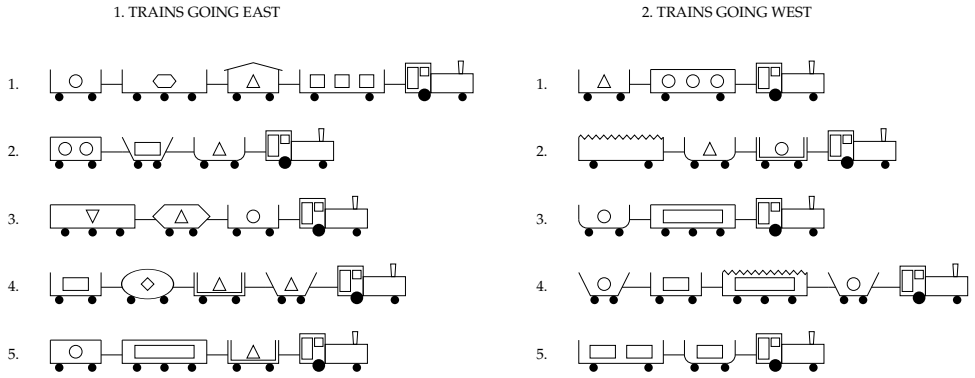


Figure 2.1: Trains in Michalski's East-West Challenge.

its carriages. A carriage can in turn be modeled using a tuple made up of six of its properties - shape, length, number of wheels, open air or closed, roof shape, and the load the carriage is carrying. From that initial analysis, we introduce the following data constructors and type synonyms.

$$
\begin{aligned}
&\mathit{Rectangular}, \mathit{DoubleRectangular}, \mathit{UShaped}, \mathit{BucketShaped}, \\
&\qquad\qquad \mathit{Hexagonal}, \mathit{Ellipsoidal} : \mathit{Shape} \\
&\mathit{Long}, \mathit{Short} : \mathit{Length} \\
&\mathit{Closed}, \mathit{Open} : \mathit{Kind} \\
&\mathit{Flat}, \mathit{Jagged}, \mathit{Peaked}, \mathit{Curved}, \mathit{None} : \mathit{Roof} \\
&\mathit{Circle}, \mathit{Hexagon}, \mathit{Square}, \mathit{Rectangle}, \mathit{LRectangle}, \mathit{Triangle}, \mathit{UTriangle}, \\
&\qquad\qquad \mathit{Diamond}, \mathit{Null} : \mathit{Object} \\
&\mathit{East}, \mathit{West} : \mathit{Direction} \\
&\mathit{NumWheels} = \mathit{Int} \\
&\mathit{NumObjects} = \mathit{Int} \\
&\mathit{Load} = \mathit{Object} \times \mathit{NumObjects} \\
&\mathit{Car} = \mathit{Shape} \times \mathit{Length} \times \mathit{NumWheels} \times \mathit{Kind} \times \mathit{Roof} \times \mathit{Load} \\
&\mathit{Train} = \mathit{List}\ \mathit{Car}.
\end{aligned}
$$

Note that type synonyms are denoted using $=$ signs here; their use allows us to give meaningful names to types. Given these type declarations, formally, the task is to learn the definition of a function *direction* with signature $\mathit{Train} \rightarrow \mathit{Direction}$. Individuals in the training examples can be easily encoded as terms of type $\mathit{List}\ \mathit{Car}$. The

first trains traveling in each direction are shown below.

$$\begin{aligned}
&\mathit{direction}\ [(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Open}, \mathit{None}, (\mathit{Square}, 3)),\\
&\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Closed}, \mathit{Peaked}, (\mathit{Triangle}, 1)),\\
&\qquad (\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Open}, \mathit{None}, (\mathit{Hexagon}, 1)),\\
&\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1))] = \mathit{East}\\
&\mathit{direction}\ [(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Closed}, \mathit{Flat}, (\mathit{Circle}, 3)),\\
&\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1))] = \mathit{West}.
\end{aligned}$$

◀

2.3 Representation of Features

We now turn our attention to the formalism for expressing features. We identify boolean features with predicates that individuals may or may not satisfy. A predicate is a function with a signature of the form $\alpha \to \Omega$ for some α.

The class of predicates we are interested in are built up incrementally by composition of simpler functions called transformations. Composition is handled by the (reverse) composition function

$$\circ : (a \to b) \to (b \to c) \to (a \to c)$$

defined by $((f \circ g)\ x) = (g\ (f\ x))$.

Definition 2.3.1. A *transformation* f is a function having a signature of the form

$$f : (\varrho_1 \to \Omega) \to \cdots \to (\varrho_k \to \Omega) \to \mu \to \sigma,$$

where any parameters in $\varrho_1, \ldots, \varrho_k$ and σ appear in μ, and $k \geqslant 0$. The type μ is called the *source* of the transformation, while the type σ is called the *target* of the transformation. The number k is called the *rank* of the transformation.

The idea behind the definition of transformation is rather intuitive. Given predicates $p_i : \varrho_i \to \Omega$ $(i = 1, \ldots, k)$, $f\ p_1 \ldots p_k$ is a function that takes individuals of type μ to individuals of type σ. By composing several such functions, the last of which is a transformation with target type boolean, a predicate on individuals of the desired type is obtained.

Note that every function having signature $\mu \to \sigma$ is potentially a transformation. Just put $k = 0$.

Example 2.3.2. There are two fundamental transformations $\mathit{top} : a \to \Omega$ and $\mathit{bottom} : a \to \Omega$ defined by $(\mathit{top}\ x) = 1$ and $(\mathit{bottom}\ x) = 0$, for each x. Each of top and bottom is a constant predicate, with top being the weakest predicate on the type a and bottom, the strongest. We will see more examples of transformations shortly. ◀

We now give the definition of the class of predicates that can be formed by composing transformations. In the definition, it is assumed that some (possibly infinite) class of transformations is given and all transformations considered are taken from this class.

Definition 2.3.3. A *standard predicate* is a term of the form

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n}),$$

where f_i is a transformation of rank k_i ($i = 1, \ldots, n$), the target of f_n is Ω, p_{i,j_i} is a standard predicate ($i = 1, \ldots, n,\ \ j_i = 1, \ldots, k_i$), $k_i \geqslant 0$ ($i = 1, \ldots, n$) and $n \geqslant 1$.

The set of all standard predicates is denote by $\mathbf{S}$.

Example 2.3.4. Referring back to the East-West challenge described in Example 2.2.4, one may ask what transformations are suitable for use with the learning task? The type we have chosen for trains gives strong clues about the possible transformations. At the top level, a train is a list of carriages. Here are some standard transformations on lists.

$$\begin{aligned} &head : \mathit{Train} \to \mathit{Car} \\ &head\ (x : xs) = x \end{aligned}$$

$$\begin{aligned} &tail : \mathit{Train} \to \mathit{Train} \\ &tail\ (x : xs) = xs \end{aligned}$$

$$\begin{aligned} &listToSet : \mathit{Train} \to \{\mathit{Car}\} \\ &listToSet\ y = \{x\ :\ member\ x\ y\} \end{aligned}$$

Given a set of carriages, the following transformations can be useful.

$$\begin{aligned} &setExists_1 : (a \to \Omega) \to \{a\} \to \Omega \\ &setExists_1\ p\ t = \exists x.((p\ x) \wedge (x \in t)) \end{aligned}$$

$$\begin{aligned} &domCard : (a \to \Omega) \to \{a\} \to \mathit{Nat} \\ &domCard\ p\ t = card\ \{x \mid (p\ x) \wedge x \in t\}. \end{aligned}$$

Given a predicate p and a set t having appropriate types,

- $setExists_1\ p\ t$ evaluates to 1 iff the set t has at least one element that satisfies the predicate p;
- $domCard\ \ p\ \ t$ returns the number of elements in t that satisfy p. (The function $card$ computes the cardinality of a set.)

Corresponding to the types Car and $Load$, which are tuples, we can define trans-

formations that project onto the components of the tuples.

$$
\begin{aligned}
&\mathit{projShape} : \mathit{Car} \to \mathit{Shape}\\
&\mathit{projShape}\ (t_1, t_2, t_3, t_4, t_5, t_6) = t_1\\
&\ldots\\
&\mathit{projLoad} : \mathit{Car} \to \mathit{Load}\\
&\mathit{projLoad}\ (t_1, t_2, t_3, t_4, t_5, t_6) = t_6\\
&\mathit{projObject} : \mathit{Load} \to \mathit{Object}\\
&\mathit{projObject}\ (t_1, t_2) = t_1\\
&\mathit{projNumObjects} : \mathit{Load} \to \mathit{NumObjects}\\
&\mathit{projNumObjects}\ (t_1, t_2) = t_2
\end{aligned}
$$

We may also want to form conjunctions of predicates (of arbitrary types). This can be defined as folows.

$$
\begin{aligned}
&\wedge_n : (a \to \Omega) \to \cdots \to (a \to \Omega) \to a \to \Omega\\
&\wedge_n\ p_1 \ldots p_n = \lambda x.((p_1\ x) \wedge \cdots \wedge (p_n\ x))
\end{aligned}
$$

The lambda notation is used here. Thus, given a term t of the appropriate type, $\wedge_n\ p_1 \ldots p_n\ t$ evaluates to 1 iff the term $((p_1\ t) \wedge \cdots \wedge (p_n\ t))$ evaluates to 1. We remark that disjunctions $\vee_n$ and negation $\neg$ at the predicate level can be defined in a similar manner.

Corresponding to the data constructors and integers, we can have the following transformations that check for equality of terms.

$$
\begin{aligned}
&(= \mathit{Rectangular}) : \mathit{Shape} \to \Omega\\
&(= \mathit{Rectangular})\ x = (x = \mathit{Rectangular})\\
&(= \mathit{DoubleRectangular}) : \mathit{Shape} \to \Omega\\
&(= \mathit{DoubleRectangular})\ x = (x = \mathit{DoubleRectangular})\\
&\ldots\\
&(= \mathit{Null}) : \mathit{Object} \to \Omega\\
&(= \mathit{Null})\ x = (x = \mathit{Null})\\
&(= 2) : \mathit{NumWheels} \to \Omega\\
&(= 2)\ x = (x = 2)\\
&(= 3) : \mathit{NumWheels} \to \Omega\\
&(= 3)\ x = (x = 3)
\end{aligned}
$$

Corresponding to any integer N, we can define the transformation $(< N)$ as fol-

lows. In a similar way, one can define $(> N)$, $(\geqslant N)$, and $(\leqslant N)$.

$$(< N) : \mathit{Int} \to \Omega$$
$$((< N)\ m) = (m < N).$$

There are many other appropriate transformations one can think of, but the small set we have identified can already be used to express complex conditions on trains. For example, the predicate

$$\mathit{listToSet} \circ (\mathit{setExists}_1\ (\mathit{projLength} \circ (= \mathit{Short})))$$

takes as input an object of type Train, converts it into a set of carriages, and checks whether there exists a short carriage in the set. For another example, the predicate

$$\mathit{head} \circ (\wedge_2\ (\mathit{projShape} \circ (= \mathit{Rectangular}))\ (\mathit{projLoad} \circ \mathit{projNumObjects} \circ (= 3)))$$

takes as input an object of type Train, pulls out the first carriage and checks whether its shape is rectangular and that it carries three objects. ◀

2.4 Predicate Construction

We next describe a mechanism to define and enumerate a set of predicates relevant to a particular application. We start with the definition of a predicate rewrite system.

Definition 2.4.1. A *predicate rewrite system* is a finite relation $\rightarrowtail$ on $\mathbf{S}$ (the set of all standard predicates) satisfying the following property: for each $p \rightarrowtail q$, the type of p is more general than the type of q.

Informally, a type α is more general than a type β if there exists a type substitution ξ instantiating parameters in α such that $\beta = \alpha\xi$.

Each $p \rightarrowtail q$ in $\rightarrowtail$ is called a *predicate rewrite*, p the *head* and q the *body* of the predicate rewrite.

The next few definitions make clear how a predicate rewrite system defines a predicate search space.

Definition 2.4.2. A subterm of a standard predicate

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$$

is *eligible* if it is a suffix of the standard predicate or it is an eligible subterm of p_{i,j_i}, for some $i \in \{1, \ldots, n\}$ and $j_i \in \{1, \ldots, k_i\}$.

Definition 2.4.3. Given a predicate rewrite system $\rightarrowtail$ and a standard predicate p, an eligible subterm r of p is a *redex* with respect to $\rightarrowtail$ if there exists a predicate rewrite $r \rightarrowtail b$ such that the type of b and the type of r in p are unifiable. We say r is a redex *via* $r \rightarrowtail b$.

Definition 2.4.4. Let $\rightarrowtail$ be a predicate rewrite system and p and q standard predicates. Then q is obtained by a *predicate derivation step* from p using $\rightarrowtail$ if there is a redex r via $r \rightarrowtail b$ in p and $q = p[r/b]$. Here, $p[r/b]$ denotes the predicate obtained from p by replacing r in p with b.

In applications, to generate a search space of predicates, we start from some predicate p_0 and generate all the predicates that can be obtained by a predicate derivation step from p_0, then all the predicates that can be obtained from those by a predicate derivation step, and so on.

Definition 2.4.5. A *predicate derivation* with respect to a predicate rewrite system $\rightarrowtail$ is a finite sequence $\langle p_0, p_1, \ldots, p_n \rangle$ of standard predicates such that p_i is obtained by a derivation step from p_{i-1} using $\rightarrowtail$, for $i = 1, \ldots, n$. The standard predicate p_0 is called the *initial predicate* and the standard predicate p_n is called the *final predicate*.

Example 2.4.6. Consider Example 2.2.4 again. Suppose we speculate that the definition of *direction* has the following general form: a train travels east (or west) iff there exists a carriage in the train that satisfies some (as yet unknown) properties. The following predicate rewrite system specifies a space of predicates of the desired form.

$$
\begin{aligned}
&top \rightarrowtail listToSet \circ setExists_1\ (\wedge_3\ top\ top\ top)\\
&top \rightarrowtail projShape \circ top\\
&top \rightarrowtail projLength \circ top\\
&top \rightarrowtail projNumWheels \circ top\\
&top \rightarrowtail projKind \circ top\\
&top \rightarrowtail projRoof \circ top\\
&top \rightarrowtail projLoad \circ top\\
&top \rightarrowtail projObjects \circ top\\
&top \rightarrowtail projNumObjects \circ top\\
&top \rightarrowtail (= Rectangular)\\
&top \rightarrowtail (= DoubleRectangular)\\
&top \rightarrowtail (= UShaped)\\
&\ldots\\
&top \rightarrowtail (= UTriangle)\\
&top \rightarrowtail (= Diamond)\\
&top \rightarrowtail (= Null)\\
&top \rightarrowtail (= 1)\\
&top \rightarrowtail (= 2)\\
&top \rightarrowtail (= 3)
\end{aligned}
$$

It is clear that the set of predicate rewrites given above satisfies the condition of Defi-

nition 2.4.1. The following is a predicate derivation.

$$
\begin{aligned}
&\mathit{top}\\
&\mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_3\ \mathit{top}\ \mathit{top}\ \mathit{top})\\
&\mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_3\ (\mathit{projShape} \circ \mathit{top})\ \mathit{top}\ \mathit{top})\\
&\mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_3\ (\mathit{projShape} \circ \mathit{top})\ (\mathit{projLength} \circ \mathit{top})\ \mathit{top})\\
&\mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_3\ (\mathit{projShape} \circ \mathit{top})\ (\mathit{projLength} \circ \mathit{top})\ (\mathit{projKind} \circ \mathit{top}))\\
&\mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_3\ (\mathit{projShape} \circ (= \mathit{Rectangular}))\\
&\qquad\qquad (\mathit{projLength} \circ \mathit{top}))\ (\mathit{projKind} \circ \mathit{top}))\\
&\mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_3\ (\mathit{projShape} \circ (= \mathit{Rectangular}))\\
&\qquad\qquad (\mathit{projLength} \circ \mathit{top})\ (\mathit{projKind} \circ (= \mathit{Open})))\\
&\mathit{listToSet} \circ \mathit{setExists}_1\ (\wedge_3\ (\mathit{projShape} \circ (= \mathit{Rectangular}))\\
&\qquad\qquad (\mathit{projLength} \circ (= \mathit{Short}))\ (\mathit{projKind} \circ (= \mathit{Open})))
\end{aligned}
$$

We used top as the initial predicate. The first predicate rewrite is used in the first step. (It is intended that the derivation provides a predicate of type $\mathit{Train} \to \Omega$.) For the second step, the redex top has type $\mathit{Car} \to \Omega$ which is the same as the type of the body of the predicate rewrite $\mathit{top} \rightarrowtail \mathit{projShape} \circ \mathit{top}$. Thus this occurrence of top is a redex via this predicate rewrite. The remaining steps are similar.

Without spelling out the algorithm for generating the search space given a predicate rewrite system (this will be done next in §2.4.1), we remark that ALKEMY searched through the predicate space and produced the following hypothesis for the problem.

$$
\begin{aligned}
&\mathit{direction}\ t = \mathit{if}\ \mathit{listToSet} \circ (\mathit{setExists}_1\ (\wedge_3\ \mathit{projLength} \circ (= \mathit{Short})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{projKind} \circ (= \mathit{Closed})\ \mathit{top}))\ t\\
&\qquad\qquad \mathit{then}\ \mathit{East}\\
&\qquad\qquad \mathit{else}\ \mathit{West}.
\end{aligned}
$$

In more user-friendly terms, "A train is eastbound iff it has a short closed car". ◀

2.4.1 Predicate Enumeration

Given a predicate rewrite system $\rightarrowtail$ and a predicate p_0, we call the *expected predicates obtainable from* p_0 *via* $\rightarrowtail$, denoted $S_{\rightarrowtail}(p_0)$ (or just $S_{\rightarrowtail}$ when p_0 is understood), the set of all the the final predicates of predicate derivations with initial predicate p_0 that can be formed with no restrictions on the selection of redexes at each derivation step. To efficiently enumerate expected predicates in practical applications, some care is necessary. The difficulty is that the search space defined by a predicate rewrite system is (usually) a graph, not a tree. In other words, there may be many paths from some initial predicate p to some predicate q. The first problem is related to the selection order of the redexes.

Example 2.4.7. Consider the predicate rewrite system in Example 2.4.6. Corresponding to the different choices of redexes, there are 3! distinct predicate derivations with initial predicate top that have

$$listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ (= Rectangular))\ (projLength \circ (= Short))\ (projKind \circ (= Open))$$

as the final predicate. Clearly, it would be preferable to construct only one of these derivations. ◀

The second problem is related to equivalence of predicates.

Example 2.4.8. Consider again the predicate rewrite system in Example 2.4.6. The standard predicates

$$listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ (= Rectangular))\ (projLength \circ (= Short))\ (projKind \circ (= Open)))$$

and

$$listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ (= Rectangular))\ (projKind \circ (= Open))\ (projLength \circ (= Short)))$$

can both be obtained as the final predicates of derivations starting from the initial predicate top. But these two standard predicates are logically equivalent. There are 3! such logically equivalent predicates altogether corresponding to the various orderings of the arguments to $\wedge_3$. Clearly, it would be preferable to only construct one of these predicates. ◀

We now give two algorithms for systematically enumerating the search space defined by a predicate rewrite system. Each has features not available in the other. Depending on the application, one may be preferred to the other. Algorithm I, first reported in [31], was designed by John W. Lloyd, Antony Bowers and Christophe Giraud-Carrier. Algorithm II was joint work between John W. Lloyd and the author and first appeared in [130].

Both algorithms make use of the following syntactic condition that reveals equivalence of predicates.

Definition 2.4.9. A transformation f is *symmetric* if it has a signature of the form

$$f : (\varrho \to \Omega) \to \cdots \to (\varrho \to \Omega) \to \mu \to \sigma,$$

and whenever $f\ p_1 \ldots p_k$ is a standard predicate, it follows that for all permutations i of $\{1, \ldots, k\}$, $f\ p_1 \ldots p_k$ and $f\ p_{i_1} \ldots p_{i_k}$ are equivalent.

Example 2.4.10. The transformation $\wedge_n$ is symmetric. Also, every transformation of rank k, where $k \leqslant 1$, is (trivially) symmetric. ◀

Since any permutation of the predicate arguments of a symmetric transformation produces an equivalent function, we can choose to allow only one particular order of arguments and ignore the others. For this purpose, a total order $\preceq$ on standard predicates needs to be defined. (For details, see [130].) Arguments for symmetric transformations are then chosen in increasing order according to $\preceq$. The class of regular predicates is defined as follows.

Definition 2.4.11. A standard predicate $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$ is *regular* if p_{i,j_i} is a regular predicate, for $i = 1, \ldots, n$ and $j_i = 1, \ldots, k_i$, and f_i is symmetric implies that $p_{i,1} \preceq \cdots \preceq p_{i,k_i}$, for $i = 1, \ldots, n$.

There is an algorithm called $Regularize$ that takes a standard predicate and returns an equivalent regular standard predicate. It can be found in [130].

2.4.1.1 Algorithm I

Figure 2.2 shows the first algorithm. It is an instance of the classical anagram algorithm [16]. A set of (regularized forms of) previously seen predicates is maintained in a $seenSet$ during search. Each time a new predicate is generated, the algorithm regularizes the predicate and checks whether it is already in the $seenSet$, adding it if it is not. This tactic essentially turns the search space back into a tree.

The algorithm works well when used in conjunction with incomplete search algorithms. Good predicates can be reached quickly via any of the many paths leading to it. A big disadvantage of the algorithm is the (un)scalability of its memory requirement. Asymptotically, the size of the $seenSet$ can be as large as the whole search space. This implies that for sufficiently large search spaces, an exhaustive enumeration is computationally intractable. This motivates the development of the next algorithm, which is memory efficient.

2.4.1.2 Algorithm II

The central idea of the second algorithm is that one can introduce a restricted form of redex selection to solve the problem illustrated in Example 2.4.7, and discard non-regular predicates during enumeration to solve the second problem illustrated in Example 2.4.8.

The algorithm is given in Figure 2.3. In the figure, the phrase 'LR redex' means the redex is selected according to the LR (left-to-right) selection rule, that is, the redex must be at or to the right of the redex selected in the parent predicate of p. More precisely, suppose p is a standard predicate and $\rightarrowtail$ a predicate rewrite system. If r is a redex in p via $r \rightarrowtail b$ and we derive $q = p[r/b]$ from p using $r \rightarrowtail b$, then the LR selection rule stipulates that the only redexes in q that can be chosen next are those that are at or to the right of b in q.

The algorithm, while conceptually simple and elegant, introduces some subtle and difficult questions:

function $Enumerate(\rightarrowtail, p_0)$ **returns** the set of expected predicates obtainable from p_0 via $\rightarrowtail$;
inputs: $\rightarrowtail$, a predicate rewrite system ;
p_0, a standard predicate;

$predicates := \{\}$;
$openList := [p_0]$;
$seenSet := \{p_0\}$;
while $openList \neq []$ **do**
$p := head(openList)$;
$openList := tail(openList)$;
$predicates := predicates \cup \{p\}$;
for each redex r via $r \rightarrowtail b$, for some b, in p **do**
$q := Regularize(p[r/b])$;
if $q \notin seenSet$ **then**
$seenSet := seenSet \cup \{q\}$;
$openList := openList$ ++ $[q]$;
return $predicates$;

Figure 2.2: Algorithm I

function $Enumerate2(\rightarrowtail, p_0)$ **returns** the set of regular predicates that can be derived from p_0 using $\rightarrowtail$;
inputs: $\rightarrowtail$, a predicate rewrite system ;
p_0, a predicate;

$predicates := \{\}$;
$openList := [p_0]$;
while $openList \neq []$ **do**
$p := head(openList)$;
$openList := tail(openList)$;
$predicates := predicates \cup \{p\}$;
for each LR redex r via $r \rightarrowtail b$, for some b, in p **do**
$q := p[r/b]$;
if q is regular **then** $openList := openList$ ++ $[q]$;
return $predicates$;

Figure 2.3: Algorithm II

1. Is the search conducted by the algorithm complete? In other words, are (the regularizations of) all the expected predicates obtainable from p_0 via $\rightarrowtail$ actually generated by such restricted predicate derivations?
2. Is each (regularization of an) expected predicate generated exactly once?

In [130], it is shown that under some weak conditions on the predicate rewrite system, completeness and uniqueness of predicate derivations can be guaranteed. The results are technical and we refer the reader to the book for more details.

Note that Algorithm II is *memoryless*, and is particularly useful for performing exhaustive searches. However, it does not work as well when used in conjunction with incomplete search algorithms. This is because it often cannot go where the search strategy is suggesting it go because only one path out of the many possible ones can be traversed. During search, non-regular predicates generated are thrown away without further examination. One of these predicates could very well be the best predicate in the whole predicate space, but it won't be examined until a regular path to it can be found.

2.4.2 Structuring the Search Space

In the top-down construction of predicates, it is crucial to ensure that if a standard predicate q is obtained by a predicate derivation step from a standard predicate p, then $p \Leftarrow q$ holds, where $\Leftarrow$ denotes logical implication. Sufficient conditions for this are now given.

Informally, we say a standard predicate p is *monotone* with respect to a predicate rewrite system $\rightarrowtail$ if for every redex r in p with respect to $\rightarrowtail$ and for every standard predicates s and s' such that $r \Leftarrow s \Leftarrow s'$, then $p[r/s] \Leftarrow p[r/s']$. (Recall that $p[r/s]$ denotes the predicate obtained from p by replacing r in p with s.) Further details on this definition of monotonicity can be found in [130]. We now give a few examples to illustrate the concept.

A standard predicate that contains no redexes with respect to a predicate rewrite system $\rightarrowtail$ is clearly monotone with respect to $\rightarrowtail$. Also, a predicate p whose only redex with respect to a predicate rewrite system $\rightarrowtail$ is a suffix of p is monotone with respect to $\rightarrowtail$.

Example 2.4.12. If $p_1, \ldots, p_n$ are standard predicates that are monotone with respect to a predicate rewrite system, then so is $\wedge_n\, p_1 \ldots p_n$.

Example 2.4.13. If p is a standard predicate that is monotone with respect to a predicate rewrite system, then so is $(domCard\ p) \circ (> N)$. This is not true of the predicate $(domCard\ p) \circ (< N)$, however.

Definition 2.4.14. We say a predicate rewrite system $\rightarrowtail$ is *monotone* if the following conditions are satisfied.

1. $p \rightarrowtail q$ implies $p \Leftarrow q$.
2. $p \rightarrowtail q$ implies q is monotone with respect to $\rightarrowtail$.

Example 2.4.15. The predicate rewrite system given in Example 2.4.6 is monotone. Clearly, each predicate rewrite $p \rightarrowtail q$ satisfies $p \Leftarrow q$. Also, the body of each predicate rewrite is monotone. Each predicate of the form $(= C)$ for some constant C has no redex and is therefore monotone. Predicates like $projShape \circ top$ contain only one redex that is a suffix and are therefore monotone. The body of the first predicate rewrite is monotone since $\wedge_n$ and $setExists_1$ are both monotone when the arguments are monotone. (Note that $setExists_1\ p$ is equivalent to $(domCard\ p) \circ (> 0)$.) ◀

It can be shown that the monotone property of standard predicates is preserved under predicate derivation steps for monotone predicate rewrite systems. In other words, given a monotone predicate rewrite system $\rightarrowtail$ and a predicate p that is monotone with respect to $\rightarrowtail$, every predicate q that can be obtained by a predicate derivation step from p is monotone with respect to $\rightarrowtail$.

Example 2.4.16. Consider the predicate derivation given in Example 2.4.6. The predicate rewrite system $\rightarrowtail$ is monotone. The initial predicate top is monotone with respect to the predicate rewrite system. Consequently, we have

$$
\begin{aligned}
& top \\
\Leftarrow\ & listToSet \circ setExists_1\ (\wedge_3\ top\ top\ top) \\
\Leftarrow\ & listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ top)\ top\ top) \\
\Leftarrow\ & listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ top)\ (projLength \circ top)\ top) \\
\Leftarrow\ & listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ top)\ (projLength \circ top)\ (projKind \circ top)) \\
\Leftarrow\ & listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ (= Rectangular)) \\
& \qquad\qquad (projLength \circ top))\ (projKind \circ top)) \\
\Leftarrow\ & listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ (= Rectangular)) \\
& \qquad\qquad (projLength \circ top)\ (projKind \circ (= Open))) \\
\Leftarrow\ & listToSet \circ setExists_1\ (\wedge_3\ (projShape \circ (= Rectangular)) \\
& \qquad\qquad (projLength \circ (= Short))\ (projKind \circ (= Open))).
\end{aligned}
$$

◀

Checking whether a predicate rewrite system is monotone is undecidable in general. However, the condition is a weak one and it is natural for predicate rewrite systems in practical applications to be monotone.

2.4.3 Operations on Predicate Rewrite Systems

In this section, we discuss a few basic operations for manipulating predicate rewrite systems. These will be needed in later chapters. It is worth noting that the notation and materials introduced in this section have not previously appeared elsewhere.

2.4.3.1 Computing the Size of Predicate Classes

A useful piece of information to have when using ALKEMY is the size of the predicate space defined by a predicate rewrite system.

Definition 2.4.17. Given an initial predicate p_0, we say a predicate rewrite system $\rightarrowtail$ is *finite* if the set $S_{\rightarrowtail}(p_0)$ of expected predicates obtainable from p_0 via $\rightarrowtail$ is finite.

We now present an algorithm for calculating $|S_{\rightarrowtail}(p_0)|$, denoted $\#(\rightarrowtail)$ henceforth, given an initial predicate p_0 and a finite predicate rewrite system $\rightarrowtail$. A naïve way to compute $\#(\rightarrowtail)$ is to simply enumerate all the predicates and count them. A more efficient procedure is possible, provided some restrictions are imposed on the form of $\rightarrowtail$. This motivates the next definition.

Definition 2.4.18. A predicate rewrite system $\rightarrowtail$ is in *normal form* if the following hold for each predicate rewrite $r \rightarrowtail s$.

1. r is a transformation of rank 0.
2. For each subterm of s having the form $f\ p_1 \ldots p_k$ ($k \geqslant 0$), if f is symmetric, then $p_i = p_j$ for all $i, j \in \{1, \ldots, k\}$.

Example 2.4.19. Consider the following two predicate rewrite systems.

$$\begin{array}{ll} top \rightarrowtail \vee_2\ (\wedge_2\ top\ top)\ (\wedge_2\ top\ top) & top \rightarrowtail \vee_2\ (\wedge_2\ q\ top)\ (\wedge_2\ q\ top) \\ top \rightarrowtail p_1 & top \rightarrowtail p_1 \\ \ldots & \ldots \\ top \rightarrowtail p_n & top \rightarrowtail p_n \end{array}$$

The predicate rewrite system on the left is in normal form, but the one on the right is not in normal form because the arguments to $\wedge_2$ are not equal. ◀

Condition (1) in Definition 2.4.18 is introduced to achieve efficiency in locating redexes. Without the constraint, we need to check all suffixes; with it, we only have to look at the last transformations. This restriction is not severe. In fact, we have never encountered the need to use anything more complex than a transformation of rank 0 as the head of a predicate rewrite. In any case, it is not hard to normalize a predicate rewrite system to satisfy condition (1); in general, such changes do not affect the value of $\#(\rightarrowtail)$.

Condition (2), which is more restrictive, is introduced for practical reasons. We would like to be able to determine the set of predicate rewrites applicable to a redex using only local type information. Terms of the form $f\ p_1 \ldots p_k$, where f is symmetric but there exists i and j in the range $\{1, \ldots, k\}$ such that $p_i \neq p_j$, can cause difficulty because the determination of applicable predicate rewrites requires information not available locally. As an illustration of the problem, consider a predicate of the form $f\ (p \circ top)\ top$. If f is not symmetric, to compute the set of applicable rewrites, AR, for the second top we need to know only its type. But if f is symmetric, to compute AR requires knowledge of $p \circ top$ and how it is rewritten. This is because we only

want to count regular predicates. In general, normalizing a predicate rewrite system $\rightarrowtail$ to conform to the second condition will change the value of $\#(\rightarrowtail)$. However, this restriction is not serious. As we shall see, given a predicate rewrite system that does not satisfy condition (2), the algorithm returns an upper bound on $\#(\rightarrowtail)$.

The algorithm implements the following recursive formula $size(p, \rightarrowtail)$ for counting the number of predicates obtainable from some standard predicate p via $\rightarrowtail$.

$$size((f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n}), \rightarrowtail) = \prod_{i=1}^{n} expand(f_i\ p_{i,1} \ldots p_{i,k_i}, \rightarrowtail)$$

1) $expand(f, \rightarrowtail) = 1$, if f is not a redex;

2) $expand(f, \rightarrowtail) = 1 + \sum_{f \rightarrowtail s \in AR} size(s, \rightarrowtail)$,
if f is a redex, and AR are the applicable rewrites;

3) $expand(f\ p_1 \ldots p_k, \rightarrowtail) = \binom{size(p_1, \rightarrowtail) + k - 1}{k}$,
if f is symmetric and $p_i = p_j$ for all $i, j \in \{1, \ldots, k\}$;

4) $expand(f\ p_1 \ldots p_k, \rightarrowtail) = \prod_{i=1}^{k} size(p_i, \rightarrowtail)$, otherwise.

The function $expand$ is to be understood as a macro, to be expanded in different ways depending on the form of the input; its use simplifies the presentation of the formula. The correctness of the algorithm relies on the following.

Proposition 2.4.20. *Given a finite, normalized predicate rewrite system $\rightarrowtail$ and a standard predicate $q = (f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$, the function call $size(q, \rightarrowtail)$ returns the size of the set of expected predicates obtainable from q via $\rightarrowtail$.*

Proof. The proof proceeds by induction on n, the number of calls to $size$. For $n = 1$, formula (1) must apply for each $f_i\ p_{i,1} \ldots p_{i,k_i}$ $(1 \leqslant i \leqslant n)$. That means q does not have a redex, and the number of predicates obtainable from q is indeed 1.

Assume now that the statement holds for all computations involving strictly less than k calls to $size$, where $k > 1$. Consider a computation involving k calls to $size$, assuming one exists. Each computation of the form $expand(f_i\ p_{i,1} \ldots p_{i,k_i}, \rightarrowtail)$ calculates the number of predicates that can be obtained from q using only redexes in $f_i\ p_{i,1} \ldots p_{i,k_i}$. The total number of predicates that can be obtained from q is then

$$\prod_{i=1}^{n} expand(f_i\ p_{i,1} \ldots p_{i,k_i}, \rightarrowtail).$$

It is sufficient to show that each individual $expand(f_i\ p_{i,1} \ldots p_{i,k_i}, \rightarrowtail)$ computation is correct. The correctness of cases (1), (2) and (4) follow directly from the inductive hypothesis. For case (3), we want to choose a multiset of k elements from m possible

choices, where m is the number of distinct predicates that can be used to rewrite the redexes. The number of such multi-combinations is given by $\binom{m+k-1}{k}$. We have the desired result by noting that $m = size(p_1, \rightarrowtail)$ by the inductive hypothesis. □

We now examine the implication of using a predicate rewrite system that does not satisfy condition (2) of Definition 2.4.18. Only subterms of the form $f\ p_1 \ldots p_k$ where f is symmetric are affected. For these, we count the number of ways they can be rewritten as if f is non-symmetric. The function $size$ thus gives us an upper bound for $\#(\rightarrowtail)$ in this case.

Example 2.4.21. Consider the predicate rewrite system $\rightarrowtail$ given in Example 2.4.6. Using the algorithm given, we can compute the size of the set of expected predicates obtainable from $top : Train \rightarrow \Omega$ via $\rightarrowtail$ as follows.

$$
\begin{aligned}
&size(top : Train \rightarrow \Omega, \rightarrowtail) \\
&= expand(top, \rightarrowtail) \\
&= 1 + size(listToSet \circ setExists_1\ (\wedge_3\ top\ top\ top), \rightarrowtail) \\
&= 1 + expand(listToSet, \rightarrowtail) \times expand(setExists_1\ (\wedge_3\ top\ top\ top), \rightarrowtail) \\
&= 1 + \binom{size(\wedge_3\ top\ top\ top, \rightarrowtail) + 1 - 1}{1} \\
&= 1 + size(\wedge_3\ top\ top\ top, \rightarrowtail) \\
&= 1 + \binom{size(top, \rightarrowtail) + 3 - 1}{3} \\
&= 10661
\end{aligned}
$$

The last step follows from this calculation.

$$
\begin{aligned}
&size(top : Car \rightarrow \Omega, \rightarrowtail) \\
&= 1 + size(projShape \circ top, \rightarrowtail) + \ldots + size(projLoad \circ top, \rightarrowtail) \\
&= 1 + expand(projShape, \rightarrowtail) \times expand(top, \rightarrowtail) + \ldots \\
&= 1 + 1 \times (1 + size((= Rectangular), \rightarrowtail) + \ldots size((= Ellipsoidal), \rightarrowtail)) + \ldots \\
&= 1 + 7 + 3 + 4 + 3 + 6 + 15 \\
&= 39
\end{aligned}
$$

◀

2.4.3.2 Negating a Predicate Rewrite System

In addition to generating predicates in $S_{\rightarrowtail}$ for a given $\rightarrowtail$, we sometimes need to generate the negation of each predicate in $S_{\rightarrowtail}$ as well. For the purpose of search, it is not sufficient to enumerate $S_{\rightarrowtail}$ and for each $p \in S_{\rightarrowtail}$ generate also $\neg p$ — the implication relationships between the negated predicates would be in the 'wrong' order. A separate predicate rewrite system $\rightarrowtail_{neg}$ is needed. Unfortunately, there is no easy way to generate $\rightarrowtail_{neg}$ by simple manipulation of $\rightarrowtail$. We actually need to look at the search space defined by $\rightarrowtail$.

A naïve way to generate $\rightarrowtail_{neg}$ from $\rightarrowtail$ is to first enumerate the set $S_{\rightarrowtail}$ and then form a predicate rewrite $top \rightarrowtail_{neg} \neg p$ for each $p \in S_{\rightarrowtail}$. But this throws away all the structure in the original predicate rewrite system and with it, associated benefits like compactness and amenability to search. A better way to generate $\rightarrowtail_{neg}$ is to simply invert the search space, and then cut off unnecessary edges. We illustrate this process with an example. Consider the following predicate rewrite system :

$$p_1 \rightarrowtail p_2$$
$$p_1 \rightarrowtail p_3$$
$$p_3 \rightarrowtail p_{31}$$
$$p_3 \rightarrowtail p_{32}.$$

The first step in defining $\rightarrowtail_{neg}$ is to invert the predicate space and then negate the predicates, resulting in the following:

$$top \rightarrowtail_{neg} \neg p_2$$
$$top \rightarrowtail_{neg} \neg p_{31}$$
$$top \rightarrowtail_{neg} \neg p_{32}$$
$$\neg p_{31} \rightarrowtail_{neg} \neg p_3$$
$$\neg p_{32} \rightarrowtail_{neg} \neg p_3$$
$$\neg p_2 \rightarrowtail_{neg} \neg p_1$$
$$\neg p_3 \rightarrowtail_{neg} \neg p_1$$

The search space defined by $\rightarrowtail_{neg}$ as it stands is a graph. We can convert it back into a tree by removing all but one predicate rewrites from each group of predicate rewrites with the same body. Lacking further information, a random choice on which predicate rewrite to keep can be made. Trimming $\rightarrowtail_{neg}$ from above, we get the final form:

$$top \rightarrowtail_{neg} \neg p_2$$
$$top \rightarrowtail_{neg} \neg p_{31}$$
$$top \rightarrowtail_{neg} \neg p_{32}$$
$$\neg p_{31} \rightarrowtail_{neg} \neg p_3$$
$$\neg p_2 \rightarrowtail_{neg} \neg p_1.$$

The predicate rewrite system $\rightarrowtail_{neg}$ is still not very compact, but the search space defined by this $\rightarrowtail_{neg}$ does preserve implication relationships between predicates, which is important for search efficiency.

2.4.3.3 Joining Predicate Rewrite Systems

We will have occasion to talk about the joining of two or more predicate rewrite systems. We use the set notation for this purpose. For example, to join two predicate rewrite systems $\rightarrowtail_1$ and $\rightarrowtail_2$, we write $\rightarrowtail_1 \cup \rightarrowtail_2$. The intended meaning is to get a

new rewrite system $\rightarrowtail\ (= \rightarrowtail_1 \cup \rightarrowtail_2)$ such that $S_{\rightarrowtail} = S_{\rightarrowtail_1} \cup S_{\rightarrowtail_2}$. Depending on the actual rewrite systems, this operation may involve more than a simple syntactic concatenation of $\rightarrowtail_1$ and $\rightarrowtail_2$. What needs to be done in each instance should be clear from the context. In any case, the well-definability of $\rightarrowtail_1 \cup \rightarrowtail_2$ is never in question: we can always define it to be the collection of all predicate rewrites of the form $top \rightarrowtail p$ where $p \in S_{\rightarrowtail_1} \cup S_{\rightarrowtail_2}$.

2.5 Data Types and Transformations

This section contains a listing of some generic transformations for different data types. They are used throughout the text and are collected here for ease of reference. All the material presented in this section appear previously in [130]; they are given here for completeness of presentation.

Sets

Example 2.5.1. We have encountered the transformation $setExists_1$ in Example 2.3.4. More generally, for $n \geqslant 1$, one can define

$$setExists_n : (a \to \Omega) \to \cdots \to (a \to \Omega) \to \{a\} \to \Omega$$

by

$$setExists_n\ p_1 \ldots p_n\ t = \exists x_1. \cdots \exists x_n.\ (p_1\ x_1) \wedge \cdots \wedge (p_n\ x_n) \wedge (x_1 \in t) \wedge \cdots \wedge (x_n \in t) \wedge (x_1 \neq x_2) \wedge \cdots \wedge (x_{n-1} \neq x_n).$$

Note the overlap between $(domCard\ b) \circ (> 0)$ and $(setExists_1\ b)$. Typically, $setExists_n$ is used for small values of n, say, 1, 2 or 3, while $domCard$ is used in conjunction with $(> n)$ for larger values of n. ◀

Multisets

Multisets (also known as bags) are a useful generalization of sets. A straightforward approach to multisets is to regard a multiset as a function of type $\mu \to Nat$, where μ is the type of elements in the multiset and the value of the multiset on some item is its multiplicity, that is, the number of times it occurs in the multiset.

Example 2.5.2. Consider the transformation

$$domMcard : (a \to \Omega) \to (a \to Nat) \to Nat$$
$$domMcard\ b\ t = mcard\ (\lambda x.if\ (b\ x)\ then\ (t\ x)\ else\ 0),$$

where $mcard$ computes the cardinality of a multiset. Thus, for each $b : \mu \to \Omega$,

$$(domMcard\ b) : (\mu \to Nat) \to Nat$$

is a function on multisets whose elements have type μ that computes the cardinality of the submultiset of elements of the argument that satisfy the predicate b.

One can obtain a predicate by composing $(domMcard\ \ b)$ with, say, a predicate $(> N)$, for some $N \geqslant 0$. This gives the predicate $(domMcard\ b) \circ (> N)$, which is true of a multiset iff the cardinality of the submultiset of elements satisfying the predicate b is strictly greater than N. ◀

Example 2.5.3. Multisets also support the analogue of the transformation $setExists_n$. The transformation $msetExists_n$, for $n \geqslant 1$, is defined as follows.

$$msetExists_n : (a \to \Omega) \to \cdots \to (a \to \Omega) \to (a \to Nat) \to \Omega$$
$$msetExists_n\ b_1 \ldots b_n\ t = \exists x_1. \cdots \exists x_n.(b_1\ x_1) \wedge \cdots \wedge (b_n\ x_n) \wedge$$
$$((t\ x_1) > 0)\ \wedge \cdots \wedge\ ((t\ x_n) > 0) \wedge (x_1 \neq x_2) \wedge \cdots \wedge (x_{n-1} \neq x_n).$$

◀

Graphs

To represent graphs, we adopt the standard mathematical definition $G = (V, E)$, where V is the set of vertices and E, the set of edges. Each vertex is uniquely labelled by a number and each edge is represented as an unordered pair of the two vertices it connects. We are thus led to the following type declarations:

$$Label = Nat$$
$$Graph\ \nu\ \varepsilon = \{Label \times \nu\} \times \{(Label \to Nat) \times \varepsilon\}.$$

Here, ν is the type of the information at a vertex, and ε is the type of the information on an edge. To avoid the need for explicit references to the label type, we introduce two type constructors $Vertex$ and $Edge$ such that the type of a vertex is $Vertex\ \nu\ \varepsilon$ and the type of an edge is $Edge\ \nu\ \varepsilon$.

Example 2.5.4. The following are some transformations on graphs. For brevity, we omit the exact definitions, giving instead only high-level descriptions.

$$vertices : Graph\ \nu\ \varepsilon \to \{Vertex\ \nu\ \varepsilon\}$$
$$edges : Graph\ \nu\ \varepsilon \to \{Edge\ \nu\ \varepsilon\}$$
$$vertex : Vertex\ \nu\ \varepsilon \to\ \nu$$
$$edge : Edge\ \nu\ \varepsilon \to\ \varepsilon$$
$$connects : Edge\ \nu\ \varepsilon \to (Vertex\ \nu\ \varepsilon \to Nat)$$
$$(subgraphs\ k) : Graph\ \nu\ \varepsilon \to \{Graph\ \nu\ \varepsilon\}.$$

Here, $vertices$ returns the set of vertices of a graph, $edges$ returns the set of edges of a graph, $vertex$ returns the information at a vertex, $edge$ returns the information on an edge, $connects$ returns the unordered pair of vertices joined by an edge, and

$(subgraphs\ k)$ returns the set of all (connected) subgraphs containing k vertices of a graph. ◀

2.6 Related Work

There is a long tradition of logical methods in machine learning, going back to the works of Plotkin, Michalski, and Vere in the 1970s. Such learning methods are now primarily studied in the field of Inductive Logic Programming (ILP) [144]. [174] contains an interesting historical account of the development of the field.

Driven by research on Prolog, first-order clausal logic has always been the principal knowledge representation language in ILP. The field has grown considerably in the last fifteen years, however, and different new representation formalisms are now being investigated. The representation formalism based on higher-order logic outlined in this chapter is a recent addition to the family of languages now studied in ILP. It was introduced and motivated in [76], [30], [31], and more recently in book form in [130]. The main ideas of the formalism are not completely new, of course. The use of higher-order logic for inductive learning has previously been considered in [146]. The use of functions for learning has also been explored; see [95] and [74]. A detailed comparison of the higher-order approach adopted here and the first-order approach traditionally studied in ILP can be found in Sect. 7.1.

The individual's whole experience is built upon
the plan of his language.
Henri Delacroix

Chapter 3

Classification

Crude classifications and false generalizations
are the curse of organized life.
George Bernard Shaw

3.1 Introduction

Classification is the simplest and best-studied problem in machine learning, fundamental to many other learning tasks of higher complexity. Having a good grip of its underlying concepts is therefore important, and this forms the subject matter of the present chapter.

The main classification learning algorithms are presented in Section 3.2. This is followed by fairly thorough analysis of their behaviour in the remaining sections, organized around three central themes.

The first of these concerns the approximation, or representational, properties of the learning algorithms. This is addressed in Section 3.3, where we describe some natural Alkemic predicate classes, state a few of their properties, and explore relationships between them.

The second, treated in Section 3.4, is about error estimation (or sample complexity) issues. There, we state some generalization bounds suitable for use with ALKEMY and present new techniques for characterizing the complexity of function classes definable using predicate rewrite systems.

The third, discussed in Section 3.5, is connected with optimization issues. Questions there are computational in nature, and are related to the effectiveness and efficiency with which certain search problems can be solved. In particular, we consider what are appropriate optimization criteria for the learning algorithms presented in Section 3.2 and examine their optimality with respect to these goals.

We conclude in Section 3.6 by bringing together results presented in the earlier sections to make some formal learnability statements in the PAC and agnostic PAC learning models.

Throughout the chapter, we will always assume that we are dealing with binary classification problems. We appeal to machine learning reduction techniques [19] in making this decision. In that view of the world, solutions to many different forms

of classification tasks, including multi-class, cost-sensitive, and importance weighted classification problems can be *reduced* to operations involving subroutine calls to a binary classification algorithm. The performance of these meta-algorithms can in turn be bounded in terms of the performance of the underlying binary classification algorithm. Thus, in a rather strong sense, it is quite sufficient to understand binary classification well.

3.2 Learning Algorithms

Induction of decision trees from data has had a long history of development, dating back to the 1960's. In that time, many algorithms have been proposed and tried out. Two of the most established among these are implemented in ALKEMY, and we describe them in this section.

The first is a variant of the standard top-down induction algorithm. We describe it in two stages. In the first stage, presented in §3.2.1, we describe the decision-stump learning algorithm, which is a special case of the general decision-tree learning algorithm. The aim is to flesh out some of the central concepts in ALKEMY in a clean-room environment. The top-down induction algorithm is then presented in §3.2.2 as a recursive algorithm around the basic stump learning procedure.

The second, discussed in §3.2.3, is a variant of Rivest's covering algorithm [169] for learning decision lists. We built on ideas presented in [184] in the design of that algorithm.

3.2.1 Learning Stumps

The decision-stump learning algorithm takes as input

1. a training set $z \in (X \times \{0,1\})^m$ of (arbitrary) size m,
2. a predicate rewrite system $\rightarrowtail$ defining predicates over X,

and produces as output a hypothesis $h : X \to \{0,1\}$ of the form

$$h(x) = \text{if } (p\ x) \text{ then } c_1 \text{ else } c_2,$$

where $p \in S_{\rightarrowtail}$, and $c_1, c_2 \in \{0,1\}$. Such rules are called decision stumps.

The aim of learning is to find in $S_{\rightarrowtail}$ the predicate that, with proper labellings, achieves the highest accuracy on the training set. This notion is made precise in §3.2.1.1. In §3.2.1.2, we exploit structures in $S_{\rightarrowtail}$ to give a search space pruning result. The learning algorithm is given in §3.2.1.3; it forms the basis of the decision-tree learning algorithm described in §3.2.2.

3.2.1.1 Predicate Selection

We now give some basic definitions, following [130]. Assume there are two classes. Let $\mathcal{E}$ be a (non-empty) set of examples, N the number of examples in $\mathcal{E}$, n_i the number of examples in $\mathcal{E}$ in the ith class, and $p_i = n_i/N$, for $i = 1, 2$.

Definition 3.2.1. We define the *majority class* of $\mathcal{E}$, denoted $maj(\mathcal{E})$, to be the class to which the greatest number of examples in $\mathcal{E}$ belong. (Ties are broken arbitrarily.)

Definition 3.2.2. We define the *accuracy*, $A_{\mathcal{E}}$, of a set $\mathcal{E}$ of examples by $A_{\mathcal{E}} = p_M$, where M is the index of the majority class of $\mathcal{E}$.

The accuracy is the fraction of examples which are correctly classified on the basis that the majority class gives the classification. $A_{\mathcal{E}}$ lies in the range $[1/2, 1]$, where larger values correspond intuitively to purer sets of examples.

If $\mathcal{E}$ is a set of examples, then a predicate p induces a partition $(\mathcal{E}_1, \mathcal{E}_2)$ of $\mathcal{E}$, where $\mathcal{E}_1$ is the set of examples that satisfy p and $\mathcal{E}_2$ is the set of examples that do not satisfy p. We now define the accuracy of a partition of a set of examples.

Definition 3.2.3. Let $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ be a partition of a set $\mathcal{E}$ of examples. We define the *accuracy*, $A_{\mathcal{P}}$, of the partition $\mathcal{P}$ by

$$A_{\mathcal{P}} = \frac{|\mathcal{E}_1|}{|\mathcal{E}|} A_{\mathcal{E}_1} + \frac{|\mathcal{E}_2|}{|\mathcal{E}|} A_{\mathcal{E}_2}.$$

Thus $A_{\mathcal{P}}$ is the weighted average of the accuracies of the individual sets of examples in the partition. It is equal to the total number of examples correctly classified in each of $\mathcal{E}_1$ and $\mathcal{E}_2$ divided by $|\mathcal{E}|$. $A_{\mathcal{P}}$ lies in the range $[1/2, 1]$, where larger values correspond to partitions that classify more accurately.

Given a set of examples, the goal of learning is to seek a predicate in the search space that has the highest accuracy. In the case of ties, the predicate with the lowest entropy, defined below, is preferred. Similar measures like the Gini index can be used. We will have more to say on this tie-breaking mechanism in §3.2.2.

Definition 3.2.4. We define the *entropy*, $EN_{\mathcal{E}}$, of a set $\mathcal{E}$ of examples by

$$EN_{\mathcal{E}} = -p_1 \log(p_1) - p_2 \log(p_2),$$

where p_1 and p_2 are the fractions of examples in the two classes.

Definition 3.2.5. Let $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ be a partition of a set $\mathcal{E}$ of examples. We define the *entropy*, $EN_{\mathcal{P}}$, of $\mathcal{P}$ by

$$EN_{\mathcal{P}} = \frac{|\mathcal{E}_1|}{|\mathcal{E}|} EN_{\mathcal{E}_1} + \frac{|\mathcal{E}_2|}{|\mathcal{E}|} EN_{\mathcal{E}_2}.$$

Useful information about accuracy and entropy, including their relationships to other commonly used predicate selection functions, can be found in [75] and [80].

3.2.1.2 Predicate Pruning

Next we introduce a measure for partitions that is crucial for pruning the search space of predicates. The result in this section is due to [130]. It works in the more general case of $n > 2$ classes, and is formulated as such.

A predicate derivation step takes a predicate p and constructs a new predicate p' by strengthening p. Thus the new predicate p' implies p and the partition $(\mathcal{E}_1', \mathcal{E}_2')$ of $\mathcal{E}$ induced by p' has the property that $\mathcal{E}_1' \subseteq \mathcal{E}_1$, where $\mathcal{E}_1 \subseteq \mathcal{E}$ is the set of examples that satisfy p. These considerations lead to the following definition.

Definition 3.2.6. Let $\mathcal{E}$ be a set of examples and $(\mathcal{E}_1, \mathcal{E}_2)$ a partition of $\mathcal{E}$. We say a partition $(\mathcal{E}_1', \mathcal{E}_2')$ of $\mathcal{E}$ is a *refinement* of $(\mathcal{E}_1, \mathcal{E}_2)$ if $\mathcal{E}_1' \subseteq \mathcal{E}_1$.

We now introduce the important measure of classification refinement bound.

Definition 3.2.7. Let $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ be a partition of a set $\mathcal{E}$ of examples, where n_i is the number of examples in $\mathcal{E}$ in the ith class and $n_{j,i}$ is the number of examples in $\mathcal{E}_j$ in the ith class, for $j = 1, 2$ and $i = 1, 2$. We define the *classification refinement bound*, $B_{\mathcal{P}}$, of the partition $\mathcal{P}$ by

$$B_{\mathcal{P}} = (\max_i \{n_i + \max_{k \neq i} n_{1,k}\})/|\mathcal{E}|.$$

We will show below that $B_{\mathcal{P}}$ is an upper bound for the accuracy of any refinement of $\mathcal{P}$. The intuitive idea behind the definition of $B_{\mathcal{P}}$ is that the refinement of $\mathcal{P}$ having the greatest accuracy can be obtained by moving all examples in one class from $\mathcal{E}_1$ across to $\mathcal{E}_2$. Here is an example to illustrate the concept of classification refinement bound. In the following, we denote by (n_1, n_2) a set of examples with n_i examples from the ith class, for $i = 1, 2$.

Example 3.2.8. Let $\mathcal{E}$ = $(6, 9)$ and suppose $\mathcal{P} = ((2, 1), (4, 8))$. Then $A_{\mathcal{P}} = 10/15$ and $B_{\mathcal{P}} = 11/15$. If $\mathcal{Q} = ((0, 9), (6, 0))$, then $A_{\mathcal{Q}} = B_{\mathcal{Q}} = 15/15$. ◀

Proposition 3.2.9 ([130]). *Let $\mathcal{E}$ be a set of examples and $\mathcal{P}$ a partition of $\mathcal{E}$. If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $A_{\mathcal{P}'} \leq B_{\mathcal{P}}$. In particular, $A_{\mathcal{P}} \leq B_{\mathcal{P}}$.*

Proof. Let $\mathcal{P}$ be $(\mathcal{E}_1, \mathcal{E}_2)$ and $\mathcal{P}'$ be $(\mathcal{E}_1', \mathcal{E}_2')$, where $\mathcal{E}_1' \subseteq \mathcal{E}_1$. Let $n_{1,i}$ be the number of examples in $\mathcal{E}_1$ in the ith class and M_2' the index of the majority class of $\mathcal{E}_2'$. Consider the partition $\mathcal{Q}$ obtained from $\mathcal{P}'$ by moving across any remaining examples from class M_2' in $\mathcal{E}_1'$ to $\mathcal{E}_2'$. Clearly $A_{\mathcal{P}'} \leq A_{\mathcal{Q}}$, since the examples so moved will be correctly classified after the move. But $A_{\mathcal{Q}} \leq B_{\mathcal{P}}$, since $A_{\mathcal{Q}} \leq (\{n_{M_2'} + \max_{k \neq M_2'} n_{1,k}\})/|\mathcal{E}|$ and $B_{\mathcal{P}} = (\max_i \{n_i + \max_{k \neq i} n_{1,k}\})/|\mathcal{E}|$. Hence the result. □

Proposition 3.2.9 is used by the learning system to prune the search space in its search for the best partition of the examples. During this search, the system records the best partition $\mathcal{P}$ found so far and its associated accuracy $A_{\mathcal{P}}$. When investigating a new partition $\mathcal{Q}$, the quantity $B_{\mathcal{Q}}$ is calculated. According to the proposition, if $B_{\mathcal{Q}} \leq A_{\mathcal{P}}$, then the partition $\mathcal{Q}$ and all its refinements can be safely pruned. Here is an example to illustrate how this works.

Example 3.2.10. Let the set of examples be $(6, 9)$. Suppose the best partition found so far is $\mathcal{P} = ((6, 3), (0, 6))$, which has accuracy 12/15. Suppose that later on in the search the partition $\mathcal{Q} = ((2, 4), (4, 5))$ is being investigated. Note that $B_{\mathcal{Q}} = 11/15$. Since $B_{\mathcal{Q}} < A_{\mathcal{P}}$, the proposition shows that $\mathcal{Q}$ and its refinements can be pruned.

On the other hand, consider the partition $\mathcal{R} = ((6,5),(0,4))$, for which $B_{\mathcal{R}} = 15/15$. Thus $\mathcal{R}$ has refinements, which could be found by the system, whose accuracies exceed that of $\mathcal{P}$. (The refinements actually investigated depend upon the hypothesis language, of course.) Thus $\mathcal{R}$ should not be pruned. ◀

3.2.1.3 Searching

We now move on to the search algorithm. We have seen in §2.4.1 how the set of predicates defined by a predicate rewrite system can be enumerated. Two algorithms are given for that purpose. What is needed for searching are variations of those algorithms that instead return a single predicate: the one deemed best by the predicate selection rule. Construction of the output decision stump is straightforward given the selected predicate.

Figure 3.1 gives the search version of Algorithm II (Figure 2.3); we will refer to it as the LR search algorithm in the following. Algorithm I (Figure 2.2) can be similarly altered, and we will refer to it as the SeenSet search algorithm.

The accuracy, classification refinement bound, and entropy of a predicate referred to in Figure 3.1 are defined with respect to the partition of examples induced.

Definition 3.2.11. Let $\mathcal{E}$ be a set of examples and p a predicate. We define the *accuracy*, A_p, of p by $A_p = A_{\mathcal{P}}$, where $\mathcal{P}$ is the partition of $\mathcal{E}$ induced by p. Likewise, we define the *classification refinement bound*, B_p, of p by $B_p = B_{\mathcal{P}}$ and the *entropy*, EN_p, of p by $EN_p = EN_{\mathcal{P}}$.

There are three inputs to the algorithm: the set of training examples; a predicate rewrite system $\rightarrowtail$; and a prune parameter P. We assume $\rightarrowtail$ is monotone and satisfies all the conditions necessary for ensuring uniqueness and completeness of predicate derivations. (See [130, §4.6] for more details.) We also assume $S_{\rightarrowtail}$ is finite.

The prune parameter P is a percentage; it causes predicates with classification refinement bounds lower than P to be pruned and thus removes predicates that do not have the potential for achieving high accuracy. In the default mode, the parameter is initially set at 0%. As better accuracies are obtained during search, the value of the parameter is updated. This kind of pruning is safe. However, for large search spaces, it is common to set a high initial value for P.

The open list is decreasingly ordered by the refinement bounds of predicates. (Predicates with the same refinement bound are decreasingly ordered by accuracy.) The refinement bound plays a crucial role in directing the search towards promising predicates, those which have the potential for being strengthened to produce splits with high accuracy. In this regard, the function *Insert* takes a predicate and the open list as arguments and returns a new open list with the predicate inserted in the appropriate place.

The function *Redexes* takes a standard predicate p and returns all the redexes in p. Clearly, a predicate with no redexes in it should not be inserted into the open list.

The third line in the **while** loop is a simple optimization. Since p could have been inserted into the open list early on in search while P was still low and has become

function $Predicate(\mathcal{E}, \rightarrowtail, P)$ **returns** a predicate;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, prune parameter;

$openList := [top]$;
$predicate := top$;
$accuracy := A_{\mathcal{E}}$;
$entropy := EN_{\mathcal{E}}$;
while $openList \neq []$ **do**
$p := head(openList)$;
$openList := tail(openList)$;
if $B_p < accuracy$ **then continue**;
for each LR redex r via $r \rightarrowtail b$, for some b, in p **do**
$q := p[r/b]$;
if q is regular **then**
if $A_q > accuracy$ **then**
$predicate := q$;
$accuracy := A_q$;
$entropy := EN_q$;
if $A_q > P$ **then** $P := A_q$;
if $A_q = accuracy \ \wedge \ EN_q < entropy$ **then**
$predicate := q$;
$entropy := EN_q$;
if $B_q \geqslant P \ \wedge \ B_q > A_q \ \wedge \ Redexes(q) \neq \emptyset$ **then**
$openList := Insert(q, openList)$;
return $predicate$;

Figure 3.1: Algorithm for finding a predicate that maximizes accuracy

irrelevant in the mean time, checking B_p against $accuracy$ at this point can avoid unnecessary computation, especially towards the end of the open list.

Incomplete Search Algorithms

Incomplete searches are useful when the predicate space is too large for an exhaustive search. ALKEMY implements two incomplete search strategies. They can be used with either forms of predicate enumeration, but experience indicates that, in general, they work better in conjunction with the SeenSet algorithm. This is because under this scheme there are (usually) multiple ways to arrive at the same predicate. The chance of finding good predicates quickly is thus higher.

The first of these incomplete search algorithms is beam search, where we keep only a small number of the most interesting predicates for strengthening at any one stage. Two open lists, one for the current iteration, and one for the next, are required for this to work.

The second is a form of resource-bounded greedy depth-first search algorithm. A parameter $cutout$ can be set such that if successively $cutout$ predicates are investigated without finding one strictly better than the current best predicate, then the algorithm terminates and returns the best predicate found so far. Every time a new best predicate is found, the $cutout$ parameter is reset to its initial value.

Besides these two, it is easy to incorporate other search strategies. For example, if accuracy is an issue and there is no harsh constraints on search time, one can implement a strategy whereby a minimum accuracy is specified by the user, and the system does not terminate until a predicate meeting that target accuracy is found. For another example, consider real-time applications where there is a hard limit on response time. In this case, the learner should try to do the best it can within the specified time.

3.2.2 Learning Trees

Having laid the groundwork in the previous section, we now proceed with the top-down induction algorithm. Let X be the set of individuals and $\rightarrowtail$ a predicate rewrite system defining predicates over X. A *decision tree* on X with node functions in $S_{\rightarrowtail}$ is a binary tree where each non-terminal node is labelled with a predicate in $S_{\rightarrowtail}$, and each terminal node is labelled with an element from $\{0, 1\}$. A tree defines a function $f : X \rightarrow \{0, 1\}$ in the usual way. To find the value of $f(x)$ for some arbitrary $x \in X$, we push it down the tree until a leaf node l is reached; the label of l then gives the value of $f(x)$. At each non-terminal node, we push x down the left subtree if $(p\ x) = 1$; otherwise we push it down the right subtree.

The top-down induction algorithm is given in Figures 3.2 and 3.3. In our implementation, the cost complexity pruning algorithm of CART [32] is used for tree post-pruning. Any other post-pruning technique can be employed.

The use of accuracy for stump learning makes perfect sense, but its use here for tree induction goes againsts conventional wisdom. Two criticisms stated in [32, §4.1] against accuracy are as follows.

function $Learn(\mathcal{E}, \rightarrowtail, P)$ **returns** a decision tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, $prune$ parameter;

$tree$:= $BuildTree(\mathcal{E}, \rightarrowtail, P)$
label each leaf node of $tree$ by its majority class;
$tree$:= $Postprune(tree)$;
return $tree$;

Figure 3.2: Decision-tree learning algorithm

function $BuildTree(\mathcal{E}, \rightarrowtail, P)$ **returns** a decision tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, $prune$ parameter;

$tree$:= single node (with examples $\mathcal{E}$);
p := $Predicate(\mathcal{E}, \rightarrowtail, P)$;
if $A_p = A_{\mathcal{E}} \ \wedge \ EN_p = EN_{\mathcal{E}}$ **then return** $tree$;
$tree.predicate$:= p;
$\mathcal{E}_+$:= $\{ (x, y) \in \mathcal{E} \ : \ (p \ x) \}$;
$\mathcal{E}_-$:= $\{ (x, y) \in \mathcal{E} \ : \ \neg(p \ x) \}$;
$tree.left$:= $BuildTree(\mathcal{E}_+, \rightarrowtail, P)$;
$tree.right$:= $BuildTree(\mathcal{E}_-, \rightarrowtail, P)$;
return $tree$;

Figure 3.3: Tree building algorithm

Dataset	$\lvert\mathcal{S}\rvert$	Searched	
Mutagenesis	535	108	(20.19%)
East West	2073	892	(43.03%)
Chemicals	56,837	2821	(4.96%)
Musk-1	1,679,615,641	2,914,727	(0.17%)

Table 3.1: Efficiency of the predicate pruning mechanism

Criticism 1. The use of accuracy can result in premature termination of tree growth. This is because tree nodes that are relatively pure, with examples coming from one predominant class, often cannot be split with a strict increase in accuracy.

Criticism 2. The accuracy heuristic does not take future growth into account in choosing the current best split. We use the example in [32, §4.1] to illustrate this point. Consider a set $\mathcal{E} = (400, 400)$ of 800 examples. Which of the following two is the better partition of $\mathcal{E}$?

$$\mathcal{P}_1 = ((300, 100), (100, 300)); \text{ or}$$
$$\mathcal{P}_2 = ((200, 400), (200, 0)).$$

$\mathcal{P}_2$ is intuitively the more appealing of the two, with potential to result in a smaller tree. But accuracy can't differentiate between them.

Criticisms 1 and 2 notwithstanding, in the context of ALKEMY, as shown in §3.2.1.2, the use of accuracy admits a simple and effective predicate pruning mechanism that warrants its reconsideration here. The pruning mechanism described has been shown to work well in many different applications. Table 3.1 gives an indication of its effectiveness. In it, we list four problems taken from [130, §6.2]. For each, we give the size of the predicate search space $|\mathcal{S}|$ and the number of predicates actually tested in a complete search of the predicate space aided by pruning. The percentage of the predicate space searched is also given.

It is not clear whether more commonly used functions like entropy admit similar (efficiently computable) pruning mechanisms. (See the remark at the end of this section.) Assuming not, then in the context of ALKEMY we can expect accuracy to be a better predicate selection function compared to other more commonly used functions. To backup this claim, we examine two cases, the first when an exhaustive search of the predicate space is computationally feasible, and the second when it is not.

In the first case, with the benefit of pruning, computing the most accurate predicate can be expected to be a lot cheaper than computing the one with, say, the lowest entropy. Assuming accuracy and entropy both yield reasonably accurate decision trees and that their relative performances are not too far apart – we will come back to examine this shortly – then adopting accuracy as the predicate selection function is the better strategy, especially if time efficiency is an issue.

In the second case, we must resort to incomplete searches. Given the same amount of time, we can expect the predicate chosen using, say, entropy in the absence of a pruning mechanism to be, in all likelihood, worse than the predicate chosen using accuracy in the presence of pruning simply because a smaller percentage of the predicate space is actually searched. Further, a predicate with high accuracy would quite naturally have low entropy.

To advocate the use of accuracy as a viable predicate selection function, we need to address the two criticisms stated earlier, both of which are valid arguments. Criticism 1 has a theoretical basis, as shown in [100] (see also §3.5.2). Criticism 2, being an intuitive argument, is weaker but persuasive nonetheless. The solution adopted here is simple: we employ accuracy as the main predicate selection function and use entropy to break ties between equally accurate predicates. We name this scheme Acc^*. Criticism 1 is thus addressed because in the case where no predicate in the search space can achieve a strict increase in accuracy, splits can still be made in accordance with entropy, which we know behaves well. Criticism 2 is also resolved. In the example given, Acc^* will pick $\mathcal{P}_2$ over $\mathcal{P}_1$, as desired.

One final question remains: Is a concave function like entropy, as a predicate selection function, always going to outperform Acc^*? We investigate this empirically. Tables 3.2 and 3.3 show the accuracies (estimated using 10-fold cross validations) obtained using two tree-growing methods on eight datasets taken from the UCI repository. The first algorithm uses Acc^*; the second, entropy. Tree post-pruning is done using the cost-complexity pruning method of CART [32]. The first set of results, shown in Table 3.2, gives the accuracies of the induced classifiers in the absence of tree post-pruning. The second set, shown in Table 3.3, shows the benefits of tree post-pruning. A ✓ is shown if the entropy-based algorithm is significantly more accurate; a $\times$ is shown if the Acc^*-based algorithm is better.

It seems safe to conclude that the performance of Acc^* is comparable to most other predicate selection functions for the following reasons. The experiment above certainly suggests that Acc^* is comparable to entropy. We know from experience that entropy and the Gini index have more-or-less similar behaviour. To top it off, it is shown in [141] and [35] that the Gini index is as good as any other known predicate selection function for the purpose of tree induction. In fact, the general agreement from [141] and [35] is that the exact predicate selection function used doesn't really matter, and it is tree post-pruning that holds the key to the final performance.

To verify the usefulness of the modification to accuracy, we also pitted Acc^* against (plain) accuracy on the eight datasets. The results are shown in Table 3.4. The experiment shows that Acc^* performs at least as well as, and usually better than, accuracy in all except one dataset after tree post-pruning.

Remark. Whether or not commonly used functions like entropy admit efficient pruning mechanisms remains an unresolved question in this thesis. It seems that the concavity of these functions is important. This is the property exploited in a similar work on efficient association rule mining reported in [143]. (See also [79].) The general technique employed in [143] may well be applicable to the solution of our problem.

Dataset	Acc^*	Ent	$Ent > Acc^*$
Audiology	0.735	0.765	✓
Lenses	0.833	0.833	
Mushroom	1.000	1.000	
Votes	0.947	0.942	
Monks-1	0.894	0.886	
Monks-2	0.692	0.692	
Monks-3	0.884	0.876	
Mutagenesis	0.820	0.840	

Table 3.2: Acc^* vs Entropy (w/o tree post-pruning)

Dataset	Acc^*	Ent	$Ent > Acc^*$
Audiology	0.710	0.755	✓
Lenses	0.850	0.850	
Mushroom	0.999	0.999	
Votes	0.959	0.956	
Monks-1	0.920	0.887	×
Monks-2	0.634	0.638	
Monks-3	0.935	0.935	
Mutagenesis	0.820	0.809	

Table 3.3: Acc^* vs Entropy (with tree post-pruning)

Dataset	Acc^*	Acc	$Acc^* > Acc$
Audiology	0.710	0.725	
Lenses	0.850	0.683	✓
Mushroom	0.999	0.999	
Votes	0.959	0.956	
Monks-1	0.920	0.792	✓
Monks-2	0.634	0.603	✓
Monks-3	0.935	0.935	
Mutagenesis	0.820	0.804	

Table 3.4: Acc^* vs Accuracy (with tree post-pruning)

3.2.3 Learning Lists

In addition to the top-down induction algorithm, ALKEMY implements an algorithm that learns decision lists. A *decision list* is a kind of decision tree where splits only ever occur at the false branches of decision nodes. More formally, let X be the set of individuals and $\rightarrowtail$ a predicate rewrite system defining predicates over X. A decision list L on X with node functions in $S_{\rightarrowtail}$ is a list of pairs

$$L = (p_1, v_1), (p_2, v_2), \ldots, (p_r, v_r)$$

where $p_j \in S_{\rightarrowtail}$ and $v_j \in \{0, 1\}$ for $j \in \{1, \ldots, r\}$. If p_r is *top*, then we say L is a *complete decision list*. Assuming L is complete, it defines a function as follows: for any $x \in X$, $L(x)$ is defined to be v_j, where j is the least index such that $(p_j\ x) = 1$. For complete decision lists, we call the last node (top, v_r) the *default node*, and v_r the *default value*.

We introduce some notation before presenting the main algorithm.

Definition 3.2.12. Let p be a predicate, and $\mathcal{E}$ a set of examples. We define the coverage of p over $\mathcal{E}$, $cov(p, \mathcal{E})$, by

$$cov(p, \mathcal{E}) = \{(x, y) \in \mathcal{E} : (p\ x) = 1\}.$$

Definition 3.2.13. A set $\mathcal{E}$ of examples is *pure* if $\forall (x_1, y_1), (x_2, y_2) \in \mathcal{E},\ y_1 = y_2$.

An empty set is vacuously pure.

Definition 3.2.14. Let $L_1 = (p_1, v_1), \ldots, (p_m, v_m)$ and $L_2 = (q_1, l_1), \ldots, (q_n, l_n)$ be two decision lists. We define the *concatenation* of L_1 and L_2, denoted $L_1 : L_2$, by

$$L_1 : L_2 = (p_1, v_1), \ldots, (p_m, v_m), (q_1, l_1), \ldots, (q_n, l_n).$$

The standard separate-and-conquer algorithm for learning decision lists was introduced by [169] and shown to work in the PAC model [193]. As is standard in PAC analysis, two fairly strong assumptions were made in the design of that algorithm:

1. the target function can be represented as a decision list of the given boolean features; and
2. the training sample is free of noise.

We consider here the problem of learning decision lists when these assumptions do not hold, a common scenario in practical applications.

Rivest's algorithm works by iteratively picking out and removing pure subsets from the training examples until a list that covers the whole training set is obtained or until no such pure subsets can be found. In the presence of noise, seeking only pure subsets at every iteration probably wouldn't work very well; for example, the scheme would rule out a predicate that covers 100 positive examples and 1 negative example, accepting instead predicates that cover singleton sets.

To overcome this problem, we can modify the covering algorithm to allow for some impurity in the nodes. A natural extension suggested in [184] is adopted here.

Let $K \in \mathbb{R}^+$ be a cost parameter. Given training examples $\mathcal{E} = P \cup N$, where P is the set of positive examples and N the negative examples, we define the utility of a predicate p with respect to $\mathcal{E}$ by

$$U_p(\mathcal{E}, K) = \max\{|cov(p, P)|, |cov(p, N)|\} - K \min\{|cov(p, P)|, |cov(p, N)|\}.$$

The algorithm works by picking the predicate with the highest utility in the search space at each step, with ties broken in some arbitrary way. The algorithm is given in Figure 3.4. We can recover Rivest's algorithm by setting K to ∞.

function $Cover(\mathcal{E}, \rightarrowtail, K)$ **returns** a decision list;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
K, misclassification cost;

D := empty decision list;
$R := \mathcal{E}$;
while true **do**
$C := \{q \in S_{\rightarrowtail} : cov(q, R) \neq \emptyset \ \wedge \ cov(q, R) \neq R \ \wedge \ U_q(R, K) < \infty\}$;
if $C = \emptyset$ **then return** $D : (top, maj(R))$;
$p := \arg\max_{q \in C}\{U_q(R, K)\}$;
$D := D : (p, maj(cov(p, R)))$;
$R := R \setminus cov(p, R)$;

Figure 3.4: Decision-list learning algorithm

A few notes on the candidate set C. Not every predicate in $S_{\rightarrowtail}$ can be considered for extending D. Predicates that cover no examples are clearly unacceptable. We also do not admit predicates that cover the whole set R; this is to avoid having to make a decision on the label of the (empty) default node if such a predicate is chosen. The third condition $U_q(R, K) < \infty$ caters for the special case when $K = \infty$. Besides these conditions, it is also common to rule out predicates that do not cover some (user-specified) minimum number of examples.

The termination condition for $Cover$ is actually not easy to get right. Some versions of the covering algorithm will stop as soon as R becomes pure. There is, however, a problem with this approach: R may never become pure while remaining non-empty. Indeed, some existing algorithms overlook this problem and can potentially run into infinite loops and/or end up deciding on a default value on the basis of an empty set. The solution adopted here is to continue splitting as long as non-empty, strict subsets of R can still be picked out. In the case when R does become (non-vacuously) pure, the output of $Cover$ would be functionally equivalent to the output of a similar algorithm that stops when R is pure, since the additional nodes will all map to the same default value. Proper termination for $Cover$ is guaranteed by this

next result.

Proposition 3.2.15. *Given a non-empty finite training set $\mathcal{E}$, a predicate rewrite system $\rightarrowtail$, and a positive K, Cover will terminate after a finite number of steps with a non-empty R.*

Proof. We use the following loop invariant: If $R \neq \emptyset \wedge C \neq \emptyset$, then R becomes strictly smaller but non-empty after every iteration. Clearly, R is non-empty to begin with. Further, C cannot be non-empty forever since $\mathcal{E}$ is finite. □

There are a few possible variations to $Cover$. We can place a (user-specified) limit on the maximum number of iterations to allow tradeoffs between error and complexity. Another form of early-stopping is to impose a minimum purity level for splits to occur. Obviously, one can also have different misclassification cost parameters for positive and negative examples.

We now address the problem of finding the predicate with the highest utility in the candidate set. Search versions of either Algorithm I or II in §2.4.1 can be used for this purpose. The definition of U_p admits a simple way to do predicate pruning.

Definition 3.2.16. Given training examples $\mathcal{E}$ and a predicate rewrite system $\rightarrowtail$, we define the *utility refinement bound* of a predicate $p \in S_{\rightarrowtail}$ with respect to $\mathcal{E}$ and $\rightarrowtail$ to be the largest utility that can be achieved by a refinement of p.

Proposition 3.2.17. *Given a training set $\mathcal{E} = P \cup N$ and a predicate rewrite system $\rightarrowtail$, the utility refinement bound of a predicate $p \in S_{\rightarrowtail}$ is given by $\max\{|cov(p, P)|, |cov(p, N)|\}$.*

Proof. Straightforward. □

Pruning works as follows: Given training examples S and a rewrite system $\rightarrowtail$, for each predicate $p \in S_{\rightarrowtail}$, if the utility refinement bound of p is not greater than the utility of the best predicate found so far, then prune.

Properties of $Cover$ are investigated in §3.5.3. Rivest shows that the algorithm (when $K = \infty$) will always return a decision list that achieves 100% accuracy on the training examples assuming one exists. We will give a slightly more general result.

Other Algorithms It is worth pointing out that there is a bottom-up approach to learning decision lists as well. In [142] and [38], the authors give algorithms that construct decision lists in reverse order: the more general cases at the end of the lists are learned first, and then exceptions to those rules are attached to the front of the lists. Probabilistic decision lists [87], popular in natural language processing, can also be learned this way.

3.2.4 Others

We have concentrated on binary classification problems (in the batch learning setting) up to this point. We end the section with brief remarks on algorithms and techniques that can be used to extend ALKEMY to handle other kinds of learning tasks. Some of these are implemented in the system.

3.2.4.1 Multi-Class Problems

The tree- and list-learning algorithms generalize easily to multi-class problems. One can also resort to reduction techniques to handle multi-class problems. For details, see, for example, [19], [62] and [168].

3.2.4.2 Estimating Class Probabilities

In some applications, it is desirable to know the confidence associated with a prediction. Various schemes exists for extracting conditional class membership probabilities from decision tree models; see, for example, [32, §4.6], [36], [136] and [121].

3.2.4.3 Boosting

It has been shown in many independent studies that boosting algorithms work well in conjunction with decision trees; see, for example, [163] and [67]. ALKEMY implements a version of the AdaBoost algorithm [78] called AdaBoost.M1 given in [91]. There are experimental evidence showing that it works well, but more studies are required.

The problem with using AdaBoost is that we quickly lose comprehensibility of the final hypothesis as the number of boosting iterations increases. The alternating decision tree algorithm proposed in [77] and [138] seems to be able to achieve a better balance between accuracy and comprehensibility, and is worthy of more investigation.

3.2.4.4 On-line Learning of Alkemic Trees

ALKEMY also has an on-line learning mode. The algorithms and their analyses are reported in full in Chapter 5, we give only a summary here. The basic idea is that we maintain a fixed window of a certain size. As each new example is encountered, it is inserted into the current tree using a function called $AddExample$. If the window is full, the oldest example is removed from the tree using a function called $RemoveExample$. The two functions have the following property. They employ some sufficient optimality-testing conditions to check whether parts of the tree have become potentially sub-optimal according to the tree-building measures as a result of the update, and mark as dirty all those nodes that needs to be re-examined. There is a function $Retrain$ that can be used to rebuild all the dirty subtrees. The function $Retrain$ has the property that the tree computed is exactly identical to the tree that would be produced by the batch algorithm using the examples in the current window. For that reason, we say the on-line algorithm is *lossless* with respect to the batch algorithm.

3.3 Function Classes and Their Relationships

Suppose we are given a predicate rewrite system $\rightarrowtail$ and suspect that the true concept is a boolean combination of some predicates in $S_{\rightarrowtail}$. How do we define a suitable hypothesis space? There are three ways to do it.

1. One can enrich $\rightarrowtail$ with predicate rewrites specifying different possible combinations of predicates and learn a decision stump.
2. Alternatively, one can stick with $\rightarrowtail$ and learn a decision tree (or list). A decision tree is nothing but a boolean function of the predicates appearing in it. The space of decision trees we search is thus equivalent to a certain class of boolean combinations of predicates in $S_{\rightarrowtail}$.
3. Or one can do both, enrich $\rightarrowtail$ *and* learn a tree (or list).

What is the best way forward? What are the tradeoffs involved?

To get a handle on these questions, we need to understand the different predicate classes that can be defined using predicate rewrite systems in combination with learning algorithms, and relationships between them. This is the subject matter of the section. Information given here provides guidance on how one might go about selecting algorithms and crafting predicate rewrite systems in applications.

3.3.1 Basic Setup

We assume all predicate rewrite systems are finite.

Definition 3.3.1. Let X be a set of individuals and $S = \{p_1, p_2, \ldots, p_n\}$ a finite set of predicates over X. We define $B(X, S)$ by

$$B(X, S) = \{S_{|x} : x \in X\}$$

where $S_{|x} = ((p_1\ x), (p_2\ x), \ldots, (p_n\ x))$.

Definition 3.3.2. A *basic hypothesis language* is a pair $(X, \rightarrowtail)$ where X is a set of individuals and $\rightarrowtail$ is a predicate rewrite system defining predicates over X.

Given a basic hypothesis language $(X, \rightarrowtail)$, $B(X, S_{\rightarrowtail})$ is in general a strict subset of $\{0,1\}^{\#(\rightarrowtail)}$. This is (partly) because there are implication relationships between predicates in $S_{\rightarrowtail}$, and these restrict the values related predicates can take. In particular, if two predicates p and q in $S_{\rightarrowtail}$ are such that p logically implies q, then they must obey the constraint $\forall x \in X, (p\ x) \leqslant (q\ x)$.

Intuitively, the $\rightarrowtail$ in a basic hypothesis language $(X, \rightarrowtail)$ defines the basic set of features on individuals in X. What we would like to do is to define function classes that would allow us to consider various boolean combinations of the basic features defined by $\rightarrowtail$. This will be done in the next subsection. Here, we first define the largest possible class of functions that can be obtained from boolean combinations of predicates in $S_{\rightarrowtail}$. This is done indirectly through the feature space $B(X, S_{\rightarrowtail})$ induced by the basic hypothesis language.

Definition 3.3.3. Let $(X, \rightarrowtail)$ be a basic hypothesis language. We define $\mathbb{BF}(X, \rightarrowtail)$ to be the set of all boolean functions over $B(X, S_{\rightarrowtail})$.

Definition 3.3.4. Given a basic hypothesis language $(X, \rightarrowtail)$, we say a class $\mathcal{F}$ of predicates over X is equivalent to $\mathbb{BF}(X, \rightarrowtail)$, denoted $\mathcal{F} \sim \mathbb{BF}(X, \rightarrowtail)$, if

1. for every $p \in \mathcal{F}$, there exists $f \in \mathbb{BF}(X, \rightarrowtail)$ such that $\forall x \in X$, $(p\ x) = f(S_{\rightarrowtail|x})$;
2. for every $f \in \mathbb{BF}(X, \rightarrowtail)$, there exists $p \in \mathcal{F}$ such that $\forall x \in X$, $f(S_{\rightarrowtail|x}) = (p\ x)$.

One can show equivalence between $\mathbb{BF}(X, \rightarrowtail)$ and the set $\mathbb{P}(X, \rightarrowtail)$ of all predicates over X that can be formed by conjoining (in syntactically-correct manner) any number of predicates from $S_{\rightarrowtail}$ using the boolean connectives $\{\wedge, \vee, \neg\}$ (lifted to the predicate level) and brackets. In that sense, $\mathbb{BF}(X, \rightarrowtail)$ captures the largest possible subset of the set of all predicates over X that we can consider if we restrict ourselves to looking at X through the glass filter that is $\rightarrowtail$. A predicate class that is equivalent to $\mathbb{BF}(X, \rightarrowtail)$ is thus as rich as $\mathbb{P}(X, \rightarrowtail)$.

3.3.2 Basic Class Definitions

We now give some class definitions and state some of their properties.

Definition 3.3.5. Given a basic hypothesis language $(X, \rightarrowtail)$, the class k-*CNF*$(\rightarrowtail)$ is defined to be the set of all predicates over X that can be represented in the form

$$\wedge_n(\vee_{k_1}\ p_{1,1} \ldots p_{1,k_1})\ (\vee_{k_2}\ p_{2,1} \ldots p_{2,k_2}) \cdots (\vee_{k_n}\ p_{n,1} \ldots p_{n,k_n}) \tag{3.1}$$

where $n \in \mathbb{N}$, $k_i \leqslant k$, and $p_{i,j} \in S_{\rightarrowtail} \cup S_{\rightarrowtail_{neg}}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, k_i$. Analogously, we define the class k-*DNF*$(\rightarrowtail)$ to be the set of all predicates over X that can be represented in the form

$$\vee_n(\wedge_{k_1}\ p_{1,1} \ldots p_{1,k_1})\ (\wedge_{k_2}\ p_{2,1} \ldots p_{2,k_2}) \cdots (\wedge_{k_n}\ p_{n,1} \ldots p_{n,k_n}) \tag{3.2}$$

where $n \in \mathbb{N}$, $k_i \leqslant k$, and $p_{i,j} \in S_{\rightarrowtail} \cup S_{\rightarrowtail_{neg}}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, k_i$.

Note that k-*CNF*$(\rightarrowtail)$ is the set of all distinct predicates over X that can be represented in the form of (3.1), *not* the set of all syntactically-distinct predicates over X having the form of (3.1). This comment applies to k-*DNF*$(\rightarrowtail)$ and every other class we define below.

Observation 3.3.6. *Let $(X, \rightarrowtail)$ be a basic hypothesis language. Then*

$$k\text{-}CNF(\rightarrowtail) = \{\neg p \mid p \in k\text{-}DNF(\rightarrowtail)\};\ \textit{and}$$
$$k\text{-}DNF(\rightarrowtail) = \{\neg p \mid p \in k\text{-}CNF(\rightarrowtail)\}.$$

Observation 3.3.7. *Let $(X, \rightarrowtail)$ be a basic hypothesis language. Then*

$$k\text{-}DNF(\rightarrowtail) = k\text{-}CNF(\rightarrowtail) \sim \mathbb{BF}(X, \rightarrowtail)$$

when $k = \#(\rightarrowtail)$.

Definition 3.3.8. Given a basic hypothesis language $(X, \rightarrowtail)$, the class j/k-*DT*$(\rightarrowtail)$ is defined to be the set of all predicates over X representable using a decision tree of at most depth k, where each predicate at the non-terminal nodes is a conjunction of

at most j predicates from $S_{\rightarrowtail}$. Decision stumps are captured as a special case when $k = 1$.

Given $(X, \rightarrowtail)$, j/k-$DT(\rightarrowtail)$ is the class of predicates considered by ALKEMY when we enrich the predicate rewrite system from $\rightarrowtail$ to $\rightarrowtail' = \{top \rightarrowtail \wedge_k top \ldots top\} \cup \rightarrowtail$ and learn with the top-down tree-induction algorithm, with an additional restriction on the depth of trees.

Observation 3.3.9. *Let $(X, \rightarrowtail)$ be a basic hypothesis language. $1/k$-$DT(\rightarrowtail) \sim \mathbb{BF}(X, \rightarrowtail)$ when $k = \#(\rightarrowtail)$.*

Proof. It is easy to check Part 1 of Definition 3.3.4. We show Part 2 here. Suppose $S_{\rightarrowtail} = \{p_1, p_2, \ldots, p_n\}$. Consider the complete depth-k decision tree T with predicate p_i appearing at every non-terminal node at depth i. Every function in $\mathbb{BF}(X, \rightarrowtail)$ can be represented using T by appropriate labelling of its terminal nodes. (Only the labels of terminal nodes that correspond to an entry in $B(X, S_{\rightarrowtail})$ matters, of course.) □

Definition 3.3.10. Given a basic hypothesis language $(X, \rightarrowtail)$, the class k-$DL(\rightarrowtail)$ is defined to be the set of all predicates over X representable using a decision list where each predicate in the list is a conjunction of at most k predicates from $S_{\rightarrowtail} \cup S_{\rightarrowtail_{neg}}$.

Given $(X, \rightarrowtail)$, k-$DL(\rightarrowtail)$ is the class of predicates considered by ALKEMY when we enrich the predicate rewrite system from $\rightarrowtail$ to

$$\rightarrowtail' = \{top \rightarrowtail \wedge_k top \ldots top\} \cup \rightarrowtail \cup \rightarrowtail_{neg}$$

and learn with the covering algorithm.

As seen earlier in §2.4.3.2, generating $\rightarrowtail_{neg}$ from $\rightarrowtail$ can be a cumbersome affair. This motivates the consideration of monotone decision lists, defined next.

Definition 3.3.11. Given a basic hypothesis language $(X, \rightarrowtail)$, the class k-$MDL(\rightarrowtail)$ is defined to be the set of all predicates over X representable using a decision list where each predicate in the list is a conjunction of at most k predicates from $S_{\rightarrowtail}$.

Observation 3.3.12. *Let $(X, \rightarrowtail)$ be a basic hypothesis language. Given a predicate p in k-$DL(\rightarrowtail)$, we can obtain $\neg p$ by switching all the labels in p. This implies that k-$DL(\rightarrowtail)$ is closed under negation. The same is true for k-$MDL(\rightarrowtail)$.*

It is known that both k-$DL(\rightarrowtail)$ and k-$MDL(\rightarrowtail)$ are universal representations.

Proposition 3.3.13. *Let $(X, \rightarrowtail)$ be a basic hypothesis language. Then*

$$k\text{-}DL(\rightarrowtail) = k\text{-}MDL(\rightarrowtail) \sim \mathbb{BF}(X, \rightarrowtail)$$

when $k = \#(\rightarrowtail)$.

Proof. To see that k-$DL(\rightarrowtail) \sim \mathbb{BF}(X, \rightarrowtail)$, observe that k-$DNF(\rightarrowtail) \subseteq k$-$DL(\rightarrowtail)$, and note k-$DNF(\rightarrowtail) \sim \mathbb{BF}(X, \rightarrowtail)$ when $k = \#(\rightarrowtail)$. For a proof of the universality of k-$MDL(\rightarrowtail)$, see Proposition 1 in [89]. □

3.3.3 Relationships

Relationship between Trees and Lists

Proposition 3.3.14 ([169]). *Let $(X, \rightarrowtail)$ be a basic hypothesis language. We have for all $k \in \{1, \ldots, \#(\rightarrowtail)\}$, $1/k$-DT$(\rightarrowtail) \subseteq k$-CNF$(\rightarrowtail) \cap k$-DNF$(\rightarrowtail)$*

Proof. Given a decision tree T in $1/k$-DT$(\rightarrowtail)$, one can create an equivalent predicate in k-DNF$(\rightarrowtail)$ by creating a conjunction of predicates for each leaf labelled 1 in T, and joining them together in a disjunction. To create an equivalent formula in k-CNF$(\rightarrowtail)$, create a conjunction of predicates for each leaf labelled 0 in T, join them together in a disjunction, then negate the whole formula and rewrite. □

Proposition 3.3.15 ([169]). *Let $(X, \rightarrowtail)$ be a basic hypothesis language. We have for all $k \in \{1, \ldots, \#(\rightarrowtail)\}$, k-CNF$(\rightarrowtail) \cup k$-DNF$(\rightarrowtail) \subseteq k$-DL$(\rightarrowtail)$*

Proof. It is clear that k-DNF$(\rightarrowtail)$ is a subset of k-DL$(\rightarrowtail)$. By Observations 3.3.6 and 3.3.12, k-CNF$(\rightarrowtail)$ is also a subset of k-DL$(\rightarrowtail)$. □

In light of Propositions 3.3.14 and 3.3.15, it is no surprise that j/k-DT$(\rightarrowtail)$ is a subset of jk-DL$(\rightarrowtail)$ since one can show that j/k-DT$(\rightarrowtail) \subseteq jk$-DNF$(\rightarrowtail)$. What is surprising is perhaps the fact that the extra predicates afforded by $\rightarrowtail_{neg}$ are not needed.

Proposition 3.3.16. *Let $(X, \rightarrowtail)$ be a basic hypothesis language. $\forall j, k \in \{1, \ldots, \#(\rightarrowtail)\}$, j/k-DT$(\rightarrowtail) \subseteq jk$-MDL$(\rightarrowtail)$.*

Proof. This is a straightforward generalization of Theorem 6 in [89]. The convention is that left branches in a tree correspond to true branches, and right branches correspond to false branches. Given a tree T in j/k-DT$(\rightarrowtail)$, we can construct an equivalent list L in jk-MDL$(\rightarrowtail)$ as follows. For each leaf $l_i, 1 \leqslant i \leqslant m$, in T from left to right, construct a node (t_{l_i}, o_{l_i}) where t_{l_i} is the conjunction of all left-branching j-conjunctions of predicates in the path from the root to the leaf l_i. If there are no left-branches in the path to a leaf, which is the case for the rightmost leaf, use the predicate top. Each t_{l_i} is clearly a conjunction of at most jk predicates. It is not hard to verify that the decision list $L = (t_{l_1}, o_{l_1}), (t_{l_2}, o_{l_2}), \ldots, (t_{l_m}, o_{l_m})$ defines the same predicate as T. □

There is a related result that doesn't restrict the depth of trees. In [71], the authors defined a certain class of rank-r decision trees and gave an efficient learning algorithm for it. Informally, the rank of a tree T is the height of the largest completely balanced subtree in T; it measures how (un)balanced a decision tree is. [27] shows that this class is also a subset of k-DL.

Relationship between k-DL, k-MDL, k-CNF and k-DNF

Under the (strong) assumption that $B(X, S_{\rightarrowtail}) = \{0, 1\}^{\#(\rightarrowtail)}$ (we will come back to examine this shortly), one can show that for all $1 \leqslant k < \#(\rightarrowtail)$,

1. (k-CNF$(\rightarrowtail) \cup k$-DNF$(\rightarrowtail)) \subset k$-DL$(\rightarrowtail)$ — See [169, Thm. 2].

2. k-*MDL*$(\rightarrowtail) \subset k$-*DL*$(\rightarrowtail)$ — Suppose $S_{\rightarrowtail} = \{p_1, p_2, \ldots, p_n\}$. The predicate

$$s = \neg(\wedge_n\ p_1\ p_2 \ldots p_n) = \vee_n\ \neg p_1\ \neg p_2 \ldots \neg p_n \tag{3.3}$$

   can be represented in k-*DL*$(\rightarrowtail)$ (in fact, 1-*DL*$(\rightarrowtail)$ suffices) but not in k-*MDL*$(\rightarrowtail)$.

3. k-*DNF*$(\rightarrowtail) \nsubseteq k$-*MDL*$(\rightarrowtail)$ — The predicate s given in (3.3) is in k-*DNF*$(\rightarrowtail)$.

4. k-*CNF*$(\rightarrowtail) \nsubseteq k$-*MDL*$(\rightarrowtail)$ — The predicate $\neg s$ is in k-*CNF*$(\rightarrowtail)$ but can't be in k-*MDL*$(\rightarrowtail)$ because k-*MDL*$(\rightarrowtail)$ is closed under negation.

5. k-*MDL*$(\rightarrowtail) \nsubseteq k$-*DNF*$(\rightarrowtail)$ — Let $p_1, \ldots, p_{k+1}$ be predicates in $S_{\rightarrowtail}$. The predicate

$$h = (\wedge_k\ p_1, \ldots p_k, 0), (\wedge_k\ p_2 \ldots p_{k+1}, 1), (top, 0) = (\wedge_{k+1}\ \neg p_1\ p_2 \ldots p_{k+1}) \tag{3.4}$$

   is in k-*MDL*$(\rightarrowtail)$ but not in k-*DNF*$(\rightarrowtail)$.

6. k-*MDL*$(\rightarrowtail) \nsubseteq k$-*CNF*$(\rightarrowtail)$ — The negation of the predicate h given in (3.4) is in k-*MDL*$(\rightarrowtail)$ but not in k-*CNF*$(\rightarrowtail)$.

The assumption $B(X, S_{\rightarrowtail}) = \{0,1\}^{\#(\rightarrowtail)}$ is, above all else, an assumption on the the relative independence between predicates in $S_{\rightarrowtail}$. It is also an assumption on the richness of the individual space X. Why do we need it? Note that, in each of the six cases above, the truth of the statement is demonstrated with the construction of a particular predicate that can't possibly exist in a certain class. Such non-existence arguments usually rely on the fact that there is some kind of structure in $B(X, S_{\rightarrowtail})$. The exact form of that structure has to be realized with an assumption, and $B(X, S_{\rightarrowtail}) = \{0,1\}^{\#(\rightarrowtail)}$ is the simplest, albeit strongest, assumption one can make.

Can we do without an assumption on $B(X, S_{\rightarrowtail})$? The answer is probably no. If there is no structure in $B(X, S_{\rightarrowtail})$, then we cannot possibly demonstrate that a certain predicate p does not exist in a function class $\mathcal{H}$ defined on top of $B(X, S_{\rightarrowtail})$ since one of the predicates in $S_{\rightarrowtail}$ can very well turn out to be p, in which case p must be in $\mathcal{H}$.

Another way to see the need for an assumption on $B(X, S_{\rightarrowtail})$ is to consider the two classes k-*MDL*$(\rightarrowtail)$ and k-*DL*$(\rightarrowtail)$. We show in (2) above that k-*MDL*$(\rightarrowtail)$ is a strict subset of k-*DL*$(\rightarrowtail)$. This is natural and we expect it to be true. But note that the relationship does not actually hold if $\rightarrowtail$ is already closed under negation, in which case k-*MDL*$(\rightarrowtail) = k$-*DL*$(\rightarrowtail)$. What all this means in practice is that (blindly) enriching a predicate rewrite system $\rightarrowtail$ may not actually get one a strictly richer class of functions, and that the exact structure of $S_{\rightarrowtail}$ (with respect to X) must be taken into account when performing such operations.

Relationship between Lists and Linear Threshold Functions

This next result suggests an interesting link between covering algorithms and statistical learning techniques like (kernelized) perceptrons and support vector machines. It is included here for interest.

Definition 3.3.17. Given a basic hypothesis language $(X, \rightarrowtail)$, the class *LT*$(\rightarrowtail)$ is de-

fined to be the set of all predicates over X that can be represented in the form

$$f(x) = \text{sgn}\left(\sum_{i=1}^{n} w_i(p_i\ x) + b\right)$$

where $n = \#(\rightarrowtail)$, $w_i, b \in \mathbb{R}$, $p_i \in S_{\rightarrowtail}$ and sgn$(\cdot)$ is the signum function.

Proposition 3.3.18 ([2]). *Given* $(X, \rightarrowtail)$, *we have* 1-*DL*$(\rightarrowtail) \subseteq$ *LT*$(\rightarrowtail)$.

Remark. Most of the results given in this section are reformulations of well-known results in the study of boolean functions. The contribution here is really the realization that boolean function theory is relevant to the study of ALKEMY. (For this, I credit the propositionalization ([122],[123], [116]) viewpoint in ILP.) The subsequent adaptation of the theory and some of its results to the setting where the full hypercube $\{0,1\}^n$ is not available (see §3.3.1) is actually not hard.

3.4 Generalization Bounds

This section deals with arguably the most important issue in classification learning, that of generalization. Specifically, we want to understand the relationship between the true and empirical errors of hypotheses produced by ALKEMY under standard assumptions on the data-generating process. The size of the training data and the complexity of the predicate class used are important parameters in this kind of investigation.

In what follows, $\log$ denotes logarithm to base 2, $\ln$ denotes the natural logarithm, and $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ denote, respectively, the ceiling and floor functions. The set of natural numbers $\{1, 2, 3, \ldots\}$ is denoted $\mathbb{N}$.

We start with a reminder of some basic concepts. Let X be an arbitrary set and $\mathcal{F}$ a class of predicates over X. The growth function of $\mathcal{F}$, $\Pi_{\mathcal{F}} : \mathbb{N} \to \mathbb{N}$, is defined by $\Pi_{\mathcal{F}}(n) = \max\{|\mathcal{F}_{|x}| : x \in X^n\}$, where

$$\mathcal{F}_{|x} = \{(f(x_1), \ldots, f(x_n)) : f \in \mathcal{F}\}.$$

Given $x \in X^n$, if $|\mathcal{F}_{|x}| = 2^n$, then we say x is *shattered* by $\mathcal{F}$. (Equivalently, we say a subset Y of X is shattered by $\mathcal{F}$ if each subset Z of Y can be picked out by a predicate in $\mathcal{F}$, i.e., there exists $f \in \mathcal{F}$ such that $\forall z \in Z.\ f(z) = 1$ and $\forall z \in Y \setminus Z.\ f(z) = 0$.) The Vapnik-Chervonenkis (VC) dimension of $\mathcal{F}$ is defined by

$$VCD(\mathcal{F}) = \max\{n : \Pi_{\mathcal{F}}(n) = 2^n\}$$

or ∞ if no such maximum exists.

We give a few useful facts about VC dimension here. All of these results, with the possible exception of Proposition 3.4.4, are well-known.

Proposition 3.4.1. *Let $\mathcal{F}$ be a predicate class with VC dimension d. Then for all positive integers m,*

$$\Pi_{\mathcal{F}}(m) \leqslant \sum_{i=0}^{d} \binom{m}{i}.$$

This, together with the fact that

$$\sum_{i=0}^{d} \binom{m}{i} < \left(\frac{em}{d}\right)^{d}$$

for $m \geqslant d \geqslant 1$, shows that the growth function for a predicate class $\mathcal{F}$ with VC dimension d satisfy

$$\Pi_{\mathcal{F}}(m) \begin{cases} = 2^m & \text{if } m \leqslant d; \\ < (\frac{em}{d})^d & \text{if } m > d. \end{cases}$$

Proposition 3.4.1, often called Sauer's Lemma, shows that the behaviour of the growth function of a predicate class is strongly constrained by its VC dimension.

Proposition 3.4.2. *Let $\mathcal{F}$ be a finite predicate class. Then $VCD(\mathcal{F}) \leqslant \lfloor \log |\mathcal{F}| \rfloor$.*

Proof. Observe that we need at least 2^d predicates to shatter a set of d elements. □

Proposition 3.4.3. *Let $\mathcal{F}$ and $\mathcal{G}$ be two classes of predicates over some set X. If $\mathcal{F} \subseteq \mathcal{G}$, then $VCD(\mathcal{F}) \leqslant VCD(\mathcal{G})$.*

Proof. The set that is shattered by $\mathcal{F}$ is shattered by $\mathcal{G}$. □

Proposition 3.4.4. *Let $\mathcal{F}$ and $\mathcal{G}$ be two classes of predicates over some set X. Then,*

$$VCD(\mathcal{F} \cup \mathcal{G}) \leqslant VCD(\mathcal{F}) + VCD(\mathcal{G}) + 1.$$

Proof. If the VC dimension of either $\mathcal{F}$ or $\mathcal{G}$ is ∞, then the bound holds trivially. Assume now that $VCD(\mathcal{F}) = d_1 < \infty$ and $VCD(\mathcal{G}) = d_2 < \infty$. Since $\Pi_{\mathcal{F}\cup\mathcal{G}}(m) \leqslant \Pi_{\mathcal{F}}(m) + \Pi_{\mathcal{G}}(m)$, it is clear that

$$VCD(\mathcal{F} \cup \mathcal{G}) \leqslant \max\{m : \Pi_{\mathcal{F}}(m) + \Pi_{\mathcal{G}}(m) \geqslant 2^m\}. \tag{3.5}$$

The maximum exists because for $m > \max\{d_1, d_2\}$, both $\Pi_{\mathcal{F}}(m)$ and $\Pi_{\mathcal{G}}(m)$ grow only polynomially. We claim that the RHS of inequality (3.5) is bounded by $m = d_1 + d_2 + 1$. To see this, observe that

$$\Pi_{\mathcal{F}}(m) + \Pi_{\mathcal{G}}(m) \leqslant \sum_{i=0}^{d_1} \binom{m}{i} + \sum_{i=0}^{d_2} \binom{m}{i} = \sum_{i=0}^{d_1} \binom{m}{i} + \sum_{i=m-d_2}^{m} \binom{m}{i}$$

by Sauer's Lemma, and this is less than 2^m if $m > d_1 + d_2 + 1$ since $\sum_{i=0}^{m} \binom{m}{i} = 2^m$. □

Proposition 3.4.5. *Let $\mathcal{F}$ be a class of predicates over some set X with VC dimension d. Define $\mathcal{F}_{neg} = \{\neg f : f \in \mathcal{F}\}$. Then $VCD(\mathcal{F}_{neg}) = d$.*

Proof. Observe that a set is shattered by $\mathcal{F}$ iff it is shattered by $\mathcal{F}_{neg}$. □

Propositions 3.4.5 and 3.4.4 together imply that closing a predicate class $\mathcal{H}$ under negation would raise the VC dimension of the resulting class to at most $2\,VCD(\mathcal{H})+1$.

Proposition 3.4.6. *Suppose $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k$ are classes of predicates over some set X with $VCD(\mathcal{F}_i) = d_i < \infty$ for each i. Let $b_k : \{0,1\}^k \to \{0,1\}$ be an arbitrary boolean function, and*

$$\mathcal{H} = \{\, b_k\text{-}(f_1, f_2, \ldots, f_k) : f_i \in \mathcal{F}_i \,\}$$

where each $b_k\text{-}(f_1, f_2, \ldots, f_k) : X \to \{0,1\}$ is defined by

$$b_k\text{-}(f_1, f_2, \ldots, f_k)(x) = \begin{cases} 1 & \text{if } b_k\left(f_1(x), f_2(x), \ldots, f_k(x)\right) = 1; \\ 0 & \text{otherwise.} \end{cases}$$

Then $VCD(\mathcal{H}) \leqslant 2\left(\frac{d}{\ln 2}\ln\left(\frac{2d}{\ln 2}\right) - \sum_{i=1}^{k} d_i \log d_i\right)$ where $d = \sum_{i=1}^{k} d_i$.

Proof. We follow the proof strategy in [185]. Suppose $S \subseteq X$ with cardinality m is shattered by $\mathcal{H}$. Clearly,

$$\Pi_{\mathcal{H}}(m) \leqslant \prod_{i=1}^{k} \Pi_{\mathcal{F}_i}(m) \leqslant \prod_{i=1}^{k} \left(\frac{em}{d_i}\right)^{d_i}.$$

Since S is shattered, $2^m \leqslant \Pi_{\mathcal{H}}(m)$. Taking logs we get

$$m \leqslant \log(em) \sum_{i=1}^{k} d_i - \sum_{i=1}^{k} d_i \log d_i \tag{3.6}$$

$$= \frac{d}{\ln 2}\ln(em) - \sum_{i=1}^{k} d_i \log d_i \tag{3.7}$$

$$\leqslant \frac{d}{\ln 2}\left(\frac{m \ln 2}{cd} + \ln\left(\frac{ced}{\ln 2}\right) - 1\right) - \sum_{i=1}^{k} d_i \log d_i. \tag{3.8}$$

To bound $\ln(em)$ in the last step, we have used the inequality [1] $\ln a \leqslant ab - \ln b - 1$ for all $a, b > 0$ with $a = em$ and $b = \ln 2/ced$ for some constant $c > 1$. Putting $c = 2$ and solving for m gives the desired result. □

More facts about VC dimension, including the proof of Proposition 3.4.1, can be found in standard texts like [3] and [103].

[1] This can be obtained from the inequality $\forall x > 0, \ln x \leqslant x - 1$, an elementary result of calculus, by putting $x = ab$.

We now state two error bounds that can be used to analyse Alkemic decision trees. The first comes from classical VC theory and is presented in §3.4.1. The second, described in §3.4.2, is a more recent result that can be obtained from modern data-dependent analysis.

3.4.1 Classical Bounds

Assume that X is the input space. Let $\mathcal{D}$ be a distribution on $X \times \{-1, +1\}$, and S a finite subset of $X \times \{0, 1\}$. We denote by $\mathbf{P}_{(x,y)\sim\mathcal{D}}[E]$ the probability under $\mathcal{D}$ of an event E, and by $\mathbf{P}_{(x,y)\sim S}[E]$ the empirical probability of E, i.e., the proportion of points in S that lie in E.

The following is a result due to Vapnik and Chervonenkis [195].

Theorem 3.4.7 ([3]). *Suppose $\mathcal{H}$ is a set of predicates over X and $\mathcal{D}$ is a probability distribution on $X \times \{0, 1\}$. For $0 < \epsilon < 1$ and m a positive integer, we have*

$$\mathcal{D}^m\{|er_{\mathcal{D}}(h) - \hat{er}(h)| \geqslant \epsilon \text{ for some } h \in \mathcal{H}\} \leqslant 4\Pi_{\mathcal{H}}(2m)exp(-\frac{\epsilon^2 m}{8})$$

where $er_{\mathcal{D}}(h) = \mathbf{P}_{(x,y)\sim\mathcal{D}}(h(x) \neq y)$ and $\hat{er}(h)$ is the empirical error of h on the sample.

The theorem states that for a predicate class $\mathcal{H}$ that is not too complex, given a large enough random sample, the empirical and true errors for *every* predicate in $\mathcal{H}$ are close with high probability. The key parameter governing this phenomena is the growth function $\Pi_{\mathcal{H}}(2m)$ of $\mathcal{H}$. The bound is trivial if $\Pi_{\mathcal{H}}(2m)$ grows exponentially in m; but if $\Pi_{\mathcal{H}}(2m)$ grows only polynomially, the bound goes to zero exponentially quickly. By Sauer's Lemma, the latter condition is assured if $\mathcal{H}$ has finite VC dimension.

Theorem 3.4.7 can be restated (modulo slight modification in the proof) in the following more usable form.

Theorem 3.4.8 ([53]). *Let $\mathcal{H}$ be a predicate class with VC dimension d. For any probability distribution $\mathcal{D}$ on $X \times \{0, 1\}$, with probability $1-\delta$ over m random examples S, any predicate $h \in H$ that makes k errors on S satisfy*

$$\mathbf{P}_{(x,y)\sim\mathcal{D}}(h(x) \neq y) \leqslant \frac{2k}{m} + \frac{4}{m}\left(d\log\frac{2em}{d} + \log\frac{4}{\delta}\right)$$

provided $d \leqslant m$.

We now give a bound on the VC dimension of decision trees with node functions in some predicate class $\mathcal{U}$. In ALKEMY, $\mathcal{U}$ is defined using a predicate rewrite system. This result is probably known in one form or another, but I have not seen it spelled out before, hence this derivation.

Assume $top \in \mathcal{U}$ and consider the set $DT(\mathcal{U}, k)$ of all trees with depth at most k. We can concentrate on the subset $DT'(\mathcal{U}, k)$ of $DT(\mathcal{U}, k)$ consisting of completely-balanced trees with exactly 2^k leaf nodes since every tree T in $DT(\mathcal{U}, k)$ with less than 2^k leaf nodes has an equivalent tree T' in $DT'(\mathcal{U}, k)$, where T' is just T with extra

top-labelled non-terminal nodes attached. To bound the VC dimension of $DT'(\mathcal{U}, k)$, we partition it into $n = 2^{2^k}$ mutually exclusive sets $\{DT'(\mathcal{U}, k, \pi_i)\}_{i=1\ldots n}$ and bound the VC dimension of each set in turn. Here, $\pi_i \in \{0,1\}^{2^k}$ and $DT'(\mathcal{U}, k, \pi_i)$ consists of all the trees in $DT'(\mathcal{U}, k)$ with leaf nodes labelled from left to right by π_i. By Proposition 3.3.14, we can bound $VCD(DT'(\mathcal{U}, k, \pi_i))$ using Proposition 3.4.6 since each π_i corresponds to a boolean function involving $m(\pi_i)k$ predicates from $\mathcal{U}$, where $m(\pi_i) = \min\{t(\pi_i), f(\pi_i)\}$, $t(\pi_i)$ is the number of 1 in π_i, and $f(\pi_i)$ the number of 0 in π_i. Denoting $VCD(\mathcal{U})$ by d and applying Proposition 3.4.4, we get

$$VCD(DT(\mathcal{U}, k)) \leqslant \sum_{i=1}^{n} \left(\frac{m(\pi_i)kd}{\ln 2} \ln \frac{2m(\pi_i)kd}{\ln 2} - m(\pi_i)kd \log d \right) + n,$$

which is finite if d is finite.

3.4.2 Data-Dependent Bounds

As shown above, generalization bounds for decision trees obtained from classical VC theory suggest that the amount of data needed for learning should grow with the size of the tree; see also [71] and [73]. More recent data-dependent bounds show that this need not be the case. In [86], starting from margin bounds for two-layer neural networks (see [12] and [178]), the authors show how decision trees with node functions in some predicate class $\mathcal{U}$ can be represented as thresholded convex combinations of predicates in $\mathcal{U}$ and from that obtain bounds for decision trees that are qualitatively different from Theorem 3.4.8. The result is quite instructive and we outline its development here. For more details, the reader is referred to [86] and [138].

For a class $\mathcal{H}$ of $\{-1, +1\}$-valued functions defined on some set X, we denote by $co(\mathcal{H})$ the convex hull of $\mathcal{H}$, i.e., the set of $[-1, +1]$-valued functions of the form $\sum_i a_i h_i$, where $a_i \geqslant 0$, $\sum_i a_i = 1$, and $h_i \in \mathcal{H}$.

Theorem 3.4.9 ([86]). *Let $\mathcal{D}$ be a distribution on $X \times \{-1, +1\}$, $\mathcal{H}_1, \ldots, \mathcal{H}_k$ predicate classes with $VCD(\mathcal{H}_i) = d_i$, and $\delta > 0$. With probability at least $1 - \delta$ over a training set S of m examples chosen according to $\mathcal{D}$, every function $f \in co(\cup_{i=1}^{k} \mathcal{H}_i)$ and every θ satisfy*

$$\mathbf{P}_{(x,y)\sim\mathcal{D}}[yf(x) \leqslant 0] \leqslant 2\mathbf{P}_{(x,y)\sim S}[yf(x) \leqslant \theta] + O\left(\frac{1}{m} \left(\frac{1}{\theta^2} (d \log m + \log k) \log \frac{m\theta^2}{d} + \log \frac{1}{\delta} \right) \right),$$

where $d = \sum_i a_i d_{j_i}$, and the a_i's and j_i's are defined by $f = \sum_i a_i h_i$, $h_i \in \mathcal{H}_{j_i}$, $j_i \in \{1, \ldots, k\}$.

The interesting twist in Theorem 3.4.9 is that multiple base classes $\{\mathcal{H}_i\}_{i=1}^{k}$ can be used, and we only pay a complexity price involving the 'average' VC dimension of these base classes. For a proof of the theorem, see [138, §2.1]. The argument follows the same line of reasoning as that used for Theorems 1 and 2 in [178]. The faster rate of convergence to $2\mathbf{P}_{(x,y)\sim S}[yf(x) \leqslant \theta]$ can be obtained using a similar argument used in [138, Lemma 1.3].

We next show that decision trees can be represented as thresholded convex combinations of functions. For a tree T with N leaves, we denote by $\sigma_i \in \{-1, +1\}$ the label assigned to leaf i, and define the leaf functions $h_i : X \to \{0, 1\}$ by $h_i(x) = 1$ iff x reaches leaf i. A decision tree T can be represented in the equivalent functional form $T(x) = sgn(f(x))$, where $sgn(\alpha)$ is the signum function that returns 1 iff $\alpha \geqslant 0$, and

$$f(x) = \sum_{i=1}^{N} w_i \sigma_i h_i(x),$$

with $w_i > 0$ and $\sum_{i=1}^{N} w_i = 1$. Given any x, the only non-zero term in the sum is the one satisfying $h_i(x) = 1$, with σ_i giving the class, and w_i some measure of 'confidence'.

Let $\mathcal{U}$ be the node functions. We assume that $\mathcal{U}$ is closed under negation and that $top \in \mathcal{U}$. Define the class of leaf functions for leaves up to depth j by

$$\mathcal{H}_j = \{\wedge_r\, u_1 \dots u_r : u_i \in \mathcal{U}, r \leqslant j\}.$$

Let k_i denote the depth of leaf i, so $h_i \in \mathcal{H}_{k_i}$, and let $k = \max_i k_i$. A straightforward application of Proposition 3.4.6, with the constant c in its proof set to $2e \ln 2$, gives

$$VCD(\mathcal{H}_j) \leqslant 2j\, VCD(\mathcal{U}) \ln(2ej). \tag{3.9}$$

Plugging (3.9) into Theorem 3.4.9 and collapsing $\log$ terms and constants, we have for fixed $\delta > 0$, there is a constant c such that for any distribution $\mathcal{D}$, with probability at least $1 - \delta$ over the sample S,

$$\mathbf{P}_{(x,y)\sim\mathcal{D}}[T(x) \neq y] \leqslant 2\mathbf{P}_{(x,y)\sim S}[yf(x) \leqslant \theta] + \frac{\bar{d}}{\theta^2}\frac{c}{m} VCD(\mathcal{U}) \log^2 m \log k$$

where $\bar{d} = \sum_{i=1}^{N} w_i d_i$.

Different choices of the w_i's and the θ will yield different estimates of the error. Given a sample S and a tree T, let $P_i = \mathbf{P}_{(x,y)\sim S}[h_i(x) = 1]$, $Q_i = \mathbf{P}_{(x,y)\sim S}[y\sigma_i = -1 | h_i(x) = 1]$ and $P'_i = P_i(1 - Q_i)/(1 - \mathbf{P}_{(x,y)\sim S}[T(x) \neq y])$. One can easily check that $P'_i \in [0, 1]$ and $\sum_{i=1}^{N} P'_i = 1$. Let $P' = (P'_1, \dots, P'_N)$ and assume without loss of generality that $P'_1 \geqslant \dots \geqslant P'_N$. A natural choice is $w_i = P'_i$ and $P'_{j+1} \leqslant \theta < P'_j$ for some $j \in \{1, \dots, N\}$. This yields the bound

$$\mathbf{P}_{(x,y)\sim\mathcal{D}}[T(x) \neq y] \leqslant 2 \sum_{i=j+1}^{N} P'_i + \frac{\bar{d}}{\theta^2}\frac{c}{m} VCD(\mathcal{U}) \log^2 m \log k.$$

To get the best bound, we need to optimize the RHS of the inequality over the choices of j and θ; this can be computed cheaply given P'.

There are ways to restate the bound in more qualitative terms. For example, [138, §2.3] gives an approach to optimizing the selection of w_i's and θ by making a reasonable and empirically-tested assumption on the distribution of P'. We restate here the approach taken in [86]. In that paper, the bound was reformulated in terms of N_{eff},

defined by $N_{\mathit{eff}} = N(1-\rho(P',U))$ where $\rho(P',U) = \sum_{i=1}^{N}(P'_i - 1/N)^2$ is the quadratic distance between P' and the uniform probability vector $U = (1/N, \ldots, 1/N)$. For consistent trees, Theorem 3.4.10 can be established. Theorem 3.4.11 applies to inconsistent trees.

Theorem 3.4.10 ([86]). *For a fixed $\delta > 0$, there is a constant c that satisfies the following. Let $\mathcal{D}$ be a distribution on $X \times \{-1,+1\}$. Consider the class of decision trees of depth up to k, with decision functions in $\mathcal{U}$. With probability at least $1-\delta$ over the training set S of size m, every decision tree T that is consistent with S satisfies*

$$\mathbf{P}_{(x,y)\sim\mathcal{D}}[T(x) \neq y] \leqslant c\left(\frac{N_{\mathit{eff}}\, VCD(\mathcal{U}) \log^2 m \log k}{m}\right)^{1/2}.$$

Theorem 3.4.11 ([86]). *For a fixed $\delta > 0$, there is a constant c that satisfies the following. Let $\mathcal{D}$ be a distribution on $X \times \{-1,+1\}$. Consider the class of decision trees of depth up to k, with decision functions in $\mathcal{U}$. With probability at least $1-\delta$ over the training set S of size m, every decision tree T satisfies*

$$\mathbf{P}_{(x,y)\sim\mathcal{D}}[T(x) \neq y] \leqslant \mathbf{P}_{(x,y)\sim S}[T(x) \neq y] + c\left(\frac{N_{\mathit{eff}}\, VCD(\mathcal{U}) \log^2 m \log k}{m}\right)^{1/3}.$$

In both cases, N_{eff} is a data-dependent quantity. Thus the same tree can be simple for one P' and complex for another.

The bounds provided by Theorems 3.4.10 and 3.4.11 are qualitatively different from that provided by Theorem 3.4.8, and they can be significantly better especially for large trees.

3.4.3 Some Tools for Calculating VC Dimensions

As shown, the VC dimension of node functions is an important parameter in the generalization behaviour of Alkemic decision trees. To understand the nature of learning with ALKEMY, we need to develop methods to calculate the VC dimensions of (more-or-less arbitrary) predicate classes definable using predicate rewrite systems. The problem seems difficult at first sight; in fact, it was listed as an open research question in [130, Ex 6.6]. But recent progress has shown that solutions to some important aspects of the general problem are actually rather straightforward. These results are reported next.

We first describe the development of some basic tools in this section. Armed with these, we will then proceed in §3.4.4 to analyse five illustrative predicate rewrite systems selected from [130]. We remark that some of the tools developed here have applications beyond ALKEMY.

For the most part, we will abstract away from predicate rewrite systems and just work on simple predicate classes defined on 'collections' of natural numbers. As we shall see, this is a useful simplification since there is a straightforward mapping between natural numbers and arbitrary finite sets that we can exploit. The results are

organized into sections on composition of functions, tuples, and sets and multisets.

In what follows, sets with k elements are called k-sets; subsets with k elements are called k-subsets.

3.4.3.1 Compositions of Functions

Proposition 3.4.12. *Suppose X_1 and X_2 are two arbitrary sets. Let f be a function from X_1 to X_2, and $\mathcal{G}$ a predicate class, where each $g \in \mathcal{G}$ is a predicate over X_2. Define $f \circ \mathcal{G}$ by*

$$f \circ \mathcal{G} = \{f \circ g : g \in \mathcal{G}\}.$$

Then $VCD(f \circ \mathcal{G}) \leqslant VCD(\mathcal{G})$.

Proof. If $VCD(\mathcal{G}) = \infty$, then the bound clearly holds. Assume $VCD(\mathcal{G}) = d < \infty$. Suppose $VCD(f \circ \mathcal{G}) > d$. Then there exists a set $\{x_1, x_2, \ldots, x_{d+1}\} \subseteq X_1$ that is shattered by $f \circ \mathcal{G}$. Now, it is necessary that $f(x_i) \neq f(x_j)$ if $i \neq j$. (Otherwise, there is no way to distinguish between x_i and x_j). This implies that the set $\{f(x_1), f(x_2), \ldots, f(x_{d+1})\}$ is shattered by $\mathcal{G}$. But this contradicts the fact that $VCD(\mathcal{G}) = d$. □

The VC dimension of $f \circ \mathcal{G}$ can be strictly smaller than that of $\mathcal{G}$, since f can restrict the domain of functions in $\mathcal{G}$ and reduce the size of the biggest set that is shattered by $\mathcal{G}$. If $VCD(\mathcal{G}) = \infty$, this is one useful way to control $\mathcal{G}$.

Next a related question. Let $\mathcal{F}$ and $\mathcal{G}$ be two function classes, where each $f \in \mathcal{F}$ is a function from X_1 to X_2, and each $g \in \mathcal{G}$ is a predicate over X_2. Define $\mathcal{F} \circ \mathcal{G}$ by

$$\mathcal{F} \circ \mathcal{G} = \{f \circ g : f \in \mathcal{F}, g \in \mathcal{G}\}.$$

Could it be that $VCD(\mathcal{F} \circ \mathcal{G}) \leqslant VCD(\mathcal{G})$? This is not true, as shown next.

Example 3.4.13. Let $\mathcal{F} = \{\min, \max, \text{sum}\}$ be the set of functions from all finite subsets of $\mathbb{N}$ to $\mathbb{N}$ defined as follows: $\min$ and $\max$ return, respectively, the minimum and maximum number of a set; and sum returns the sum of the numbers in a set. Let $\mathcal{G}$ be the class of intervals on $\mathbb{N}$. It is known that $VCD(\mathcal{G}) = 2$. The VC dimension of $\mathcal{F} \circ \mathcal{G}$ is at least 3 since the set $\{X_1 = \{1, 2, 3\}, X_2 = \{3, 4\}, X_3 = \{2, 3\}\}$ is shattered by $\mathcal{F} \circ \mathcal{G}$. The key observation here is that we can use functions in $\mathcal{F}$ to map distinct individuals in $2^{\mathbb{N}}$ to the same number. For example, to label X_1 and X_3 true, and X_2 false, we can use $\max$ to nullify the difference between X_1 and X_3, resulting in the set $\{\max(X_1), \max(X_2), \max(X_3)\} = \{3, 4\}$. The desired labelling can then be achieved using $\mathcal{G}$. ◀

3.4.3.2 Predicate Classes on Tuples

This next result is a recast of Exercise 6.5 in [130].

Proposition 3.4.14. *Suppose $m > 1$. Let $\mathcal{F}_m$ be the following class of predicates*

$$\mathcal{F}_m = \{top\} \cup \{\wedge_k \, f_{i_1,j_1} \ldots f_{i_k,j_k} \; : \; k \in \mathbb{N}, i_l \in \{1, \ldots, m\}, j_l \in \mathbb{N}\}$$

where each $f_{i,j} : \mathbb{N}^m \to \{0,1\}$ *is defined by*

$$f_{i,j}\ (x_1, \ldots, x_m) = \begin{cases} 1 & \textit{if } (x_i = j); \\ 0 & \textit{otherwise.} \end{cases}$$

$VCD(\mathcal{F}_m) = m$.

Proof. We first show that $VCD(\mathcal{F}_m) \geqslant m$. The set $X = \{X_1, \ldots, X_m\}$ where $X_i \in \mathbb{N}^m$ has value 2 at the i-th component and value 1 everywhere else is shattered by $\mathcal{F}_m$. It is easy to pick out the empty set, the whole set X, the 1-subsets and $(m-1)$-subsets. To pick out a subset $S = \{X_{i_1}, \ldots, X_{i_n} \ : 2 \leqslant n \leqslant m-2\} \subset X$, use the predicate $\wedge_{m-n}\ f_{i_1,1} \ldots f_{i_{m-n},1}$ where i_j are the components where all the elements in S have the same value 1.

We next show $VCD(\mathcal{F}_m) \leqslant m$. Consider an arbitrary set $Y \subset \mathbb{N}^m$ with $m+1$ elements. We claim that there is no way to pick out all m-subsets of Y. This is because there are $\binom{m+1}{m} = m+1$ such subsets, and we need to use at least one component to pick out each, but there are only m components available for use. □

Proposition 3.4.14 has been known for a long time in another form. The upper bound was given in [72], and the lower bound in [151]. Essentially the same arguments were used.

The condition $m > 1$ is needed here since $VCD(\mathcal{F}_m) = 2$ if $m = 1$.

Observation 3.4.15. *In the proof of Proposition 3.4.14, we never needed conjunction of more than m-2* $f_{i,j}$*'s to shatter a set of size* m*. This implies that if* k *in the definition of* $\mathcal{F}_m$ *is restricted to the range* $\{1, \ldots, m-2\}$*,* $VCD(\mathcal{F}_m)$ *would still be* m*.*

Proposition 3.4.16. *Suppose* $m \geqslant 1$*. Let* $\mathcal{G}_m$ *be the following class of predicates*

$$\mathcal{G}_m = \{\textit{bottom}\} \cup \{\wedge_k\ g_{i_1,j_1} \ldots g_{i_k,j_k} \ : \ k \in \mathbb{N}, i_l \in \{1, \ldots, m\}, j_l \in \mathbb{N}\}$$

where each $g_{i,j} : \mathbb{N}^m \to \{0,1\}$ *is defined by*

$$g_{i,j}\ (x_1, \ldots, x_m) = \begin{cases} 1 & \textit{if } (x_i \neq j); \\ 0 & \textit{otherwise.} \end{cases}$$

Then $VCD(\mathcal{G}_m) = \infty$.

Proof. For each $n \in \mathbb{N}$, the set

$$X_n = \{(1,1,1,\ldots,1), (2,1,1,\ldots,1), \ldots, (n,1,1,\ldots,1)\}$$

is shattered. We only need the first component. The 1's in the remaining components are just place-fillers. □

3.4.3.3 Predicate Classes on Sets and Multisets

One of the distinguishing features of our knowledge representation formalism is the admission of sets and multisets for data modelling. We now look at some predicate classes defined on sets.

Proposition 3.4.17. *Let $\mathcal{F}_\forall$ be the class of predicates $\mathcal{F}_\forall = \{f_{i,j} : i, j \in \mathbb{N}, j \geqslant i\}$ where each $f_{i,j} : 2^\mathbb{N} \to \{0, 1\}$ is defined by*

$$f_{i,j}(t) = \begin{cases} 1 & \text{if } \forall x \in t.\ i \leqslant x \leqslant j; \\ 0 & \text{otherwise.} \end{cases}$$

Then $VCD(\mathcal{F}_\forall) = 2$.

Proof. It is easy to show that $VCD(\mathcal{F}_\forall) \geqslant 2$. Assume there exists a set of three elements $S = \{X, Y, Z\}$ that is shattered by $\mathcal{F}_\forall$. Clearly, none of the elements in S can be the empty set, which evaluates to 1 for each $f \in \mathcal{F}_\forall$. Further, each element in S must be finite. (Shattering is impossible otherwise.) Denote by $\max(A)$ and $\min(A)$ the biggest and smallest numbers in a (finite) set A of numbers and define the range of A by $range(A) = \{\min(A), \ldots, \max(A)\}$. We have

$$\forall A, B \in S,\ A \neq B \Rightarrow range(A) \nsubseteq range(B)$$

since if $range(A) \subseteq range(B)$, there is no way to make B true without also making A true. Without loss of generality, assume $\min(X) < \min(Y) < \min(Z)$. This implies $\max(X) < \max(Y) < \max(Z)$. Now, there is no $f_{i,j} \in \mathcal{F}_\forall$ such that $f_{i,j}(X) = 1$, $f_{i,j}(Z) = 1$, and $f_{i,j}(Y) = 0$ since any (i, j)-interval that covers both $\min(X)$ and $\max(Z)$ must also cover every number in the range $\{\min(Y), \ldots, \max(Y)\} \supseteq Y$. □

Proposition 3.4.18. *Let $\mathcal{F}_\exists$ be the class of predicates $\mathcal{F}_\exists = \{f_{i,j} : i, j \in \mathbb{N}, j \geqslant i\}$ where each $f_{i,j} : 2^\mathbb{N} \to \{0, 1\}$ is defined by*

$$f_{i,j}(t) = \begin{cases} 1 & \text{if } \exists x \in t.\ i \leqslant x \leqslant j; \\ 0 & \text{otherwise.} \end{cases}$$

Then $VCD(\mathcal{F}_\exists) = \infty$.

Proof. It is sufficient to show that the subset $\mathcal{F}'_\exists = \{f_{i,i} : i \in \mathbb{N}\}$ of $\mathcal{F}_\exists$ has infinite VC dimension. For each $n \in \mathbb{N}$, we can construct a set $\{X_1, X_2, \ldots, X_n\}$ that is shattered by $\mathcal{F}'_\exists$ as follows. Enumerate all the subsets of $N = \{1, 2, \ldots, n\}$, assigning them numbers from 1 to 2^n. For instance, when $n = 3$ we get

1	2	3	4	5	6	7	8
$\emptyset$	$\{1\}$	$\{2\}$	$\{3\}$	$\{1,2\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3\}$

.

Now define X_i to be the set of all numbers assigned to a subset of N having i as a member. Continuing with our example for $n = 3$, we obtain the set

$$\{X_1 = \{2, 5, 6, 8\},\ X_2 = \{3, 5, 7, 8\},\ X_3 = \{4, 6, 7, 8\}\}.$$

It is clear that $\{X_1, X_2, \ldots, X_n\}$ constructed this way is shattered by $\mathcal{F}'_\exists$. □

There are different ways one can constrain $\mathcal{F}_\exists$. For example, by restricting the domain of each $f_{i,j}$ to sets of numbers with some maximum cardinality m, we can cut down the VC dimension of the class to at most $2m$.

The next result is quite informative, in light of Proposition 3.4.18.

Proposition 3.4.19. *Let $\mathcal{F} = \{f_i : i \in \mathbb{N}\}$ and $\mathcal{G} = \{g_i : i \in \mathbb{N}\}$ be predicate classes where each $f_i : 2^{\mathbb{N}} \to \{0, 1\}$ is defined by*

$$f_i(t) = \begin{cases} 1 & \text{if } \exists x \in t.\ x \geqslant i; \\ 0 & \text{otherwise,} \end{cases}$$

and each $g_i : 2^{\mathbb{N}} \to \{0, 1\}$ is defined by

$$g_i(t) = \begin{cases} 1 & \text{if } \exists x \in t.\ x \leqslant i; \\ 0 & \text{otherwise.} \end{cases}$$

Then $VCD(\mathcal{F}) = VCD(\mathcal{G}) = 1$. Further, $VCD(\mathcal{F} \cup \mathcal{G}) = 2$.

Proof. It is easy to construct examples to show that $VCD(\mathcal{F}) \geqslant 1$, $VCD(\mathcal{G}) \geqslant 1$ and $VCD(\mathcal{F} \cup \mathcal{G}) \geqslant 2$. Simple indirect arguments show that larger shatterable sets do not exist. □

Careful examination of the essential difference between Proposition 3.4.18 and Proposition 3.4.19 led to the development of Proposition 3.4.21 below. As we shall see, it is a useful tool for analysing many different predicate rewrite systems.

Definition 3.4.20. Let X be a set and $\mathcal{F}$ a class of predicates over X. We say a set $S \subseteq X$ is *disintegrated* by $\mathcal{F}$ if for every $x \in S$, there exists an $f \in \mathcal{F}$ such that $f(x) = 1$ and $f(y) = 0$ for all $y \in S \setminus \{x\}$.

Proposition 3.4.21. *Let X be a set and $\mathcal{F}$ a class of predicates over X. Let $\mathcal{G} = \{g_f : f \in \mathcal{F}\}$ be the class of predicates where each $g_f : 2^X \to \{0, 1\}$ is defined by*

$$g_f(t) = \begin{cases} 1 & \text{if } \exists x \in t.\ f(x) = 1; \\ 0 & \text{otherwise.} \end{cases}$$

If there exists a finite $S \subseteq X$ such that $|S| \geqslant 2$ and S is disintegrated by $\mathcal{F}$, then $VCD(\mathcal{G}) \geqslant \lfloor \log |S| \rfloor$.

Proof. Proceeding as in Proposition 3.4.18, we can assign a different element of S to each subset of $N = \{1, 2, \ldots, \lfloor \log |S| \rfloor\}$. Defining X_i as the set of all elements assigned to a subset of N in which i occurs gives us a subset of 2^X that is shattered by $\mathcal{G}$. □

As a simple application of Proposition 3.4.21, we give this next result for sets of tuples of constants. It is a handy tool for obtaining lower bounds on the VC dimension of many predicate rewrite systems.

Proposition 3.4.22. *Let N be a finite subset of $\mathbb{N}$ satisfying $|N| \geqslant 2$. Suppose $m \geqslant n > 0$ and let $\mathcal{G}_{m,n}$ be the class of predicates*

$$\mathcal{G}_{m,n} = \{g_{\{(i_l,j_l)\}_{1\leqslant l\leqslant k}} : k \in \{1,\ldots,n\}, i_l \in \{1,\ldots,m\}, j_l \in N\}$$

where each $g_{\{(i_l,j_l)\}_{1\leqslant l\leqslant k}} : 2^{N^m} \to \{0,1\}$ is defined by

$$g_{\{(i_l,j_l)\}_{1\leqslant l\leqslant k}}(t) = \begin{cases} 1 & \text{if } \exists (x_1,\ldots,x_m) \in t.\ (x_{i_1} = j_1) \wedge \cdots \wedge (x_{i_k} = j_k); \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$VCD(\mathcal{G}_{m,n}) \geqslant \begin{cases} \lfloor m \log |N| \rfloor & \text{if } n = m; \\ \lfloor \log(\sum_{k=1}^{n} \binom{m}{k}(|N|-1)^k) \rfloor & \text{otherwise.} \end{cases}$$

Proof. Let

$$\mathcal{F}_{m,n} = \{f_{\{(i_l,j_l)\}_{1\leqslant l\leqslant k}} : k \in \{1,\ldots,n\}, i_l \in \{1,\ldots,m\}, j_l \in N, i_1 < \cdots < i_k\}$$

where each $f_{\{(i_l,j_l)\}_{1\leqslant l\leqslant k}} : N^m \to \{0,1\}$ is defined by

$$f_{\{(i_l,j_l)\}_{1\leqslant l\leqslant k}}(x_1,\ldots,x_m) = \begin{cases} 1 & \text{if } (x_{i_1} = j_1) \wedge \cdots \wedge (x_{i_k} = j_k); \\ 0 & \text{otherwise.} \end{cases}$$

We use Proposition 3.4.21 to get the lower bounds. When $n = m$, we can use for S the whole set N^m, which is clearly disintegrated by $\mathcal{F}_{m,m}$. When $n < m$, we construct S as follows. Pick an $x \in N$ at random and consider the following subset of $\mathcal{F}_{m,n}$:

$$\mathcal{F}_{m,n,x} = \{f_{\{i_l,j_l\}_{1\leqslant l\leqslant k}} : k \in \{1,\ldots,n\}, i_l \in \{1,\ldots,m\}, j_l \in N \setminus \{x\}, i_1 < \cdots < i_k\}.$$

For each predicate $f_{\{(i_l,j_l)\}_{1\leqslant l\leqslant k}} \in \mathcal{F}_{m,n,x}$ add to S the tuple that has value j_l at the i_l-th component, and x everywhere else. (For instance, when $m = 5, k = 2, N = \{1,2,3\}$ and $x = 3$, given $f_{\{(1,2),(3,1)\}}$, we add $(3,3,1,3,2)$ to S.) It's not hard to see that each element in S can be picked out by the predicate that generated it. Further,

$$|S| = |\mathcal{F}_{m,n,x}| = \sum_{k=1}^{n} \binom{m}{k}(|N|-1)^k.$$

The condition $|N| \geqslant 2$ ensures that $|S| \geqslant 2$ in both cases. □

We next look at multisets. The difference between a set and a multiset is that an element can occur multiple times in a multiset. Some of the results given for sets clearly carry over to multisets with little change. The multiplicity of elements allowed in multisets can sometimes be exploited, as done in this next result. First some notation.

Let A be a multiset of elements from some set X. In the following, we denote by $\#(A,x)$ the multiplicity of $x \in X$ in A. Further, we denote by $\mathbb{N}_0$ the set $\{0\} \cup \mathbb{N}$.

Definition 3.4.23. Let A and B be multisets of elements from some set X. We define the pairwise maximum between A and B, denoted $A \sqcup B$, as follows: $A \sqcup B$ is the multiset that contains, for all $x \in X$, $\max\{\#(A,x), \#(B,x)\}$ occurrences of x. For example, $\{1,1,2,2,2\} \sqcup \{1,2,2,2,2,3,3,3\} = \{1,1,2,2,2,2,3,3,3\}$.

Proposition 3.4.24. *Suppose X and Y are non-empty finite subsets of $\mathbb{N}$. Let $\mathcal{F}$ be the class of predicates $\mathcal{F} = \{f_{i,j} : i \in X, j \in Y\}$ where each $f_{i,j} : {\mathbb{N}_0}^{\mathbb{N}} \to \{0,1\}$ is defined by*

$$f_{i,j}(t) = \begin{cases} 1 & \text{if } \#(t,i) \geqslant j; \\ 0 & \text{otherwise.} \end{cases}$$

Let $d \in \mathbb{N}$. If $|Y| \geqslant d+1$ and $|X| \geqslant \binom{d}{i}$ for all $i \in \{1,\ldots,d\}$, then $VCD(\mathcal{F}) \geqslant d$.

Proof. The proof is in two stages. In the first stage, we show that given a function ψ from the powerset of $D = \{1,\ldots,d\}$ to ${\mathbb{N}_0}^{\mathbb{N}}$ satisfying a certain property, we can construct a set $Z = \{Z_1,\ldots,Z_d\}$ that is shattered by $\mathcal{F}$. In the second stage, we show that ψ exists and give a simple algorithm for constructing it.

Stage 1 We denote by (x,y) the multiset that contains y occurrences of x and nothing else. Assume ψ satisfies the following property: For all $S \subseteq D$, we have

1. $\psi(S) = (x,y)$ for some $x, y \in \mathbb{N}$, and
2. for all $A \subseteq D$ not equal to S, if $\psi(A) = (x,z)$ for some $z \in \mathbb{N}$ and $|A| \geq |S|$, then $S \subset A$ and $y > z$.

Given such a function ψ, define $Z_i = \bigsqcup \{\psi(S) : S \subseteq D, i \in S\}$ for each $i \in D$. We now argue that the set $Z = \{Z_1,\ldots,Z_d\}$ so-constructed is shattered by $\mathcal{F}$. Specifically, we show that for all $S \subseteq D$, $f_{x,y}(Z_i) = 1$ if $i \in S$ and $f_{x,y}(Z_i) = 0$ otherwise, given that $\psi(S) = (x,y)$.

Consider an arbitrary $S \subseteq D$ with $\psi(S) = (x,y)$. If $i \in S$, by construction, Z_i contains at least y occurrences of x and $f_{x,y}(Z_i) = 1$. Consider now the case when $i \notin S$. If $\#(Z_i,x) = 0$, then $f_{x,y}(Z_i) = 0$ as desired. If $\#(Z_i,x) > 0$, then there exists $A \subseteq D$ such that $i \in A$ and $\psi(A) = (x,z)$ for some $z \in \mathbb{N}$. We can assume without loss of generality that A is the set with the largest z. If $|A| \geq |S|$, then by the property of ψ, we have $y > z$ and $S \subset A$, which implies $f_{x,y}(Z_i) = 0$. If $|A| < |S|$, then by the property of ψ, we have $z > y$ and $A \subset S$. (Simply substitute the set A for the variable S and the set S for the variable A in the statement of the property of ψ.) This case can't arise since $A \subset S$ and $i \in A$ together imply $i \in S$, contradicting $i \notin S$.

Stage 2 It suffices to show that one such ψ exists. We will give a more general result that shows that not only does ψ exists, we can actually find many instances of it efficiently using well-studied algorithms in graph theory.

Given X and Y both non-empty finite subsets of $\mathbb{N}$, we first use the **Label** algorithm given below to label the subsets of D. For each $S \subseteq D$, we then define $\psi(S)$ to be the label assigned to S. To get some intuition, we first give a high-level description

of the labelling algorithm. Conceptually, we first lay out in a sequence the subsets of D in groups, starting from the empty set (group 0), followed by the 1-subsets (group 1), the 2-subsets (group 2), ..., and finally finishing at D (group $|D|$). The algorithm starts by labelling the largest group and then iteratively label the next two largest unlabelled groups until every subset of D has a label.

We now give the algorithm. The variables l, u and m are integers. In the algorithm, we denote by $Y[i]$ and $X[i]$ the i-th largest elements in Y and X. The condition $|X| \geq \binom{d}{i}$ for all i comes about because of Step 2. The condition $|Y| \geq d+1$ comes from the fact that there are $d+1$ groups of subsets of D. Example 3.4.27 below gives a concrete example of the labelling.

Alg. **Label**

1. $l \leftarrow 1;\ u \leftarrow 1;\ m \leftarrow \min\{i : \forall j.\ \binom{d}{j} \leqslant \binom{d}{i}\}$;
2. Label the m-subsets of D with $(X[i], Y[\lceil d/2+1 \rceil])$ in increasing order of i.
3. If $m - l < 0$, goto Step 6;
4. $C \leftarrow$ the $(m-l+1)$-subsets of D;
5. For each $(m-l)$-subset S of D
   (a) Pick an $L \in C$ with label $(x, Y[m])$ such that $S \subset L$ and label S with $(x, Y[m+1])$;
   (b) $C \leftarrow C \setminus L$;
6. If $m + u > d$, terminate;
7. $C \leftarrow$ the $(m+u-1)$-subsets of D;
8. For each $(m+u)$-subset S of D
   (a) Pick an $L \in C$ with label $(x, Y[m])$ such that $L \subset S$ and label S with $(x, Y[m-1])$;
   (b) $C \leftarrow C \setminus L$;
9. $l \leftarrow l+1;\ u \leftarrow u+1;$ Goto Step 3;

By design, the function ψ constructed from a labelling obtained by **Label**, assuming it terminates, satisfies the condition stated earlier. We now show that the Label algorithm always terminate successfully. For that, we need to show that Steps 5(a) and 8(a) can always be performed for each S. We will show this for Step 5(a); the argument for Step 8(a) is similar. What we are trying to do is in fact to find a matching in a bipartite graph. The vertices of the graph consists of the $(m-l)$ and $(m-l+1)$-subsets of D, with the $(m-l)$-subsets forming the first partition, and the $(m-l+1)$-subsets the second. There is an edge from an $(m-l)$-subset A to an $(m-l+1)$-subset B iff $A \subset B$. By the choice of m, we have

$$\text{no. of } (m-l)\text{-subsets} = \binom{d}{m-l} \leqslant \binom{d}{m-l+1} = \text{no. of } (m-l+1)\text{-subsets}.$$

Thus we seek a matching of cardinality $\binom{d}{m-l}$.

To show that such a matching exists and can be found efficiently, we introduce a concept from graph theory.

> **Definition 3.4.25.** A vertex cover of a graph $G = (V, E)$ is a set $U \subseteq V$ such that every edge of G is incident with a vertex in U.

We make use of the following known result. For a proof, see, for example, [60].

> **Theorem 3.4.26 (König 1931).** *The maximum cardinality of a matching in a bipartite graph G is equal to the minimum cardinality of a vertex cover of G.*

The set of $(m-l)$-subsets with cardinality $\binom{d}{m-l}$ is clearly a vertex cover. A straightforward indirect argument shows that there is no smaller vertex cover. The existence of our desired matching then follows from Theorem 3.4.26. There are efficient network flow algorithms for finding (all) such matchings; see, for instance, [159, Chap. 10] and [50, Chap. 27].

Finally, the labelling algorithm will always terminate at Step 6 by the choice of m in Step 1. □

Example 3.4.27. Suppose $X = \{1, \ldots, 6\}$ and $Y = \{1, \ldots, 5\}$. Let $\mathcal{F}$ be as defined in Proposition 3.4.24. To construct a set $Z = \{Z_1, Z_2, Z_3, Z_4\}$ that is shattered by $\mathcal{F}$, we first label the subsets of $D = \{1, 2, 3, 4\}$ according to the **Label** algorithm. One acceptable labelling is the following.

$$\emptyset\ (1,5)$$
$$\{1\}\,(1,4),\ \{2\}\,(4,4),\ \{3\}\,(2,4),\ \{4\}\,(3,4)$$
$$\{1,2\}\,(1,3),\ \{1,3\}\,(2,3),\ \{1,4\}\,(3,3),\ \{2,3\}\,(4,3),\ \{2,4\}\,(5,3),\ \{3,4\}\,(6,3)$$
$$\{1,2,3\}\,(1,2),\ \{1,2,4\}\,(5,2),\ \{1,3,4\}\,(2,2),\ \{2,3,4\}\,(6,2)$$
$$\{1,2,3,4\}\,(1,1)$$

Based on the function ψ obtained from the labelling, we construct

$$\begin{aligned} Z = \{\ & Z_1 = \{1,1,1,1,2,2,2,3,3,3,5,5\}, \\ & Z_2 = \{4,4,4,4,1,1,1,5,5,5,6,6\}, \\ & Z_3 = \{2,2,2,2,4,4,4,6,6,6,1,1\}, \\ & Z_4 = \{3,3,3,3,5,5,5,6,6,6,2,2,1\}\ \}. \end{aligned}$$

It can be easily verified that Z is shattered by $\mathcal{F}$. ◀

Observation 3.4.28. *It is possible to weaken the condition on $|Y|$ in Proposition 3.4.24 using a more scrupulous grouping of the subsets, especially for large values of d. We note here a simple way to weaken that to $|Y| \geqslant d - 1$ by treating the empty set as part of the 1-subsets, and the whole set D as part of the $(d-1)$-subsets during labelling. Labelling is possible because the empty set, being a subset of every other set, is connected to all the 2-subsets; and the set D, being a superset of every other set, is connected to all the $(d-2)$-subsets.*

3.4.4 Five Illustrations

Building on the results presented in the previous section, we now analyse some instructive examples of predicate rewrite systems taken from [130, Chap. 6]. For each illustration, we will briefly introduce the problem and give details on the way individuals are represented and the hypothesis language used. Unimportant details like training examples have been left out for brevity; readers can consult [130] or [31] for more information. We will also revert back to the notation introduced in Chapter 2.

3.4.4.1 Tennis

The first illustration is a simple problem due to Quinlan (see [162]) that involves learning the concept of playing, or not playing, tennis, according to the weather.

Representation of Individuals The types $Outlook$, $Temperature$, $Humidity$, $Wind$ and the type synonym $Weather$ are defined as follows.

$$Sunny, Overcast, Rain : Outlook$$
$$Hot, Mild, Cool : Temperature$$
$$High, Normal, Low : Humidity$$
$$Strong, Medium, Weak : Wind$$
$$Weather = Outlook \times Temperature \times Humidity \times Wind$$

The function $playTennis$ to be learned has signature $playTennis : Weather \to \Omega$.

Predicate Rewrite System

$$top \rightarrowtail \wedge_2\ top\ top;$$
$$top \rightarrowtail projOutlook \circ top;\ top \rightarrowtail projTemperature \circ top;$$
$$top \rightarrowtail projHumidity \circ top;\ top \rightarrowtail projWind \circ top;$$
$$top \rightarrowtail (= Sunny);\ top \rightarrowtail (= Overcast);\ top \rightarrowtail (= Rain);$$
$$top \rightarrowtail (= Hot);\ top \rightarrowtail (= Mild);\ top \rightarrowtail (= Cool);$$
$$top \rightarrowtail (= High);\ top \rightarrowtail (= Normal);\ top \rightarrowtail (= Low);$$
$$top \rightarrowtail (= Strong);\ top \rightarrowtail (= Medium);\ top \rightarrowtail (= Weak);$$

Proposition 3.4.29. $VCD(S_{\rightarrowtail}) = 4$.

Proof. This follows from Proposition 3.4.14 and Observation 3.4.15. □

3.4.4.2 Musk

This is the Musk problem introduced in [63]. Briefly, the problem is to determine whether or not a molecule has a musk odour. The molecules generally have many different conformations and, presumably, only one conformation is responsible for

the activity. Each conformation is a tuple of 166 floating-point numbers, where 162 of these numbers represent the distance in angstroms from some origin in the conformation out along a radial line to the surface of the conformation and the other four numbers represent the position of a specific oxygen atom. For convenience, we discretized the floating-point numbers into 13 intervals, resulting in the following.

Representation of Individuals

$$-6, -5, \ldots, 5, 6 : \mathit{Distance}$$
$$\mathit{Conformation} = \mathit{Distance} \times \cdots \times \mathit{Distance}$$
$$\mathit{Molecule} = \{\mathit{Conformation}\}.$$

Here the product type $\mathit{Distance} \times \cdots \times \mathit{Distance}$ contains 166 components. The function musk to be learned has signature $\mathit{musk} : \mathit{Molecule} \to \Omega$.

Predicate Rewrite System

$$\mathit{top} \rightarrowtail \mathit{setExists}_1\ (\wedge_3\ \mathit{top}\ \mathit{top}\ \mathit{top})$$
$$\mathit{top} \rightarrowtail \mathit{proj}_i \circ (= j) \text{ where } i \in \{1, 2, \ldots, 166\}, j \in \{-6, -5, \ldots, 6\}$$

Here, a macro language is used to represent all predicate rewrites of a certain form. We will use this notation whenever it is convenient to do so.

Proposition 3.4.30. $\mathit{VCD}(S_{\rightarrowtail}) = 30$.

Proof. We have $\mathit{VCD}(S_{\rightarrowtail}) \leqslant \lfloor \log |S_{\rightarrowtail}| \rfloor = \lfloor \log 1,679,615,641 \rfloor = 30$ by Proposition 3.4.2. We have the lower bound

$$\mathit{VCD}(S_{\rightarrowtail}) \geqslant \lfloor \log(\sum_{k=1}^{3} \binom{166}{k} (12)^k) \rfloor = \lfloor \log 1,295,658,552 \rfloor = 30$$

by Proposition 3.4.22. □

3.4.4.3 Climate

Consider next the problem of deciding whether a climate in some country is pleasant or not. The climate is modelled by a multiset. Each item in a multiset is a term characterizing the main features of the weather during a day and the multiplicity of the item is the number of times during a year a day with those particular weather features occurs.

Representation of Individuals We use the same representation for $\mathit{Weather}$ as in the Tennis problem. A climate is modelled as a multiset $\mathit{Climate} = \mathit{Weather} \to \mathit{Nat}$ and the function $\mathit{pleasant}$ to be learned has signature $\mathit{pleasant} : \mathit{Climate} \to \Omega$.

Predicate Rewrite System

$$
\begin{aligned}
&\mathit{top} \rightarrowtail (\mathit{domMcard}\ \mathit{top}) \circ (> 0);\\
&\mathit{top} \rightarrowtail \wedge_4\ (\mathit{projOutlook} \circ \mathit{top})\ (\mathit{projTemperature} \circ \mathit{top})\\
&\qquad\qquad (\mathit{projHumidity} \circ \mathit{top})\ (\mathit{projWind} \circ \mathit{top});\\
&\mathit{top} \rightarrowtail (= \mathit{Sunny});\ \mathit{top} \rightarrowtail (= \mathit{Overcast});\ \mathit{top} \rightarrowtail (= \mathit{Rain});\\
&\mathit{top} \rightarrowtail (= \mathit{Hot});\ \mathit{top} \rightarrowtail (= \mathit{Mild});\ \mathit{top} \rightarrowtail (= \mathit{Cool});\\
&\mathit{top} \rightarrowtail (= \mathit{High});\ \mathit{top} \rightarrowtail (= \mathit{Normal});\ \mathit{top} \rightarrowtail (= \mathit{Low});\\
&\mathit{top} \rightarrowtail (= \mathit{Strong});\ \mathit{top} \rightarrowtail (= \mathit{Medium});\ \mathit{top} \rightarrowtail (= \mathit{Weak});\\
&(> i) \rightarrowtail (> i + 50)\ \text{ where }\ i \in \{0, 50, \ldots, 300\}.
\end{aligned}
$$

Proposition 3.4.31. $8 \leqslant \mathit{VCD}(S_{\rightarrowtail}) \leqslant 11$.

Proof. We have $\mathit{VCD}(S_{\rightarrowtail}) \leqslant \lfloor \log |S_{\rightarrowtail}| \rfloor = \lfloor \log 2057 \rfloor = 11$ by Proposition 3.4.2. We use Proposition 3.4.24 and Observation 3.4.28 to establish the lower bound. All the tuples of type $\mathit{Weather}$ can be numbered and form the set X as in Proposition 3.4.24, with $|X| = 81$. Each predicate in $S_{\rightarrowtail}$ of the form

$$
\begin{aligned}
(\mathit{domMcard}\ (\wedge_4\ (\mathit{projOutlook} \circ (= A))\ (\mathit{projTemperature} \circ (= B))\\
(\mathit{projHumidity} \circ (= C))\ (\mathit{projWind} \circ (= D)))) \circ (> j)
\end{aligned}
$$

is equivalent to some $f_{i,j+1}$ as defined in Proposition 3.4.24, where i is the labelling number of (A, B, C, D). There are 81 ways to instantiate the variables A, B, C and D. The variable j can take on values in the set $Y = \{1, 51, 101, 151, 201, 251, 301, 351\}$. The largest d satisfying $|Y| \geqslant d - 1$ and $|X| \geqslant \binom{d}{i}$ for all i is $d = 8$. □

3.4.4.4 East-West Challenge

We now examine the East-West challenge, a problem involving lists we introduced earlier in Example 2.2.4.

Proposition 3.4.32. *Let $\rightarrowtail$ be as in Example 2.4.6.* $10 \leqslant \mathit{VCD}(S_{\rightarrowtail}) \leqslant 13$.

Proof. We have $\mathit{VCD}(S_{\rightarrowtail}) \leqslant \lfloor \log |S_{\rightarrowtail}| \rfloor = \lfloor \log 10661 \rfloor = 13$ by Proposition 3.4.2. To establish the lower bound, we proceed in the same way as Proposition 3.4.22, but taking into account the different domains of the components in Car. An element from each component is reserved as a default value, in the same way an $x \in N$ is used in Proposition 3.4.22. From that, we get a set Z of Car objects that can be used to construct a shatterable set D of sets of Car objects, where $|D| = \lfloor \log |Z| \rfloor$ by Proposition 3.4.21. Clearly, one can recover a Train object from each element in D. A straightforward counting exercise yields $|Z| = 1175$, giving the lower bound $\lfloor \log |Z| \rfloor = 10$. □

3.4.4.5 Chemicals

The next illustration involves learning a theory to predict whether a given chemical molecule has a certain property.

Representation of Individuals We use an undirected graph to model a molecule – an atom is a vertex in the graph and a bond is an edge. The type $Element$ is the type of the (relevant) chemical elements; the type $Bond$ is the type of the chemical bonds between atoms.

$$Br, C, Cl, F, H, I, N, O, S : Element$$
$$S, D, T, R : Bond$$

We also make the following type synonyms.

$$AtomType = Nat$$
$$Charge = Float$$
$$Atom = Element \times AtomType \times Charge$$

$Float$ here is the type of floating-point numbers. The vertices of a graph have type $Atom$ and the edges have type $Bond$, leading to the following definition.

$$Molecule = Graph\ Atom\ Bond.$$

The function $active$ to be learned has signature $active : Molecule \to \Omega$.

Predicate Rewrite System

$$top \rightarrowtail (subgraphs\ 3) \circ (setExists_1\ top)$$
$$top \rightarrowtail \wedge_2\ (vertices \circ (domCard\ top) \circ (> 0))\ (edges \circ (domCard\ top) \circ (> 0));$$
$$top \rightarrowtail \wedge_2\ (connects \circ (msetExists_2\ top\ top))\ (edge \circ top);$$
$$top \rightarrowtail vertex \circ projAtomType \circ top;$$
$$top \rightarrowtail (= 3);\ top \rightarrowtail (= 22);\ top \rightarrowtail (= 27);\ top \rightarrowtail (= 38);$$
$$top \rightarrowtail (= 40);\ top \rightarrowtail (= 45);\ top \rightarrowtail (= 195);$$
$$top \rightarrowtail (= S);\ top \rightarrowtail (= D);\ top \rightarrowtail (= T);\ top \rightarrowtail (= R);$$
$$(> 0) \rightarrowtail (> 1).$$

Proposition 3.4.33. $8 \leqslant VCD(S_{\rightarrowtail}) \leqslant 12$.

Proof. We have $VCD(S_{\rightarrowtail}) \leqslant \lfloor \log |S_{\rightarrowtail}| \rfloor = \lfloor \log 8137 \rfloor = 12$ by Proposition 3.4.2. We use Proposition 3.4.21 to establish the lower bound. Consider the subset of predicates in $S_{\rightarrowtail}$ of the form

$$\begin{aligned}(subgraphs\ 3) \circ (setExists_1\ (\wedge_2 \\ (vertices \circ (domCard\ (vertex \circ projAtomType \circ (= X))) \circ (> 0))) \\ (edges \circ (domCard\ (\wedge_2\ (connects \circ (msetExists_2\ (vertex \circ projAtomType \circ (= Y)) \\ (vertex \circ projAtomType \circ (= Z)))) \\ (edge \circ (= B)))) \circ (> 0))),\end{aligned}$$

where X, Y, Z are variables of type $AtomType$ satisfying the constraints $X \neq Y$, $X \neq Z$ and $Y \leqslant Z$, and B is a variable of type $Bond$ taking a value in the set $\{S, D, T\}$. There are $7 \cdot \binom{6+2-1}{2} \cdot 3 = 441$ such predicates. For each such predicate, we construct a graph of the form

$$(\{(1,(C,X,0.142)),(2,(C,Y,0.142)),(3,(C,Z,0.142)\},\{(\langle 1,2\rangle,R),(\langle 2,3\rangle,B)\}).$$

Using this set of graphs, which is disintegrated by the set of all predicates derivable from the second predicate rewrite using $\rightarrowtail$, we construct a set D of sets of graphs as in Proposition 3.4.18. For each element g in D, we put all the (sub)graphs in g together to form a graph with $|g|$ disconnected components, after appropriate relabelling of the vertices. It can be checked that the set of graphs constructed this way is shattered by $S_{\rightarrowtail}$. The cardinality of the set is $\lfloor \log 441 \rfloor = 8$. □

3.4.5 Tighter Bounds

Let us recap the general procedure used to analyse the predicate rewrite systems. In all but one example, we establish an upper bound by counting the size of the predicate class. A lower bound is then given by an explicit construction of a set of individuals that is shattered by the predicate class in question, making use of the rich structures available. Interestingly, the upper and lower bounds are never too far apart. Now one would expect that it is possible to do a lot better than a naïve counting of the predicate class; apparently not. It turns out that a body of work culminating in [7] in the field of ILP came up with the same general conclusion for first-order predicate classes. What are we to make of these results?

It was shown in [182] (see also [3, Chap. 5]) that for a predicate class with high VC dimension, there exist distributions that will force the learning algorithm to require a large training sample to obtain good generalization. This, together with the results presented in the previous section, implies that, in general, the true errors of hypotheses in the rich predicate classes used by ALKEMY cannot be easily estimated from empirical data, and that, *in the worst case*, the number of training data needed grows with the size of the predicate classes used. The problem is that if we don't make any assumption on the underlying distribution, then we must be prepared to accept the possibility that everything can conspire against the learning algorithm. The more structures we introduce into the representation of individuals and the hypothesis language, the more structures there are to be exploited for producing bad cases.

However, learning with predicate classes that have high VC dimensions is possible if the underlying distribution is benign, and this information can be obtained from the training data. For instance, [181] shows that the VC dimension of a predicate class on the training sample can be used as a measure of how helpful the distribution is in identifying the target concept, and gives error bounds in terms of that. More recently, [14] gives error bounds in terms of the Rademacher and Gaussian complexities of predicate classes, and these can be estimated from the training data. PAC-Bayes and PAC-MDL bounds, which are also data-dependent results, can also help us obtain

tighter bounds. Some relevant work along this line of investigation include [134] and [170]. Such data-dependent analysis can yield error bounds that are much tighter than those given in §3.4.1 and §3.4.2.

Another way forward is to accept the negative worst-case results and seek, instead, positive average-case results, possibly along the line of [147].

3.5 Optimization Issues

Up until this point, we have been concerned with information-theoretic questions. This section deals with computation issues. We analyse the search behaviour of the three main algorithms presented in Section 3.2. For each algorithm, we state

1. the desired optimization goal;
2. the optimality of the algorithm with respect to the optimization goal; and
3. the time and space complexities of the algorithm.

Different techniques – the tricks of the trade so to speak – that can be used to speed up search in practice are also discussed.

3.5.1 The Stump Algorithm

3.5.1.1 Optimization Goal

Given a predicate rewrite system $\rightarrowtail$, the set of all decision stumps that can be formed using $\rightarrowtail$ is exactly the set $S_{\rightarrowtail} \cup S_{\rightarrowtail_{neg}}$. As we have seen in Section 3.4, if the VC dimension of $S_{\rightarrowtail} \cup S_{\rightarrowtail_{neg}}$ is finite, in the presence of sufficiently many examples, the true and empirical errors of every stump will be close with high probability. This suggests that picking the decision stump with the lowest empirical error is a good winning strategy.

3.5.1.2 Optimality

In the default pruning mode, ALKEMY conducts an exhaustive search of the hypothesis space. The output decision stump is thus guaranteed to be the best on the training data according to the optimization goal.

Incomplete Searches Sometimes an exhaustive search is impossible and we have to resort to incomplete searches. Several such algorithms are given in §3.2.1.3. There is a rich body of work on understanding the behaviour of such algorithms (see, for example, [172] for a survey and references), we will not dwell on them here except to remark that, in practice, it is often sufficient to just find an acceptably-accurate decision stump in a reasonable amount of time.

3.5.1.3 Time and Space Complexities

In considering computational complexity issues, we will focus on the LR search algorithm (Figure 3.1). The SeenSet algorithm tends to behave poorly when the search space is massive, and its use is largely restricted to quick incomplete searches.

Given a predicate rewrite system $\rightarrowtail$ and a set of examples $\mathcal{E} = \{(x_i, y_i)\}_{i=1}^{N}$ as input, the worst-case time complexity of learning is

$$\sum_{i=1}^{\#(\rightarrowtail)} \sum_{j=1}^{N} cost(p_i, x_j)$$

where $cost(p, x)$ is the computational cost of evaluating $(p\ x)$. Thanks to pruning, the cost in actual applications is usually a lot lower, often by orders of magnitude. There is, however, not much one can say about the effectiveness of pruning in general; it is largely an application-dependent factor.

The space complexity is dominated by the size of the open list and this is at all times smaller than the number of predicates in $S_{\rightarrowtail}$ without a redex. This number can usually be bounded by a simple function of the number of predicate rewrites in $\rightarrowtail$ whose body has no redex. Computationally, it is a small quantity of little concern.

The worst-case time complexity of the learning algorithm is quite bad. We now outline a few techniques and strategies that can alleviate the computational burden.

Estimating the Prune Parameter Having a good initial value for the prune parameter can cut down search time significantly. A simple way to do this is to use the SeenSet search algorithm in conjunction with one of the incomplete search strategies proposed to find a good decision stump quickly, and then to use its accuracy as the initial value for the prune parameter.

Predicate Evaluation If the cost of predicate evaluation is the dominating factor, the following optimization procedure is worthwhile. Consider two predicates p and q in the search space. If q is a descendant of p, then q is stronger than p and we have for all x, $(p\ x) = 0$ implies $(q\ x) = 0$. This fact can be used to save time. As learning proceeds, a complete evaluation record of *certain* predicates are stored in a data structure. To evaluate each new predicate r on a set of individuals, we locate in the data structure its closest ancestor s in the predicate search space and compute only those individuals evaluated to 1 by s; the rest can be assigned 0 without further ado.

The decision on which predicates to keep is another instance of the space-time complexity tradeoff question. Obviously, predicates with no redexes should not be kept. The number of predicates kept also shouldn't be so large that the cost of finding the closest ancestor outweighs the potential saving in predicate evaluation time.

How much speed are we likely to gain from this algorithm? Experiments suggest that a performance boost of around 20–30% can be achieved.

More details about the algorithm can be found in §3.5.2.3.

Predicate Rewrite Pruning Sometimes pruning can be done at the predicate rewrite level, which is a lot more effective than pruning at the predicate level. We make use of the following simple fact.

Proposition 3.5.1. *Let $\mathcal{E}$ be a set of examples and $\mathcal{P}_i$, $1 \leqslant i \leqslant n$, partitions of $\mathcal{E}$. If $\mathcal{P}$ is a refinement of every $\mathcal{P}_i$, then $A_{\mathcal{P}} \leqslant \min_{1 \leqslant i \leqslant n} B_{\mathcal{P}_i}$.*

Proof. By part (a) of Proposition 3.2.9, $A_{\mathcal{P}} \leqslant B_{\mathcal{P}_i}$ for $1 \leqslant i \leqslant n$. □

We illustrate the technique with two examples.

Example 3.5.2. Consider the following predicate rewrite system

$$\begin{aligned} &top \rightarrowtail \wedge_k \; top \ldots top \\ &top \rightarrowtail p_1 \quad top \rightarrowtail p_2 \quad \ldots \quad top \rightarrowtail p_n \end{aligned}$$

where each top in the body of the first rewrite can be instantiated with any one of the p_i's. Given training examples, by Proposition 3.5.1, the accuracy of every predicate in $S_{\rightarrowtail}$ of the form $\wedge_k \; p_{i_1} \ldots p_{i_k}$ is upper-bounded by $\min_{j \in \{i_1, \ldots, i_k\}} B_{p_j}$. Knowing this, one can compute the accuracy and refinement bound of each p_i in a preprocessing step. We can then safely throw away all the predicate rewrites whose body p_i has a refinement bound lower than $\max_j A_{p_j}$ since if $B_{p_i} < \max_j A_{p_j}$, then all conjunctions involving p_i will have accuracy lower than $\max_j A_{p_j}$ and therefore not worth investigating. ◀

Example 3.5.3. Consider the following predicate rewrite system

$$\begin{aligned} &top \rightarrowtail setExists_k \; top \ldots top \\ &top \rightarrowtail p_1 \quad top \rightarrowtail p_2 \quad \ldots \quad top \rightarrowtail p_n \end{aligned}$$

where each top in the body of the first rewrite can be instantiated with any one of the p_i's. Define $s_i = setExists_1 \; p_i$. Given training examples, by Proposition 3.5.1, the accuracy of every predicate in $S_{\rightarrowtail}$ of the form $setExists_k \; p_{i_1} \ldots p_{i_k}$ is upper-bounded by $\min_{j \in \{i_1, \ldots, i_k\}} B_{s_j}$. This means we can perform predicate rewrite pruning in the same fashion as in Example 3.5.2 by calculating the accuracy and refinement bound of each s_i in a preprocessing step. ◀

Notice that the individual components in the two examples are similar in form; the components in Example 3.5.2 actually have the form $\wedge_1 \; p_i$.

Parallelization of Search Advances in parallel computing are making it possible for us to tackle problems hitherto unimaginable. In a recent ILP position paper [158], the authors identified parallelization of search using cheap cluster computers [37] as a target area for further research. We show here how the flexibility of predicate rewrite systems can be exploited to parallelize certain kinds of large-scale 'embarrassingly-parallel' search problems.

The method is best illustrated with an example. Consider again the predicate rewrite system shown in Example 3.5.2. The size of the search space is $O(n^k)$, which can be very large for non-trivial values of k and n. To search effectively using a cluster computer, we can partition the search space into many different independent parts by partially expanding out the first predicate rewrite as follows.

$$
\begin{aligned}
&top \rightarrowtail \wedge_k\ p_1\ top \ldots top \\
&top \rightarrowtail \wedge_k\ p_2\ top \ldots top \\
&\ldots \\
&top \rightarrowtail \wedge_k\ p_n\ top \ldots top \\
&top \rightarrowtail p_1 \quad \ldots \quad top \rightarrowtail p_n
\end{aligned}
$$

A master-slave architecture [200] can then be employed to dish out the different subspaces for parallel processing by slave processors. The subspaces are non-overlapping if we use the LR search algorithm. As defined, the subspaces are not equally big. This can cause load balancing problems in the sense that a small number of processors may end up working on a large subset of the space, hence delaying the overall running time unnecessarily. The solution is to simply create more partitions by further expansion.

The communication overhead required under this scheme is minimal, and linear speedup can be expected. This technique was implemented by Wu on Bunyip, a prize-winning Beowulf cluster [2] to solve the musk problem [63]. For more details, see [31].

3.5.2 The Top-Down Tree-Induction Algorithm

3.5.2.1 Optimization Goal

Occam's razor stipulates that the simplest, most accurate hypothesis is to be preferred over more complex ones. The generalization bounds presented in Section 3.4 also suggest that finding the smallest, most accurate decision tree is *intuitively* a reasonable inductive bias. There are contentions, both experimental and theoretical, on the validity of such claims (see, for example, [17], [150] and [198]), but lacking a better idea, we will adopt this as the optimization goal.

3.5.2.2 Optimality

We start by stating two well-known negative results. The first, due to [97], states that, in the propositional setting, the problem of finding the smallest decision tree consistent with a training set, assuming one exists, is NP-complete. Since the propositional setting is a special case of our general setting, one can conclude that the corresponding problem for ALKEMY is also computationally hard.

[2] See `http://tux.anu.edu.au/Projects/Beowulf`.

The second observation is that the greedy top-down induction algorithm will not always return a reasonable classifier even when an accurate one exists in the hypothesis space, a weakness exposed in naked form in this next example.

Example 3.5.4. Consider the target function

$$parity : \{0,1\} \times \{0,1\} \to \{0,1\}$$
$$parity\,(t_1, t_2) = (t_1 \wedge t_2) \vee (\neg t_1 \wedge \neg t_2)$$

that computes the parity of the two input variables. Assume that the hypothesis space consists of all predicates of the form $proj_i \circ (= j)$ for $i \in \{1,2\}$ and $j \in \{0,1\}$. Given the complete set of examples

$$\{((1,1),1),((0,0),1),((0,1),0),((1,0),0)\},$$

the algorithm will return the default classifier that predicts 1 (or 0) for every individual because no meaningful first split can be found. This follows from this next fact about strictly concave functions.

Proposition 3.5.5 ([32]). *Let $H(p_1,\ldots,p_j)$ be a strictly concave function where $0 \leqslant p_i \leqslant 1$ and $\sum_i p_i = 1$. Let $\mathcal{E} = (E_1,\ldots,E_j)$ be a set of examples, where E_i denotes the subset of $\mathcal{E}$ with class label i. Let $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ be a partition of $\mathcal{E}$ where both $\mathcal{E}_1 = (E_{1,1},\ldots,E_{1,j})$ and $\mathcal{E}_2 = (E_{2,1},\ldots,E_{2,j})$ are non-empty. Denoting by $q_i = |E_i|/|\mathcal{E}|$, $r_i = |E_{1,i}|/|\mathcal{E}_1|$, and $s_i = |E_{2,i}|/|\mathcal{E}_2|$, the reduction in impurity from the partition $\mathcal{P}$ is given by*

$$H(q_1,\ldots,q_j) - \frac{|\mathcal{E}_1|}{|\mathcal{E}|}H(r_1,\ldots,r_j) - \frac{|\mathcal{E}_2|}{|\mathcal{E}|}H(s_1,\ldots,s_j) \geqslant 0$$

with equality if and only if $q_i = r_i = s_i$ for $i = 1\ldots j$.

◀

The top-down induction algorithm does, however, have the following nice property. A training set $\mathcal{E}$ free of inconsistently-labelled examples will be classified perfectly by the algorithm if the predicate space $\mathcal{H}$ is rich enough to split every impure subset of $\mathcal{E}$ (or at least those actually encountered in search) with a strict decrease in impurity. In fact, if $\mathcal{H}$ is such that significant decreases can be achieved at every stage, a small accurate tree can be expected. This well-known observation has actually been formalized in [100]. For the convenience of the reader, we restate the result here. The paper [61] contains more discussion about it.

The result was established in the PAC setting [193]. However, the authors of [100] remark that Definition 3.5.6 below can be extended to the case of probabilistic concept learning [101], and Theorem 3.5.7 continues to hold in that more general setting.

First some notation. Let T be a decision tree. We denote by $leaves(T)$ the leave nodes of T. Each $l \in leaves(T)$ has a weight $w(l)$ denoting the probability that a randomly drawn x reaches l, and a value $q(l)$ denoting the probability that $f(x) = 1$ given that x reaches l, where f is the target function. The function $H(q)$ is the node

function $TopDown(\mathcal{E}, F, N)$ **returns** a decision tree;
inputs: $\mathcal{E}$, a set of examples;
F, a class of predicates;
N, a positive integer;

T := single node (with examples $\mathcal{E}$);
for $t = 1$ to N **do**
$\Delta_{best} := 0$;
for each $(l, h) \in leaves(T) \times F$ **do**
$\Delta := H(T) - H(T(l, h))$;
if $\Delta \geqslant \Delta_{best}$ **then**
$\Delta_{best} := \Delta$; $l_t := l$; $h_t := h$;
$T := T(l_t, h_t)$;
return T;

Figure 3.5: Algorithm $TopDown(\mathcal{E}, F, N)$

impurity function, and the impurity of a tree T is defined by

$$H(T) = \sum_{l \in leaves(T)} w(l) H(q(l)).$$

If l is a leaf node of T and h is a predicate, we denote by $T(l, h)$ the tree that is the same as T except that node l is split using h. Figure 3.5 shows a variant of the top-down tree induction algorithm used in the analysis.

Definition 3.5.6 ([100]). Let f be any predicate over the input space X. Let F be any class of predicates over X. Let $\gamma \in (0, 1/2]$. We say f γ-satisfies the *Weak Hypothesis Assumption* (WHA) with respect to F if for any distribution P over X, there exists $h \in F$ such that $\mathbf{P}_{x \sim P}[h(x) \neq f(x)] \leqslant 1/2 - \gamma$.

Theorem 3.5.7 ([100]). *Let F be any class of predicates, let $\gamma \in (0, 1/2]$, and let f be any target function that γ-satisfies the WHA with respect to F. Let $\mathcal{E}$ be a set of examples, and let T be the tree output by $TopDown(\mathcal{E}, F, N)$. Then for any ϵ, the error of T on $\mathcal{E}$ is less than ϵ provided that*

$$N \geqslant \left(\frac{1}{\epsilon}\right)^{c/(\gamma^2 \epsilon^2 \log(1/\epsilon))} \quad \text{if } H(q) = 4q(1-q)$$

for some constant $c > 0$; or provided that

$$N \geqslant \left(\frac{1}{\epsilon}\right)^{c \log(1/\epsilon)/\gamma^2} \quad \text{if } H(q) = -q\log(q) - (1-q)\log(1-q)$$

function $Evaluate(p, x)$ **returns** the value of $(p\ x)$;
inputs: p, a predicate;
x, an individual;

if $ptable[p][x] \neq undefined$ **then return** $ptable[p][x]$;
$q := ancestor(p, ptable)$;
if $ptable[q][x] = 0$ **then** $ptable[p][x] := 0$;
else $ptable[p][x] := (p\ x)$;
return $ptable[p][x]$;

Figure 3.6: Predicate evaluation algorithm

for some constant $c > 0$.

In essence, the theorem states that the error on the training set approaches zero as N increases provided the WHA is satisfied. Two remarks are in order here.

First, the bound on N is independent of the size of $\mathcal{E}$. This means the true error of the output tree can be bounded using Theorem 3.4.8.

Second, the proof of Theorem 3.5.7 will not go through if the error function $H(q) = \min(q, 1-q)$ is used. This is the reason we introduce the use of entropy to break ties between equally-accurate predicates in ALKEMY. (See §3.2.2.) The original algorithms presented in [31] and [130] do not have this feature.

It is interesting to see how Definition 3.5.6 captures the conditions given earlier for the top-down induction algorithm to produce an accurate tree (on the training set). Note that a partition that satisfies the equality condition in Proposition 3.5.4 will not raise the accuracy of the original set. Under the WHA, this can never happen because we assume that the accuracy can always be improved.

Besides the development of Kearns and Mansour, there are other attempts to formalize what can be learned with the top-down induction algorithm. One example is [148]. It would be useful to clarify the relationship between the class of lookahead functions as defined in [148] and pairs (f, F) satisfying the weak hypothesis assumption as defined in Definition 3.5.6.

3.5.2.3 Time and Space Complexities

The time complexity of a simple-minded implementation of the top-down induction algorithm is just the cost of learning a stump multiplied by the number of nonterminal nodes in the output tree. There is, however, scope for improvements, especially when the computational cost is dominated by expensive predicate evaluations. Figure 3.6 gives a table-lookup algorithm to speed up this part of learning. The basic idea is to keep a record of all (or some) predicate evaluations the first time they are computed. Subsequent repeat evaluations then involve only a (relatively inexpensive) table lookup operation.

The algorithm takes as input a predicate p and an individual x and returns the value of $(p\ x)$. The data structure $ptable$ is a $\#(\rightarrowtail) \times |\mathcal{E}|$ table that stores the values of $(p\ x)$ for each p and x. It is initialized by setting every entry of top to 1 and every other entry to $undefined$. The $ancestor$ function in line 2 finds the closest ancestor of p in the predicate search space which has at least one defined entry. The third line works because for all x, $(q\ x) = 0$ implies $(p\ x) = 0$. This point has earlier been discussed in §3.5.1.3.

To realize the (conceptual) algorithm, we need to find a way to encode predicates for efficient indexing in, and retrieval from, $ptable$. We now give a simple scheme. Every predicate is encoded using a vector of non-negative integers. The root node of the search tree is encoded with the 1-dimensional vector $[1]$. The y-th child q of a predicate p is encoded with the $(n+1)$-dimensional vector $[x, y]$, where $[x]$ is the n-dimensional vector encoding for p. Under this scheme, checking whether q is a descendant of p is a simple matter of checking whether the encoding of p is a prefix of the encoding of q. Predicate indexing in $ptable$ can be done in $O(1)$ time by defining a bijection from $\mathbb{N}^n$ to $\mathbb{N}$, where n is the biggest vector dimension needed to encode the predicates. (Predicates with smaller encoding vectors are padded with zeros at the end.)

In practice, the dimension of $ptable$ is a lot smaller than $\#(\rightarrowtail) \times |\mathcal{E}|$ since a large number of predicates in $S_{\rightarrowtail}$ will usually be pruned during learning, and these won't go into $ptable$. The insertion of predicates into $ptable$ is also not orderly, a process determined completely by the search algorithm. For these reasons, $ptable$ is actually implemented as a hash map in ALKEMY. This way, memory usage can be controlled easily. Also, with a good hash function, the retrieval time would be close to $O(1)$.

Assuming $ptable$ can fit in memory, it is straightforward to show that every predicate evaluation $(p\ x)$ is computed at most once by $Evaluate$ throughout learning. This would in turn imply that the cost of learning decision trees is not much higher than the cost of learning decision stumps.

The memory requirement of this algorithm is large. One can cut the size of $ptable$ in half by storing only the smaller subtree induced by each predicate, and pay a small price for the value lookup operation. Of course, even that can be too big to fit into physical memory. In that case, one can always impose a limit on the number of predicate entries in $ptable$. This effectively turns the algorithm into a kind of caching mechanism.

A word of caution on the predicate encoding scheme. It works well with the LR search algorithm, less so with the SeenSet search algorithm. In the latter case, a predicate with multiple paths to it can be recorded multiple times in $ptable$ if it is reached via different paths at different times. The algorithm would still be correct, but memory consumption would be unnecessarily large.

The space complexity of learning without the table-lookup algorithm is dominated by the size of the open list, which is negligible. The space complexity of learning with the table-lookup algorithm, however, can be massive.

3.5.3 The Covering Algorithm

3.5.3.1 Optimization Goal

Reasoning as in §3.5.2.1, the goal here is to produce the smallest, most accurate decision list given the training examples. We'll next see how successfully this can be achieved.

3.5.3.2 Optimality

Under standard complexity-theoretic assumptions, the problem of finding the smallest, most accurate decision list is computationally difficult. This is because one can show that two special cases of the general problem are hard. Specifically, in the propositional setting,

1. the problem of computing the smallest consistent decision list given a set of examples, assuming a consistent list exists, is NP-hard (see [169]); and
2. there is no polynomial-time algorithm for the problem of computing the most accurate decision list given a set of (potentially noisy) examples unless P=NP (see §3.6.4.2).

With these in mind, we now state some properties of the $Cover$ algorithm presented in §3.2.3. We consider two cases corresponding to the misclassification cost parameter K having infinite and finite values. First some basic definitions.

We'll adopt the notation and terminology introduced in §3.2.3. Unless indicated otherwise, all decision lists referred to below are complete decision lists.

Definition 3.5.8. Given a set $\mathcal{E}$ of training examples and a decision list L, we say L is *properly labelled* with respect to $\mathcal{E}$ if every node t in L is labelled with the majority class of the subset of examples from $\mathcal{E}$ falling in t. (Ties are broken arbitrarily.)

Definition 3.5.9. Given training examples $\mathcal{E}$, the accuracy $Acc(L, \mathcal{E})$ of a decision list L on $\mathcal{E}$ is defined to be the number of examples in $\mathcal{E}$ correctly classified by L.

Definition 3.5.10. Given a set $\mathcal{E}$ of training examples and a decision list L, we define the *covered examples* of L, denoted $cov(L, \mathcal{E})$, as the examples falling in the non-default node(s) of L. Likewise, we define the *default examples* of L, denoted $default(L, \mathcal{E})$, as the examples falling in the default node of L.

Definition 3.5.11. Let $L_1 = (p_1, v_1), \ldots, (p_m, v_m)$ and $L_2 = (q_1, l_1), \ldots, (q_n, l_n)$ be two complete decision lists. We define the *extension* of L_1 by L_2, denoted $L_1 + L_2$, by

$$L_1 + L_2 = (p_1, v_1), \ldots, (p_{m-1}, v_{m-1}), (q_1, l_1), \ldots, (q_n, l_n).$$

Class 1 Decision Lists

We now investigate properties of $Cover$ when the misclassification cost K is ∞. In the following, we assume that all decision lists have node functions in some set $\mathcal{H}$ of predicates.

Definition 3.5.12. Given a set $\mathcal{E}$ of training examples, a properly labelled decision list is called a *Class 1 decision list* with respect to $\mathcal{E}$ if the subset of examples covered by each node, except perhaps the default node, is pure.

Definition 3.5.13. Given training examples $\mathcal{E}$, we say a Class 1 decision list L with respect to $\mathcal{E}$ is *maximal* if no predicate in $\mathcal{H}$ (the class of node functions) covers a non-empty pure strict subset of $default(L, \mathcal{E})$.

Definition 3.5.14. Given training examples $\mathcal{E}$, we say a Class 1 decision list L is *optimal* on $\mathcal{E}$ if no other Class 1 decision list has accuracy higher than L on $\mathcal{E}$.

Decision lists returned by $Cover$ when $K = \infty$ are, by definition, maximal Class 1 decision lists. We now show that these lists are, in fact, also optimal.

First, a simple lemma.

Proposition 3.5.15. *Let L_1 and L_2 be two Class 1 decision lists. Given training examples $\mathcal{E}$, we have*

$$Acc(L_1 + L_2, \mathcal{E}) \geqslant Acc(L_2, \mathcal{E}).$$

Proof. Let X_0 and X_0' be the number of examples labelled 0 in $default(L_2, \mathcal{E})$ and $default(L_1 + L_2, \mathcal{E})$, respectively. Similarly, let X_1 and X_1' be the number of examples labelled 1 in $default(L_2, \mathcal{E})$ and $default(L_1 + L_2, \mathcal{E})$. Further, let i be the index of the majority class in $default(L_2, \mathcal{E})$. We have the following.

$$Acc(L_2, \mathcal{E}) = |cov(L_2, \mathcal{E})| + X_i \tag{3.10}$$

$$Acc(L_1 + L_2, \mathcal{E}) = |cov(L_2, \mathcal{E})| + (X_0 - X_0') + (X_1 - X_1') + X_i' \tag{3.11}$$

To see (3.11), observe that every example in $cov(L_2, \mathcal{E})$ must fall into $cov(L_1 + L_2, \mathcal{E})$; also, some examples in $default(L_2, \mathcal{E})$ can fall into $cov(L_1, \mathcal{E})$. Given that $X_0 \geqslant X_0'$ and $X_1 \geqslant X_1'$, it's easy to verify that $Acc(L_1 + L_2, \mathcal{E}) - Acc(L_2, \mathcal{E}) \geqslant 0$. □

Let $\mathcal{E}$ be a set of training examples, and T an optimal Class 1 decision list on $\mathcal{E}$. After every iteration in the processing of $\mathcal{E}$ using $Cover$, T remains optimal on the still uncovered examples R. This is true independently of the way predicates are chosen at every step. To see this, let D be the decision list under construction. If there is a Class 1 decision list T' satisfying $Acc(T', R) > Acc(T, R)$, then we will have $Acc(D : T', \mathcal{E}) > Acc(D : T, \mathcal{E}) = Acc(T, \mathcal{E})$, contradicting the optimality of T.

Proposition 3.5.16. *Let $\mathcal{E}$ be a set of training examples and L a Class 1 decision list with respect to $\mathcal{E}$. If L is maximal, then L is optimal on $\mathcal{E}$.*

Proof. Let D be an arbitrary Class 1 decision list with respect to $\mathcal{E}$. Since L is maximal, we have $Acc(L, \mathcal{E}) \geqslant Acc(L + D, \mathcal{E}) \geqslant Acc(D, \mathcal{E})$ by Proposition 3.5.15. □

The converse is not true, of course – there are optimal Class 1 decision lists that are not maximal. This is clear since splitting a node that is already pure does not increase the accuracy.

Proposition 3.5.17. *Given training examples $\mathcal{E}$, the decision list L returned by Cover when $K = \infty$ is an optimal Class 1 decision list on $\mathcal{E}$.*

Proof. *Cover* ensures that L is a maximal Class 1 decision list. The optimality of L then follows from Proposition 3.5.16. □

Rivest shows in [169, §5.2] that the covering algorithm will always find a consistent (100% accuracy) decision list assuming one exists. Proposition 3.5.17 is, in essence, a simple generalization of that result, since a consistent decision list is clearly a Class 1 decision list with respect to the training examples.

One can state a more informative result.

Proposition 3.5.18. *Let $\mathcal{E}$ be a set of training examples, and let L_1 and L_2 be two maximal Class 1 decision lists with respect to $\mathcal{E}$. Then $cov(L_1, \mathcal{E}) = cov(L_2, \mathcal{E})$.*

Proof. Since L_1 is maximal, we have $cov(L_2, \mathcal{E}) \subseteq cov(L_1, \mathcal{E})$. A similar reasoning gives $cov(L_1, \mathcal{E}) \subseteq cov(L_2, \mathcal{E})$. □

In other words, all maximal (hence optimal) Class 1 decision lists cover exactly the same set of examples. This implies that the only difference two distinct predicate selection rules can make is in the length of the output decision lists, nothing else.

This begs the obvious question: how good is the greedy strategy? The next example shows that it can be *very* bad.

Example 3.5.19. Suppose we have training examples

$$\mathcal{E} = \{(x_1, 0), (x_2, 1), (x_3, 1), \ldots, (x_k, 1), (x_{k+1}, 1)\}.$$

Suppose also that the predicate space is $\{q, p_1, p_2, \ldots, p_k\}$, where $cov(p_i, \mathcal{E}) = \{(x_i, v_i)\}$ and $cov(q, \mathcal{E}) = \{(x_1, 0), (x_2, 1), \ldots, (x_k, 1)\}$. For large k, the list $(p_1, 0), (q, 1), (top, 0)$ is clearly the smallest optimal Class 1 decision list. But *Cover* can get very unlucky and pick the 'wrong' predicate at each step and end up with the list

$$(p_2, 1), (p_3, 1), \ldots, (p_k, 1), (q, 0), (top, 1),$$

whose size is of the order of $|\mathcal{E}|$. ◀

Doing an m-step lookahead, that is, growing the list m nodes at a time doesn't help. The same construction, with q covering m 0-labelled examples in addition to $k - m$ 1-labelled examples, shows that a big list whose size is of the order of $|\mathcal{E}|$ can again be generated.

In summary, *Cover* can find an accurate Class 1 decision list assuming one exists, but no guarantee on the size of the returned list can be given without further conditions on the training examples and the predicate space.

Remark. Example 3.5.19 answers an (unstated) question in [184]. The algorithm proposed in that paper constructs decision lists using data-dependent features generated from the training examples. Sample compression bounds [125] for these lists can be

established and the size of the list is a parameter in these bounds. The size of classifiers can be bounded for the related set-covering machine [135], but it wasn't known whether the size of decision lists generated by the algorithm in [184] can be bounded. Example 3.5.19 shows that this can't be done.

Class 2 Decision Lists

We now examine properties of $Cover$ when K is finite.

Definition 3.5.20. Given a set $\mathcal{E}$ of training examples, a properly labelled decision list is called a *Class 2 decision list* with respect to $\mathcal{E}$ if it is not a Class 1 decision list with respect to $\mathcal{E}$.

One hopes that a similar optimality result like Proposition 3.5.17 holds, but it is easy to see that this is not true.

Example 3.5.21. Suppose $K = 0.5$, and that

$$\mathcal{E} = \{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1), (x_5, 0), (x_6, 0), (x_7, 0)\}.$$

Suppose further that the predicate space contains only two predicates p and q, and

$$cov(p, \mathcal{E}) = \{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1), (x_5, 0), (x_6, 0)\};$$
$$cov(q, \mathcal{E}) = \{(x_5, 0), (x_6, 0)\}.$$

$Cover$ will return the list $L = (p, 1), (top, 0)$ with accuracy 5, but the optimal list is $L_{opt} = (q, 0), (p, 1), (top, 0)$ with accuracy 7. This construction can be adapted for arbitrary K values. ◀

In practice, we seldom need the most accurate decision list; often a reasonably accurate decision list is quite sufficient. There are experimental evidence in [135] showing that $Cover$ is good at finding high-accuracy hypotheses when K is finite.

3.5.3.3 Time and Space Complexities

Decision-list learning and decision-tree learning have similar time and space complexities and all the comments in §3.5.2.3 apply here. The main difference between the two algorithms is in the pruning mechanism. In general, one can expect pruning to be less effective in decision-list learning; somehow, covering a big largely-pure subset of examples seems harder than getting a high-accuracy partition.

3.6 PAC Learnability

In this section, we bring together results presented so far in this chapter to make some formal learnability statements in the PAC [193] and agnostic PAC [92] learning models. We are primarily interested in establishing whether the stump-, list- and tree-learning algorithms presented in Section 3.2 qualify as effective and efficient learning algorithms in the formal models.

We start with a reminder of some basic concepts in §3.6.1. Conditions for the generation of polynomially-computable predicates, a requirement for efficient learnability, are given in §3.6.2. We address the PAC-ness of the stump-, list-, and tree-learning algorithms in §3.6.3. Similar analyses in the agnostic PAC setting are given in §3.6.4.

3.6.1 PAC Learning

Let X be the set of individuals and $\mathcal{H}$ a set of predicates over X. A learning algorithm L is said to be a *probably approximately correct (PAC) learning algorithm* for $\mathcal{H}$ if it satisfies the following: given any $\epsilon, \delta \in (0,1)$, there is an integer $m(\epsilon,\delta)$ such that for all $m \geqslant m(\epsilon,\delta)$, for any $t \in \mathcal{H}$ and any probability distribution μ on X, with probability at least $1-\delta$, given a sample of size m drawn independently according to μ and labelled with t, the error of the function $h \in \mathcal{H}$ output by L with respect to t and μ defined by

$$er_\mu(h,t) = \mu\{x \in X \ : \ h(x) \neq (t\ x)\}$$

is less than ϵ. The number $m(\epsilon,\delta)$ is called the sample complexity of learning $\mathcal{H}$.

The algorithm L is said to be an *efficient PAC learning algorithm* if, in addition to the above, it runs in time polynomial in $1/\epsilon$, $1/\delta$, the size of the encoding of instances in X, and the size $size(t)$ of the encoding of the target function t. In the following, for finite function classes $\mathcal{H}$, we shall assume that $size(t) = \log|\mathcal{H}|$ for all $t \in \mathcal{H}$.

The emphasis in PAC analysis has always been on *efficient* learnability. This is because the class of all predicates over a finite domain X, which is what we have on digital computers, being finite, has a PAC learning algorithm.

Theorem 3.6.1 ([29]). *Let $\mathcal{H}$ be a finite set of predicates over X. Let L be a consistent learning algorithm for $\mathcal{H}$ in the sense that given any sample $(x_1,\ldots,x_m) \in X^m$ labelled by some $t \in \mathcal{H}$, L will output a hypothesis $h \in \mathcal{H}$ satisfying $h(x_i) = t(x_i)$ for $i = 1,\ldots,m$. Then L is a PAC learning algorithm for $\mathcal{H}$ with sample complexity*

$$m(\epsilon,\delta) \leqslant \frac{1}{\epsilon}\ln\left(\frac{|\mathcal{H}|}{\delta}\right).$$

In the PAC setting, the important question is thus not whether one can learn, but whether one can learn quickly and economically.

Remark. For learning to be possible, finiteness of $\mathcal{H}$ can be replaced with finiteness of the VC dimension of $\mathcal{H}$. Our decision to stick with finite predicate classes here is partly motivated by the results in §3.4.3, where we show that in the rich language setting of ALKEMY, the VC dimension of predicate classes is usually not too much lower than the simple upper bound given by the logarithm of the size of the predicate class.

3.6.2 Generating Efficiently-Computable Predicates

As pointed out in [46], polynomial computability of concept classes is a prerequisite for efficient PAC learning. In fact, it is defined as part of the condition for efficient

PAC learnability in [103]. This is justifiable. In [177], it is shown that if a predicate class $\mathcal{H}$ contains concepts that cannot be evaluated in polynomial time, then under reasonable complexity-theoretic assumptions, there cannot exists efficient PAC learning algorithms for $\mathcal{H}$.

We now give a sufficient condition on predicate rewrite systems that will ensure the production of only polynomially-computable predicates. Predicate classes defined on such restricted rewrite systems can then be shown to contain only predicates that can be efficiently evaluated.

In the following, the concept of an algorithm is assumed to be realized by some standard model of computation like Turing machines. Also assumed is an appropriate encoding scheme for the set of individuals X, together with a function for computing the size of the encoding of every $x \in X$, denoted $|x|$.

Definition 3.6.2. A function $r : \alpha \to \sigma$ is said to be *polynomial-time computable* if there exists an algorithm A that computes r and a polynomial $p(n)$ such that the number of steps required by A to compute $(r\ x)$ for any input $x : \alpha$ is at most $p(|x|)$.

The following is a standard result in computability theory. See, for instance, [55].

Proposition 3.6.3. *Let $f : \alpha \to \sigma$ and $g : \sigma \to \phi$ be polynomial-time computable functions. Then the function $f \circ g$ is polynomial-time computable.*

Definition 3.6.4. A transformation f having a signature of the form

$$f : (\varrho_1 \to \Omega) \to \cdots \to (\varrho_k \to \Omega) \to \alpha \to \sigma$$

is said to be *polynomial-time computable qua transformation* if $f\ p_1 \ldots p_k$ is polynomial-time computable given that each p_i, $1 \leqslant i \leqslant k$, is polynomial-time computable.

Proposition 3.6.5. *Let T be a set of transformations, and let S_T be the set of all standard predicates that can be formed using transformations in T. If every $f \in T$ is polynomial-time computable qua transformation, then every $p \in S_T$ composed of a finite number of transformations is polynomial-time computable.*

Proof. The proof is by induction on the number of transformations in p. Suppose the result holds for standard predicates that have $< m$ transformations and p has m transformations. By definition, p has the form $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$. By the inductive hypothesis, each $p_{i,j}$, $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant k_i$, is polynomial-time computable. Since each f_i, $1 \leqslant i \leqslant n$, is polynomial-time computable qua transformation, it follows that each $f_i\ p_{i,1} \ldots p_{i,k_i}$, $1 \leqslant i \leqslant n$, is polynomial-time computable. By Proposition 3.6.3, compositions of polynomial-time computable functions yield another polynomial-time computable function. □

A corollary of Proposition 3.6.5 gives the desired condition. In defining a predicate rewrite system $\rightarrowtail$, if we restrict ourselves to transformations that are polynomial-time computable qua transformation, then we can be assured that every $p \in S_{\rightarrowtail}$ is polynomial-time computable since $S_{\rightarrowtail} \subseteq S_T$ and all the predicates we will ever construct have a finite number of transformations in them.

Given a predicate rewrite system $\rightarrowtail$ composed entirely of transformations that satisfy Definition 3.6.4, it is easy to see that k-*DL*$(\rightarrowtail)$ and j/k-*DT*$(\rightarrowtail)$, the two predicate classes we actually use in ALKEMY, contain only concepts that can be evaluated in polynomial time. Computationally, the hardest predicates in k-*DL*$(\rightarrowtail)$ are the decision lists that are made up of all possible predicates in $S_{\rightarrowtail}$, located in different internal nodes. Given that $\wedge_k$ and $\neg$ are both polynomially-computable qua transformation, in the worst case, an individual x that evaluates to false for all predicates in the list will take time $O\left(\binom{\#(\rightarrowtail)}{k}|x|^r\right) = O((\#(\rightarrowtail))^k|x|^r)$ for some constant r. This is clearly a polynomial in $|x|$. A similar argument shows that the hardest functions in j/k-*DT*$(\rightarrowtail)$ take time $O(j|x|^s) = O(|x|^s)$ for some constant s.

How restrictive is the condition? Most natural transformations can be computed efficiently in the sense of Definition 3.6.4; this is true of even seemingly complex transformations. We give two examples here.

Example 3.6.6. Consider the transformation

$$(subgraphs\ k) : Graph\ v\ \epsilon \rightarrow \{Graph\ v\ \epsilon\}$$

that returns the set of all connected subgraphs of size k in a given graph. To see that $(subgraphs\ \ k)$ is polynomial-time computable qua transformation, observe that the number of connected subgraphs of size k in a graph G containing m vertices is at most $\binom{m}{k}$, which is the number of connected subgraphs of size k in a complete graph with m vertices. A simple algorithm that first generates all $\binom{m}{k} = O(m^k)$ (possibly disconnected) subgraphs of size k and then proceeds to pick out the connected ones is clearly a polynomial-time algorithm, since checking the connectivity of a graph is a polynomial-time operation. ◀

Example 3.6.7. Consider next the transformation $setExists_n$ introduced in Section 2.5. An algorithm for $setExists_n$ works as follows: Given a set x, first compute S_i, the subset of x satisfying p_i for each $i \in \{1, \ldots, n\}$. If any of the S_i's is empty, return false. If each S_i has size greater than or equal to n, return true. Otherwise, try to form a set of size n by taking an element from each S_i. Return true if such a set can be found. Return false otherwise. The first step can be computed in time polynomial in $|x|$ if each p_i is polynomial-time computable. The second step of checking the existence of a desired subset of x satisfying the definition can be done, in the worst case scenario, by enumerating all n-tuples that can be formed by taking an element from each S_i. This step requires $n^n = O(1)$ time. (Note that n is not an input to $setExists_n$, and n^n is really a constant.) This completes the argument that $setExists_n$ is polynomial-time computable qua transformation. ◀

In principle, one can write down transformations that are believed not to have polynomial-time algorithms, for example by coding well-known NP-hard problems in transformations; in practice, however, such computationally-difficult transformations are seldom, if ever, needed.

3.6.3 PAC Learnability of Stumps, Lists, and Trees

3.6.3.1 Stump Learning

Proposition 3.6.8. *Let X be the set of individuals and $\rightarrowtail$ a predicate rewrite system. The stump algorithm is a PAC learning algorithm for $1/1$-DT$(\rightarrowtail)$ with sample complexity*

$$m(\epsilon, \delta) \leqslant \frac{1}{\epsilon} \ln \left(\frac{2\#(\rightarrowtail)}{\delta} \right).$$

Proof. This is a restatement of Theorem 3.6.1. The stump algorithm is clearly a consistent learning algorithm. Further, $|1/1\text{-}DT(\rightarrowtail)| \leqslant 2\#(\rightarrowtail)$. Note here that $1/1$-DT$(\rightarrowtail)$ is the number of distinct predicates representable as a stump, *not* the number of syntactically distinct stumps. □

However, the stump algorithm is not an efficient PAC learning algorithm for the class $1/1$-DT$(\rightarrowtail)$. As we saw in §3.5.1.3, the time complexity of the algorithm is a function of $\#(\rightarrowtail)$ in the worst case. For it to qualify as an efficient PAC algorithm, the running time must be bounded by a polynomial in $size(t) \leqslant \log 2\#(\rightarrowtail)$. This is not at all surprising; one cannot expect an exhaustive search algorithm to be computationally efficient.

In principle, the class of predicates learnable with stumps is a significant class of functions since the predicate rewrite system can be made arbitrarily complex. Many results in Section 3.3, together with some of the optimization techniques presented in Section 3.5, can be used to guide the process of crafting predicate rewrite systems for learning using the stump algorithm.

3.6.3.2 List Learning

We next analyse the $Cover$ algorithm (Figure 3.4).

Proposition 3.6.9 ([169]). *Let $(X, \rightarrowtail)$ be a basic hypothesis language. Then*

$$\ln |k\text{-}DL(\rightarrowtail)| = O(\#(\rightarrowtail)^t)$$

for some constant t that is a function of k.

Proof. $|k\text{-}DL(\rightarrowtail)|$ is clearly upper-bounded by the number of syntactically-distinct decision lists that can be formed. There are $C = \sum_{i=1}^{k} \binom{2\#(\rightarrowtail)}{i}$ distinct conjunctions of at most k predicates from $S_{\rightarrowtail} \cup S_{\rightarrowtail_{neg}}$. These can be ordered in $C!$ ways, and each conjunction can either be missing, labelled 0, or labelled 1. The number $|k\text{-}DL(\rightarrowtail)|$ is thus upper-bounded by $O(3^C C!)$. Taking logs, we get $\ln(|k\text{-}DL(\rightarrowtail)|) = O(\#(\rightarrowtail)^t)$ for some constant t. □

Proposition 3.6.10 ([169]). *Let X be the set of individuals and $\rightarrowtail$ a predicate rewrite system. The algorithm Cover with $K = \infty$ is a PAC learning algorithm for k-DL$(\rightarrowtail)$ with sample*

complexity

$$m(\epsilon, \delta) \leqslant \frac{1}{\epsilon}(O(\#(\rightarrowtail)^t) + \ln \frac{1}{\delta}).$$

Proof. This follows from Theorem 3.6.1, Proposition 3.5.17, and Proposition 3.6.9. □

Proposition 3.6.9 gives us the number of distinct representations of functions in k-*DL*$(\rightarrowtail)$. If we use a (suboptimal) coding scheme based on the number of distinct representations in k-*DL*$(\rightarrowtail)$ (as opposed to the number of distinct functions in k-*DL*$(\rightarrowtail)$), which we may have to do since the equivalence of two decision lists cannot be decided in polynomial time unless P=NP (see [89]), then we may assume that for all $t \in k$-*DL*$(\rightarrowtail)$, $size(t)$ is a polynomial in $\#(\rightarrowtail)$. Under this assumption, if the predicate rewrite system $\rightarrowtail$ is made up of only transformations that satisfy Definition 3.6.4 and that the sample complexity of learning k-*DL*$(\rightarrowtail)$ is a polynomial in the size of the encoding of instances in X, then $Cover$ with $K = \infty$ is in fact an efficient PAC learning algorithm for k-*DL*$(\rightarrowtail)$ since it runs in time polynomial in $\#(\rightarrowtail)$ and $m(\epsilon, \delta)$.

Given $(X, \rightarrowtail)$, k-*DL*$(\rightarrowtail)$ is one of the largest subset of $\mathbb{BF}(X, \rightarrowtail)$ for which an efficient PAC learning algorithm exists. As indicated in Section 3.3, many predicate classes, some of which are defined independently of decision lists, have been shown to be a subset of k-*DL*$(\rightarrowtail)$, and therefore efficiently learnable using the $Cover$ algorithm with appropriate enrichment of $\rightarrowtail$.

3.6.3.3 Tree Learning

Example 3.5.4 shows that the top-down induction algorithm cannot be a PAC learning algorithm for j/k-*DT*$(\rightarrowtail)$. It turns out that this algorithm can be analysed in the weak learning framework. Under the weak hypothesis assumption (see Definition 3.5.6), Theorem 3.5.7 shows that the algorithm can construct a decision tree with arbitrarily low error on the training examples, and this is sufficient to guarantee PAC learnability. This development is sketched in §3.5.2.2; the reader is referred to [100] for more details.

Another relevant piece of work on PAC-learnability of decision trees is [34].

3.6.4 Agnostic PAC Learning

We now consider the learnability of Alkemic function classes in the more realistic agnostic PAC setting [92]. In the agnostic PAC setting, we do not presuppose the existence of a target function but assume that examples are generated independently according to an unknown probability distribution P on $X \times \{0, 1\}$. The aim of learning is similar, but instead of trying to find a function in the predicate space $\mathcal{H}$ that approximates the target function arbitrarily well, we seek a function in $\mathcal{H}$ that is arbitrarily close to the function $t^* \in \mathcal{H}$ that has the smallest true error with respect to P.

The correct (and, in fact, only) strategy in the agnostic PAC setting is simple: find the $h \in \mathcal{H}$ that achieves the lowest empirical error on the training examples. (See, for details, [3, Chap. 23] or [92].) An algorithm that can achieve this goal for a predicate

class $\mathcal{H}$ is called an agnostic PAC learning algorithm for $\mathcal{H}$. If, in addition, the algorithm runs in time polynomial in the usual key parameters, then it is said to be an *efficient* agnostic PAC learning algorithm for $\mathcal{H}$.

3.6.4.1 Stumps

The stump algorithm conducts an exhaustive search of the predicate space and finds the one with the lowest empirical error. It is therefore an agnostic PAC learning algorithm. However, it is *not* an efficient agnostic PAC learning algorithm.

3.6.4.2 Lists

We have seen that the *Cover* algorithm is not particularly effective at achieving the goal of finding the decision list with the smallest empirical error given a set of training examples. In fact, this is to be expected since one can show that the problem of agnostic PAC learning k-$DL(\rightarrowtail)$ is computationally hard. This is because we can show that there is no efficient algorithm for the corresponding problem in the propositional setting, which is a special case of the general problem, unless P=NP.

The argument is an adaptation of the proof for [3, Thm 24.2]. Consider the following two decision problems.

1. VERTEX-COVER
   Instance: A graph $G = (V, E)$ and an integer $k \leqslant |V|$.
   Question: Is there a vertex cover $U \subseteq V$ such that $|U| \leqslant k$?

2. DL-FIT
   Instance: $z \in (\{0,1\}^n \times \{0,1\})^m$ and an integer k between 1 and m.
   Question: Is there $h \in$ 1-$DL(n)$ such that $\hat{er}(h, z) \leqslant k/m$?

In the definition of DL-FIT, 1-$DL(n)$ is the class of decision lists (see the beginning of §3.2.3) over $\{0,1\}^n$ with node functions in $\mathcal{H} = \{top\} \cup \{h_{i,j} : 1 \leqslant i \leqslant n, j \in \{0,1\}\}$ where each $h_{i,j} : \{0,1\}^n \rightarrow \{0,1\}$ is defined by

$$h_{i,j}(x) = \begin{cases} 1 & \text{if } x_i = j \\ 0 & \text{otherwise.} \end{cases}$$

Further, $\hat{er}(h, z)$ is defined to be $|\{(x, y) \in z : h(x) \neq y\}|/m$, in other words, the number of errors h makes on the set z of examples.

It is known that VERTEX-COVER is NP-hard [81]. We now show that given an instance of VERTEX-COVER, we can construct (in polynomial time) an instance of DL-FIT (of a size polynomially related to that of the instance of VERTEX-COVER) in such a way that the answer to the constructed DL-FIT problem is the same as the answer to the original VERTEX-COVER problem.

Consider an instance $G = (V, E)$ of VERTEX-COVER where $|V| = n$ and $|E| = r$. We assume that each vertex in V is labelled with a number from $\{1, 2, \ldots, n\}$ and we denote by ij an edge in E connecting vertex i and vertex j. The size of the instance is

$\Omega(r+n)$. We construct $z(G) \in (\{0,1\}^n \times \{0,1\})^{r+n}$ as follows. For any two integers i, j between 1 and n, let $e_{i,j}$ denote the binary vector of length n with ones in positions i and j and zeroes everywhere else. The sample $z(G)$ consists of the labelled examples $(e_{i,i}, 1)$ for $i = 1, 2, \ldots, n$ and, for each edge $ij \in E$, the labelled example $(e_{i,j}, 0)$. The size of z is $(r+n)(n+1)$, which is polynomial in the size of the original VERTEX-COVER instance.

Example 3.6.11. Consider the graph $G = \{\{1,2,3,4\}, \{11,12,13,14,23,33\}\}$. Then

$$z(G) = \{(1000,1), (0100,1), (0010,1), (0001,1),\\ (1000,0), (0010,0), (1100,0), (1010,0), (1001,0), (0110,0)\}.$$

Proposition 3.6.12. *Given any graph $G = (V, E)$ with n vertices and r edges and any integer $k \leqslant n$, let $z(G)$ be as defined above. Then there is $h \in$ 1-$DL(n)$ such that $\hat{er}(h, z(G)) \leqslant k/(n+r)$ if and only if there is a vertex cover of G of cardinality at most k.*

The proof of Proposition 3.6.12 makes use of the following result.

Proposition 3.6.13 ([2]). *1-$DL(n) \subseteq LT(n)$.*

Here, $LT(n)$ is the class of linear threshold functions over $\{0,1\}^n$. Formally, $LT(n)$ is the set of of all functions from $\{0,1\}^n$ to $\{0,1\}$ that can be represented in the form

$$f(x) = \text{sgn}\left(\sum_{i=1}^{n} w_i x_i - b\right)$$

where $w_i, b \in \mathbb{R}$ and sgn(x) evaluates to 1 if $x \geq 0$, and 0 otherwise.

Proof. (of Proposition 3.6.12)
$(\rightarrow)$ Suppose there is such an $h \in$ 1-$DL(n)$. By Proposition 3.6.13, there is an $h' \in LT(n)$ that is equivalent to h. We represent h' by its weights $w = (w_1, w_2, \ldots, w_n, b)$. We construct a subset U of V as follows.

1. For each $(e_{i,i}, y) \in z(G)$, if $h'(e_{i,i}) = 0$, then include i in U.
2. For each $(e_{i,j}, 0) \in z(G)$, $i \neq j$, if $h'(e_{i,j}) = 1$, then include either one of i, j in U.

The set U so-constructed contains at most k vertices since $\hat{er}(h', z(G)) = \hat{er}(h, z(G)) \leqslant k/(n+r)$. We now show that U is a vertex cover for G. Consider an arbitrary edge ij in E. If either $h'(e_{i,i}) = 0$ or $h'(e_{j,j}) = 0$ then we're done. Suppose not, that is, $h'(e_{i,i}) = h'(e_{j,j}) = 1$. Then we may deduce that

$$w_i \geqslant b \text{ and } w_j \geqslant b.$$

This implies that $h'(e_{i,j}) = 1$. Because of the way U is constructed, it follows that at least one of the vertices i, j is in U. Since ij is an arbitrary edge, we conclude that U is indeed a vertex cover.

($\leftarrow$) Suppose $U = \{i_1, \ldots, i_{|U|}\} \subseteq V$ is a vertex cover of G and $|U| \leqslant k$. We define $h \in$ 1-$DL(n)$ to be

$$h = (h_{i_1,1}, 0), \ldots, (h_{i_{|U|},1}, 0), (top, 1).$$

We claim that $\hat{er}(h, z(G)) \leqslant k/(n+r)$. Observe that if $ij \in E$, then since U is a vertex cover, one of i, j belongs to U and thus $h(e_{i,j}) = 0$. This means that all the examples in $z(G)$ arising from the edges of G are correctly classified. Consider now examples in $z(G)$ of the form $(e_{i,i}, 1)$. We have $h(e_{i,i}) = 0$ if $i \in U$ and $h(e_{i,i}) = 1$ otherwise. It follows that

$$\hat{er}(h, z(G)) = \frac{|U|}{n+r} \leqslant \frac{k}{n+r}.$$

□

Given that $z(G)$ can be computed from G in time polynomial in the size of G, we have therefore established the following.

Proposition 3.6.14. DL-FIT *is NP-hard.*

Now, if there is an algorithm that, given a set $\mathcal{E}$ of examples, computes in polynomial time $\arg\min_{h \in \text{1-}DL(n)} \hat{er}(h, \mathcal{E})$, then it can be used to solve DL-FIT in polynomial time. By Proposition 3.6.14, such an algorithm cannot exists unless P=NP.

3.6.4.3 Trees

The efficient learnability of decision trees remains one of the longest-standing open problems in computational learning theory; see [99]. We will give some relevant references here.

There are hardness results on the problem of computing the *smallest* decision trees given training examples; see [97] and [54]. It is not known whether the problem of computing the most accurate decision tree given a set of training examples is hard.

An efficient agnostic PAC learning algorithm is given in [10] for small (depth-two) decision trees. The algorithm given in the paper conducts an exhaustive search of the tree space in a clever way.

3.6.5 Learning In Practice

> This is all very fine in theory but almost worthless in practice.
> *Donald Knuth [112]*

The study of PAC and agnostic PAC learnability has yielded mostly negative results. This is hardly surprising given the stringent definition of what qualifies as successful learning. Information-theoretically, a lot of examples may be needed for valid generalization under *arbitrary* distribution. Computationally, the implied strategy of finding the function with the lowest empirical error is hard for most non-trivial function classes.

In practice, we seldom need to find the best possible hypothesis. The normal scenario is that we are given a set of training examples and required to come up with one reasonably-accurate hypothesis (not necessarily the best one). To accomplish our task, we have at our discretion the choice of the hypothesis language, the function class, and the learning algorithm. The tools we actually need are:

1. error bounds that are tight and easily computable;
2. principled model selection strategies; and
3. a good understanding of phenomena surrounding small-sample learning.

Significant progress has been achieved in all three areas, although much more work needs to be done for the techniques and tools developed to be widely-applicable. For advances in error bounds, see [14], [119] and [120]. For studies in model selection, see [102] and [13]. For work on small-sample learning, see [36] and [113].

3.7 Related Work

On the learning setting By design, learning in our higher-order logic setting can be readily understood as a direct generalization of attribute-value learning. As shown throughout the chapter, this similarity in nature between the two allows many results and algorithms in propositional learning to be immediately applicable in our setting. In that sense, learning in higher-order logic is closely related to the learning from interpretations [59] setting in ILP. In both frameworks, there is a clear separation between descriptions of different individuals. (In the more standard learning from entailment [144] setting, training data and background knowledge are all lumped together in one logic program.) This separation of information admits simpler analysis and, more importantly, makes tractable learning possible. (Compare the positive result of [59] with the negative PAC-learning results for the learning from entailment setting. See also [23, §4.7] and [57].) In the learning from interpretations setting, there is a price in separating examples from one another in that recursive predicates cannot be learned. In the higher-order setting, this limitation can be overcome by introducing into the hypothesis language higher-order functions like `foldr` that package up recursion into convenient forms.

On structural decision tree learning There are quite a few decision-tree systems that learn from structured data of one form or another. These include KATE [133], RIPPER [45], STRUCT [196], S-CART [118], TILDE ([24], [23]), TRITOP and its predecessor INDIGO ([85], [84]), and the LINUS/DINUS [122] family of learners.

KATE is probably the first decision-tree system that learns from structured data. It uses a frame-based language that is equivalent to first-order predicate calculus.

RIPPER works with an attribute-value language that is extended with set-valued features. The language used is a strict subset of the language of ALKEMY.

The underlying language for TILDE and S-CART is Prolog. In that setting, training examples, background knowledge, and the tree induced are all logic programs. Vari-

ables are shared across decision nodes down true branches in trees induced by these two systems. For that reason, the search space can change from node to node.

ALKEMY is closest in nature to the class of algorithms that learn from propositionalized knowledge. Such systems include TRITOP and the LINUS/DINUS family of learners. For more information on propositionalization approaches to structure learning, see [117], [116] and [115].

On sample complexity bounds Lower bounds on the number of examples required for learning have been studied in ILP. See [8], [104] and [7]. The results presented in §3.4.3 are closely related to these work. The fact that the same general conclusion was obtained from the analyses of two very different knowledge representation formalisms tells us something about the sample complexity of learning with rich expressive languages in general.

On PAC-learnability There is a body of work on the PAC-learnability (and non-PAC-learnability) of different classes of first-order logic programs. (Not much work has been done in the more general agnostic PAC setting, however.) Positive results are usually obtained by analyzing concrete algorithms for variously-restricted classes of logic programs. Negative results, in turn, are usually shown by reducing known difficult problems like 3-SAT and the PAC-predictability of boolean formulae in disjunctive normal form to different aspects of learning syntactically-restricted logic programs. See, for a survey, [145], [70], [105], [43], [44], [96] and [6]. All these analyses have a strong *syntactic* flavour to them, and the arguments are usually intimately and intricately linked to the computational model of first-order logic programming. It is not clear whether these results, which reflect the nature of learning with a first-order language like Prolog, reflect the nature of learning with rich expressive languages in general. Results presented in Section 3.6 represent a useful step in our attempt to understand the learnability of predicate classes in a higher-order logic that uses equational reasoning as its basic computational model. We hope this kind of analysis, with more development and subsequent comparison with results in ILP, will lead to better understanding of the nature of learning with rich expressive languages in general.

Miscellaneous The separation of work in supervised learning into issues of approximation, estimation and computation is popular in learning theory. See, for example, [11] and [3, §1.1]

Chapter 4

Regression

4.1 Introduction

Besides (binary) classification, regression is another fundamental task in supervised machine learning, with wide-ranging applications. Extending ALKEMY to do regression is the subject matter of this chapter.

The basic learning problem is simple to state: Given training examples

$$((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)) \in (X, \mathbb{R})^m$$

where X is the individual space, find a function $\hat{f} : X \to \mathbb{R}$ (in the form of a regression tree) that approximates the underlying function $f(x) = \mathbb{E}(y|x)$ as well as possible.

The learning algorithms are presented in Section 4.2. Generalization issues are then discussed in Section 4.3. This chapter is considerably shorter than Chapter 3. This is partly because many of the approximation and optimization results presented in Chapter 3 carry over with little change to the regression setting and they are not repeated here.

4.2 Learning Algorithms

Two algorithms are presented, the first a special case of the second. We start with the simpler problem of learning regression stumps in §4.2.1. Regression-tree learning is then presented in §4.2.2 as a recursive algorithm built around the stump algorithm. The basic learning procedure is standard (see, for example, CART [32]) and we claim no originality in that part. The main contribution here is the derivation of a predicate search space pruning result in §4.2.1.2.

4.2.1 Learning Stumps

We now present an algorithm that takes as input

1. a training set $z \in (X \times \mathbb{R})^m$ of arbitrary size m,
2. a predicate rewrite system $\rightarrowtail$ defining predicates over X,

and produces as output a hypothesis $f : X \to \mathbb{R}$ of the form

$$f(x) = \text{if } (p\ x) \text{ then } c_1 \text{ else } c_2 \tag{4.1}$$

where $p \in S_{\rightarrowtail}$, and $c_1, c_2 \in \mathbb{R}$. Such a rule is called a regression stump.

Given training examples $\mathcal{E}$, informally, the aim is to partition $\mathcal{E}$ into two clusters $\mathcal{E}_1$ and $\mathcal{E}_2$ such that the distances between individuals in different clusters are big, and the distances between individuals in the same cluster are small. This notion is made precise in §4.2.1.1. We then give a search space pruning result in §4.2.1.2. Putting the pieces together gives us the learning algorithm in §4.2.1.3.

4.2.1.1 Predicate Selection

Given training examples $\mathcal{E}$ and a predicate rewrite system $\rightarrowtail$, each predicate p in $S_{\rightarrowtail}$ partitions $\mathcal{E}$ into two subsets:

$$\mathcal{E}_1 = \{(x, y) \in \mathcal{E} \mid (p\ x)\} \text{ and}$$
$$\mathcal{E}_2 = \{(x, y) \in \mathcal{E} \mid \neg(p\ x)\}.$$

In coming up with a predicate selection rule, we need to know the regression function f_p associated with p. It has the general form given in (4.1), but what values should c_1 and c_2 take? In other words, how should $\mathcal{E}_1$ and $\mathcal{E}_2$ be labelled?

A convenient loss function in the regression setting is the quadratic loss. Adopting it here, the loss of f_p on $\mathcal{E}$ can be written as

$$\sum_{(x,y)\in\mathcal{E}_1} (y - c_1)^2 + \sum_{(x,y)\in\mathcal{E}_2} (y - c_2)^2.$$

The expression is minimized when c_1 and c_2 are equal to the averages of the regression values in $\mathcal{E}_1$ and $\mathcal{E}_2$, and this implies that $\mathcal{E}_1$ and $\mathcal{E}_2$ should be labelled as such.

Definition 4.2.1. Given a set of examples $\mathcal{E}$, we define the *squared error $E_{\mathcal{E}}$ of $\mathcal{E}$* by

$$E_{\mathcal{E}} = \min_c \sum_{(x,y)\in\mathcal{E}} (y - c)^2.$$

Definition 4.2.2. Let $\mathcal{E}$ be a set of examples, p a predicate, and $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ the partition of $\mathcal{E}$ induced by p. We define the *squared error Q_p of p* by

$$Q_p = Q_{\mathcal{P}} = E_{\mathcal{E}_1} + E_{\mathcal{E}_2} = \sum_{(x,y)\in\mathcal{E}} (f_p(x) - y)^2$$

Given a predicate rewrite system $\rightarrowtail$ and a set of examples $\mathcal{E}$, the goal is thus to find a predicate in $S_{\rightarrowtail}$ that minimizes Q. As pointed out in [3, §16.1], this formulation is well-founded since one can show that

$$\mathbb{E}(f(x) - y)^2 = \mathbb{E}(\mathbb{E}(y|x) - f(x))^2 + \mathbb{E}(\mathbb{E}(y|x) - y)^2,$$

which implies that choosing a function f to minimize Q is equivalent to finding the best approximation of the conditional expectation of y given x. See, for more details, [32, §8.3] and [20, Chap. 7].

Before proceeding, we state here some simple properties of Q and E.

Proposition 4.2.3. *Let $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ be a partition of a set $\mathcal{E}$ of examples. Then $E_{\mathcal{E}} \geqslant Q_{\mathcal{P}}$.*

Proof. We have

$$E_{\mathcal{E}} = \sum_{\mathcal{E}_1}(y-\bar{y})^2 + \sum_{\mathcal{E}_2}(y-\bar{y})^2 \geqslant \sum_{\mathcal{E}_1}(y-\bar{y}_1)^2 + \sum_{\mathcal{E}_2}(y-\bar{y}_2)^2 = Q_{\mathcal{P}}$$

where $\bar{y}$, $\bar{y}_1$ and $\bar{y}_2$ are the empirical means of the regression values in $\mathcal{E}$, $\mathcal{E}_1$ and $\mathcal{E}_2$. □

Proposition 4.2.4. *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be sets of examples. Let p be a predicate and $\mathcal{P}_p(\mathcal{E}_1)$ and $\mathcal{P}_p(\mathcal{E}_2)$ the partitions of $\mathcal{E}_1$ and $\mathcal{E}_2$ induced by p.*

1. *If $\mathcal{E}_1 \subseteq \mathcal{E}_2$, then $E_{\mathcal{E}_1} \leqslant E_{\mathcal{E}_2}$.*
2. *If $\mathcal{E}_1 \subseteq \mathcal{E}_2$, then $Q_{\mathcal{P}_p(\mathcal{E}_1)} \leqslant Q_{\mathcal{P}_p(\mathcal{E}_2)}$.*

Proof.

1. Let $\bar{y}_1, \bar{y}_2$ be the empirical means of the regression values in $\mathcal{E}_1$ and $\mathcal{E}_2$. Then

$$\begin{aligned} E_{\mathcal{E}_2} &= \sum_{(x,y)\in\mathcal{E}_2}(y-\bar{y}_2)^2 \\ &= \sum_{(x,y)\in\mathcal{E}_1}(y-\bar{y}_2)^2 + \sum_{(x,y)\in\mathcal{E}_2\setminus\mathcal{E}_1}(y-\bar{y}_2)^2 \\ &\geqslant \sum_{(x,y)\in\mathcal{E}_1}(y-\bar{y}_1)^2 + \sum_{(x,y)\in\mathcal{E}_2\setminus\mathcal{E}_1}(y-\bar{y}_2)^2 \geqslant E_{\mathcal{E}_1}. \end{aligned}$$

2. Let $\mathcal{P}_p(\mathcal{E}_1) = (\mathcal{E}_{11}, \mathcal{E}_{12})$ and $\mathcal{P}_p(\mathcal{E}_2) = (\mathcal{E}_{21}, \mathcal{E}_{22})$. Clearly, $\mathcal{E}_{11} \subseteq \mathcal{E}_{21}$ and $\mathcal{E}_{12} \subseteq \mathcal{E}_{22}$. The result then follows by Part (1).

□

4.2.1.2 Predicate Pruning

To search through the space of predicates efficiently, we need a way to do predicate pruning. To do predicate pruning, we need a way to predict the smallest error that can be obtained by a refinement of a partition $\mathcal{P}$. This motivates the next definition.

Definition 4.2.5. Let $\mathcal{E}$ be a set of examples and $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ a partition of $\mathcal{E}$. We define the *regression refinement bound* of $\mathcal{P}$ by

$$C_{\mathcal{P}} = \min_{\mathcal{P}'} Q_{\mathcal{P}'}$$

where $\mathcal{P}'$ is a refinement of $\mathcal{P}$. A refinement $\mathcal{P}^*$ of $\mathcal{P}$ that satisfies $Q_{\mathcal{P}^*} = C_{\mathcal{P}}$ is called a *minimizing refinement* of $\mathcal{P}$.

function $RegRefinementBound(\mathcal{P})$ **returns** $C_{\mathcal{P}}$;
input: $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$, a partition of a set of examples;

if $\mathcal{E}_1 = \emptyset$ **return** $E_{\mathcal{E}_2}$;
$sort(\mathcal{E}_1)$;
$minerr := Q_{\mathcal{P}}$;
for each i **from** 1 **to** $|\mathcal{E}_1|$ **do**

$\mathcal{E}_{11} := \mathcal{E}_1[1, i]$;
$\mathcal{E}_{12} := \mathcal{E}_1[i+1, |\mathcal{E}_1|]$;
$\mathcal{P}_1 := (\mathcal{E}_{11}, \mathcal{E}_{12} \cup \mathcal{E}_2)$;
$\mathcal{P}_2 := (\mathcal{E}_{12}, \mathcal{E}_{11} \cup \mathcal{E}_2)$;
$minerr := \min\{minerr, Q_{\mathcal{P}_1}, Q_{\mathcal{P}_2}\}$

return $minerr$;

Figure 4.1: Algorithm for calculating $C_{\mathcal{P}}$

The following is the regression analogue of Proposition 3.2.9.

Proposition 4.2.6. *Let $\mathcal{E}$ be a set of examples and $\mathcal{P}$ a partition of $\mathcal{E}$. If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $Q_{\mathcal{P}'} \geqslant C_{\mathcal{P}}$. In particular, $Q_{\mathcal{P}} \geqslant C_{\mathcal{P}}$.*

Proof. By the definition of $C_{\mathcal{P}}$. □

We now examine the efficient computability of $C_{\mathcal{P}}$. A lower bound for $C_{\mathcal{P}}$ can be easily obtained.

Proposition 4.2.7. *Let $\mathcal{E}$ be a set of examples and $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ a partition of $\mathcal{E}$. Then $E_{\mathcal{E}_2} \leqslant C_{\mathcal{P}}$. In particular, when $\mathcal{E}_1 = \emptyset$, $E_{\mathcal{E}_2} = C_{\mathcal{P}}$.*

Proof. Let $\mathcal{P}^* = (\mathcal{E}_1^*, \mathcal{E}_2^*)$ be a minimizing refinement of $\mathcal{P}$. By Part (1) of Proposition 4.2.4, we have $E_{\mathcal{E}_2} \leqslant E_{\mathcal{E}_2^*}$ since $\mathcal{E}_2 \subseteq \mathcal{E}_2^*$. From that, we have

$$Q_{\mathcal{P}^*} = E_{\mathcal{E}_1^*} + E_{\mathcal{E}_2^*} \geqslant E_{\mathcal{E}_2^*} \geqslant E_{\mathcal{E}_2}.$$

Clearly, when $\mathcal{E}_1 = \emptyset$, $\mathcal{E}_1^* = \emptyset$ and $\mathcal{E}_2^* = \mathcal{E}_2$. □

Figure 4.1 shows an algorithm that, given a partition $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$, finds the value of $C_{\mathcal{P}}$. In the algorithm, $\mathcal{E}[i, j]$ denotes the subset of $\mathcal{E}$ formed from taking the i-th to j-th element(s). If $i > j$, we define $\mathcal{E}[i, j]$ to be $\emptyset$.

The $sort$ function in line 4 in Figure 4.1 is with respect to the following total order $\preccurlyeq$ on the examples. Examples are first ordered increasingly by their regression values. Examples with the same regression values are then ordered according to a lexicographic order on the individuals. We denote by $\max_{(x,y)}(\mathcal{E}, \preccurlyeq)$ and $\min_{(x,y)}(\mathcal{E}, \preccurlyeq)$ the largest and smallest examples in $\mathcal{E}$ as ordered by $\preccurlyeq$.

We now show the value returned by $RegRefinementBound$ on input $\mathcal{P}$ is $C_{\mathcal{P}}$. First, a technical lemma.

Proposition 4.2.8. *Let $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$, $\mathcal{E}_1 \neq \emptyset$, be a partition of a set $\mathcal{E}$ of examples. Let $\mathcal{P}_{min} = (\mathcal{E}_{11}, \mathcal{E}_{12} \cup \mathcal{E}_2)$ be a minimizing refinement of $\mathcal{P}$, where $\mathcal{E}_{11} \cup \mathcal{E}_{12} = \mathcal{E}_1$. Then $\mathcal{P}_{min}$ must satisfy the following property:*

1. $\forall (x_1, y_1) \in \mathcal{E}_{11}, \forall (x_2, y_2) \in \mathcal{E}_{12},\ y_1 \leqslant y_2$; *or*
2. $\forall (x_1, y_1) \in \mathcal{E}_{12}, \forall (x_2, y_2) \in \mathcal{E}_{11},\ y_1 \leqslant y_2$.

Proof. If one of $\mathcal{E}_{11}$ and $\mathcal{E}_{12}$ is empty, then the property holds trivially. Now consider the case when both $\mathcal{E}_{11}$ and $\mathcal{E}_{12}$ are non-empty. Suppose the property does not hold. Then there exist (x_1, y_1) in $\mathcal{E}_{11}$ and (x_2, y_2) in $\mathcal{E}_{12}$ such that $y_1 > y_2$, and (x_3, y_3) in $\mathcal{E}_{12}$ and (x_4, y_4) in $\mathcal{E}_{11}$ such that $y_3 > y_4$. From that we get

$$\max_y(\mathcal{E}_{11}) \geqslant y_1 > y_2 \geqslant \min_y(\mathcal{E}_{12}); \text{ and} \tag{4.2}$$

$$\max_y(\mathcal{E}_{12}) \geqslant y_3 > y_4 \geqslant \min_y(\mathcal{E}_{11}). \tag{4.3}$$

Now we show that we can always pick (x', e') from $\mathcal{E}_{11}$ and (x'', e'') from $\mathcal{E}_{12}$ and interchange them to produce another refinement of $\mathcal{P}$ with a lower error, thus contradicting the minimality of $\mathcal{P}_{min}$.

Let $\bar{y}_1$ and $\bar{y}_2$ denote, respectively, the empirical means of the regression values in $\mathcal{E}_{11}$ and $\mathcal{E}_{12} \cup \mathcal{E}_2$. There are two cases to consider, and we state the elements we choose in each case.

1. If $\bar{y}_1 \leqslant \bar{y}_2$, pick (x', e') to be $\max_{(x,y)}(\mathcal{E}_{11}, \preccurlyeq)$, and (x'', e'') to be $\min_{(x,y)}(\mathcal{E}_{12}, \preccurlyeq)$. We have $e' > e''$ from (4.2).
2. If $\bar{y}_1 > \bar{y}_2$, pick (x', e') to be $\min_{(x,y)}(\mathcal{E}_{11}, \preccurlyeq)$, and (x'', e'') to be $\max_{(x,y)}(\mathcal{E}_{12}, \preccurlyeq)$. We have $e'' > e'$ from (4.3).

Now let $\mathcal{E}'_1 = \mathcal{E}_{11} \cup \{e''\} \setminus \{e'\}$ and $\mathcal{E}'_2 = \mathcal{E}_2 \cup \mathcal{E}_{12} \cup \{e'\} \setminus \{e''\}$. Clearly, $\mathcal{P}' = (\mathcal{E}'_1, \mathcal{E}'_2)$ is a refinement of $\mathcal{P}$. We have

$$\begin{aligned}
Q_{\mathcal{P}_{min}} &= \sum_{(x,y)\in\mathcal{E}_{11}} (y - \bar{y}_1)^2 + \sum_{(x,y)\in\mathcal{E}_2\cup\mathcal{E}_{12}} (y - \bar{y}_2)^2 \\
&= (e' - \bar{y}_1)^2 + \sum_{(x,y)\in\mathcal{E}_{11}\setminus\{e'\}} (y - \bar{y}_1)^2 + (e'' - \bar{y}_2)^2 + \sum_{(x,y)\in\mathcal{E}_2\cup\mathcal{E}_{12}\setminus\{e''\}} (y - \bar{y}_2)^2 \\
&\geqslant (e'' - \bar{y}_1)^2 + \sum_{(x,y)\in\mathcal{E}_{11}\setminus\{e'\}} (y - \bar{y}_1)^2 + (e' - \bar{y}_2)^2 + \sum_{(x,y)\in\mathcal{E}_2\cup\mathcal{E}_{12}\setminus\{e''\}} (y - \bar{y}_2)^2 \\
&> Q_{\mathcal{P}'}.
\end{aligned}$$

We can make the third step because

$$(e' - \bar{y}_1)^2 + (e'' - \bar{y}_2)^2 - [(e'' - \bar{y}_1)^2 + (e' - \bar{y}_2)^2] = 2(e'' - e')(\bar{y}_1 - \bar{y}_2) \geqslant 0$$

in both cases. The fourth step follows because $e' \neq e''$. (Recall Definition 4.2.1.1 and the remark preceding it.) □

Proposition 4.2.8 assures us that in trying to compute $C_{\mathcal{P}}$, we can restrict our attention to refinements that are obtainable by cutting a sorted version of $\mathcal{E}_1$ into two halves, and assigning one or the other to the new $\mathcal{E}_2$.

Proposition 4.2.9. *Given a partition $\mathcal{P}$, RegRefinementBound correctly finds $C_{\mathcal{P}}$.*

Proof. Let $\mathcal{P}$ be $(\mathcal{E}_1, \mathcal{E}_2)$. There are two cases. If $\mathcal{E}_1 = \emptyset$, the algorithm returns $E_{\mathcal{E}_2}$, which is equal to $C_{\mathcal{P}}$ by Proposition 4.2.7. If $\mathcal{E}_1$ is non-empty, then one of the minimizing refinements must satisfy the property stated in Proposition 4.2.8. The algorithm conducts an exhaustive search of all such partitions and must therefore find $C_{\mathcal{P}}$. □

Since the *RegRefinementBound* algorithm needs to be invoked very frequently during search, once for every predicate we test, it is necessary to minimize its computation time. The algorithm, when implemented naïvely, has time complexity $O(|\mathcal{E}|^2)$, since there are $O(|\mathcal{E}|)$ iterations, and the computation of $Q_{\mathcal{P}_1}$ and $Q_{\mathcal{P}_2}$ in each iteration takes $O(|\mathcal{E}|)$ time. This can be significantly improved. We now give an implementation of *RegRefinementBound* that runs in time linear in the size of $\mathcal{E}$.

Proposition 4.2.10. *Given a set of examples $\mathcal{E}$ and a partition $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ of $\mathcal{E}$, assuming $\mathcal{E}_1$ is sorted according to $\preccurlyeq$ a priori, $C_{\mathcal{P}}$ can be computed in time $O(|\mathcal{E}|)$.*

Proof. Let $\bar{y}, \bar{y}_1, \bar{y}_2$ denote the empirical means of the regression values in $\mathcal{E}, \mathcal{E}_1$ and $\mathcal{E}_2$. We can rewrite the error function as follows:

$$
\begin{aligned}
Q_{\mathcal{P}} &= \sum_{(x,y)\in\mathcal{E}_1} (y-\bar{y}_1)^2 + \sum_{(x,y)\in\mathcal{E}_2} (y-\bar{y}_2)^2 \\
&= \sum_{(x,y)\in\mathcal{E}_1} [(y-\bar{y}) - (\bar{y}_1-\bar{y})]^2 + \sum_{(x,y)\in\mathcal{E}_2} [(y-\bar{y}) - (\bar{y}_2-\bar{y})]^2 \\
&= \sum_{(x,y)\in\mathcal{E}_1} [(y-\bar{y})^2 - (\bar{y}_1-\bar{y})^2] + \sum_{(x,y)\in\mathcal{E}_2} [(y-\bar{y})^2 - (\bar{y}_2-\bar{y})^2] \\
&= \sum_{(x,y)\in\mathcal{E}_1} (y-\bar{y})^2 - |\mathcal{E}_1|(\bar{y}_1-\bar{y})^2 + \sum_{(x,y)\in\mathcal{E}_2} (y-\bar{y})^2 - |\mathcal{E}_2|(\bar{y}_2-\bar{y})^2 \\
&= \sum_{(x,y)\in\mathcal{E}} (y-\bar{y})^2 - (|\mathcal{E}_1|\bar{y}_1^2 + |\mathcal{E}_1|\bar{y}^2 - |\mathcal{E}_1|2\bar{y}_1\bar{y} + |\mathcal{E}_2|\bar{y}_2^2 + |\mathcal{E}_2|\bar{y}^2 - |\mathcal{E}_2|2\bar{y}_2\bar{y}) \\
&= \sum_{(x,y)\in\mathcal{E}} (y-\bar{y})^2 - (|\mathcal{E}_1+\mathcal{E}_2|\bar{y}^2 - 2\bar{y}(|\mathcal{E}_1|\bar{y}_1 + |\mathcal{E}_2|\bar{y}_2) + |\mathcal{E}_1|\bar{y}_1^2 + |\mathcal{E}_2|\bar{y}_2^2) \\
&= \sum_{(x,y)\in\mathcal{E}} (y-\bar{y})^2 + \bar{y}|\mathcal{E}|\bar{y} - (|\mathcal{E}_1|\bar{y}_1^2 + |\mathcal{E}_2|\bar{y}_2^2) \\
&= \sum_{(x,y)\in\mathcal{E}} (y-\bar{y})^2 + \frac{1}{|\mathcal{E}|}(\sum_{(x,y)\mathcal{E}} y)^2 - \left(\frac{1}{|\mathcal{E}_1|}(\sum_{(x,y)\in\mathcal{E}_1} y)^2 + \frac{1}{|\mathcal{E}_2|}(\sum_{(x,y)\in\mathcal{E}_2} y)^2 \right).
\end{aligned}
\tag{4.4}
$$

function $RegRefinementBound2(\mathcal{P})$ **returns** $C_{\mathcal{P}}$;
input: $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$, a partition, $\mathcal{E}_1$ sorted;

if $\mathcal{E}_1 = \emptyset$ **return** $E_{\mathcal{E}_2}$;
$K := \sum_{(x,y)\in\mathcal{E}}(y-\bar{y})^2 + \frac{1}{|\mathcal{E}|}(\sum_{(x,y)\in\mathcal{E}} y)^2$;
$sum2 := \sum_{(x,y)\in\mathcal{E}_2} y$;
$S := prefixSum(\mathcal{E}_1)$;
$minerr := Q_{\mathcal{P}}$;
for each i **from** 1 **to** $|\mathcal{E}_1|$ **do**

$Q_{\mathcal{P}_1} := K - (\frac{1}{i}(S[i])^2 + div(1, |\mathcal{E}_2| + |\mathcal{E}_1| - i)(sum2 + S[|\mathcal{E}_1|] - S[i])^2)$;
$Q_{\mathcal{P}_2} := K - (div(1, |\mathcal{E}_1| - i)(S[|\mathcal{E}_1|] - S[i])^2 + \frac{1}{|\mathcal{E}_2|+i}(sum2 + S[i])^2)$;
$minerr := \min\{minerr, Q_{\mathcal{P}_1}, Q_{\mathcal{P}_2}\}$

return $minerr$;

Figure 4.2: An efficient implementation of $RegRefinementBound$.

From that, we can reformulate the optimization problem as

$$C_{\mathcal{P}} = \min_{\mathcal{P}'} Q_{\mathcal{P}'} = \sum_{(x,y)\in\mathcal{E}} (y-\bar{y})^2 + \frac{1}{|\mathcal{E}|}(\sum_{\mathcal{E}} y)^2 - \max_{\mathcal{P}'} \left(\frac{1}{|\mathcal{E}'_1|}(\sum_{\mathcal{E}'_1} y)^2 + \frac{1}{|\mathcal{E}'_2|}(\sum_{\mathcal{E}'_2} y)^2 \right)$$

where $\mathcal{P}' = (\mathcal{E}'_1, \mathcal{E}'_2)$ is a refinement of $\mathcal{P}$. Thus we are left with a maximization problem that can be computed in time linear in the size of $\mathcal{E}$. All that is required are a few preprocessing steps to compute the sum of all the regression values in $\mathcal{E}_2$ and the prefix sums ($S[i] = \sum_{1\leqslant j\leqslant i} \mathcal{E}_1[j]$) of each element in the sorted $\mathcal{E}_1$. Here, $\mathcal{E}[j]$ denotes the regression value of the j-th element in $\mathcal{E}$. The details are given in Figure 4.2. In the algorithm, the function $div(x, y)$ is defined to be x/y if $y \neq 0$, and 0 otherwise. Note that the formula for $Q_{\mathcal{P}_1}$ and $Q_{\mathcal{P}_2}$ has the same general form of (4.4).

Each of the preprocessing steps can be done in $O(|\mathcal{E}|)$ time. There are $O(|\mathcal{E}|)$ iterations in the **for** loop, and each iteration takes $O(1)$ time. The overall complexity of the algorithm is thus $O(|\mathcal{E}|)$. □

Can we avoid conducting an exhaustive search of the solution set? Given the form of the minimizing refinement, one might conjecture that given a partition $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$, if $\bar{y}_1 < \bar{y}_2$, where $\bar{y}_1$ and $\bar{y}_2$ denote, respectively, the empirical means of the regression values in $\mathcal{E}_1$ and $\mathcal{E}_2$, then the minimizing refinement must be in the set

$$S1 = \{(\mathcal{E}'_1, \mathcal{E}'_2) \ : \ \mathcal{E}'_1 = \mathcal{E}_1[1, i],\ \mathcal{E}'_2 = \mathcal{E}_1[i+1, |\mathcal{E}_1|] \cup \mathcal{E}_2, 1 \leqslant i \leqslant |\mathcal{E}_1|\}$$

where $\mathcal{E}_1$ here is assumed sorted. Similarly when $\bar{y}_1 > \bar{y}_2$. This is not true, as shown in the next example.

Example 4.2.11. Consider the partition

$$\mathcal{P} = (\{(x_1, 0.3), (x_2, 0.5), (x_3, 0.98)\}, \{(x_4, 0.41), (x_5, 0.53), (x_6, 0.77), (x_7, 0.9)\})$$

with $\bar{y}_1 = 0.59$ and $\bar{y}_2 = 0.65$. The best refinement in the set $S1$, with error 0.255, is

$$(\{(x_1, 0.3), (x_2, 0.5)\}, \{(x_3, 0.98), (x_4, 0.41), (x_5, 0.53), (x_6, 0.77), (x_7, 0.9)\}).$$

The actual minimizing refinement is

$$(\{(x_3, 0.98)\}, \{(x_1, 0.3), (x_2, 0.5), (x_4, 0.41), (x_5, 0.53), (x_6, 0.77), (x_7, 0.9)\})$$

with error 0.254. ◀

Another conjecture is that the errors obtained by increasing the index i in both $S1$ and $S2$ (defined similarly to $S1$, see below) traces a quadratic curve with a single (local) minimum. This also turns out to be false, as shown in the next example.

Example 4.2.12. Consider the partition

$$\mathcal{P} = (\{(x_1, 0.29), (x_2, 0.36), (x_3, 0.81), (x_4, 0.92)\}, \{(x_5, 0.95)\}).$$

The refinements in the set

$$S2 = \{(\mathcal{E}_1', \mathcal{E}_2') \ : \ \mathcal{E}_1' = \mathcal{E}_1[i+1, |\mathcal{E}_1|], \ \mathcal{E}_2' = \mathcal{E}_1[1, i] \cup \mathcal{E}_2, 0 \leqslant i \leqslant |\mathcal{E}_1|\}$$

where $\mathcal{E}_1$ is assumed sorted produces the following sequence of errors with increasing values of i: $[0.3001, 0.3938, 0.2689, 0.3202, 0.4009]$. ◀

4.2.1.3 Searching

Figure 4.3 gives the main predicate searching algorithm. It is based on the LR predicate enumeration algorithm (Algorithm II, Figure 2.3). The SeenSet enumeration algorithm (Algorithm I, Figure 2.2) can be similarly adapted for use here.

The parameters and subroutine calls in Figure 4.3 have the usual meanings; see §3.2.1.3 for details. The open list is ordered increasingly in the regression refinement bounds of predicates. As is usual, we assume that the input predicate rewrite system is monotone and satisfies all the conditions for ensuring uniqueness and completeness of predicate derivations.

Given the best predicate returned by *RegPredicate*, construction of the hypothesis regression stump is straightforward.

4.2.2 Learning Trees

We now proceed with the top-down tree-induction algorithm. Let X be the set of individuals and $\rightarrowtail$ a predicate rewrite system. A *regression tree* is a binary tree where each non-terminal node is labelled with a predicate in $S_{\rightarrowtail}$, and each terminal node is labelled with a real number. A tree defines a function $f : X \to \mathbb{R}$ in the usual way.

function $RegPredicate(\mathcal{E}, \rightarrowtail, P)$ **returns** a predicate;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, *prune* parameter;

$openList := [top]$;
$predicate := top$;
$error := E_{\mathcal{E}}$;
while $openList \neq []$ **do**
$p := head(openList)$;
$openList := tail(openList)$;
if $C_p > error$ **then continue**;
for each LR redex r via $r \rightarrowtail b$, for some b, in p **do**
$q := p[r/b]$;
if q is regular **then**
if $Q_q < error$ **then**
$predicate := q$;
$error := Q_q$;
if $Q_q < P$ **then** $P := Q_q$;
if $C_q \leqslant P \ \wedge \ Q_q > C_q \ \wedge \ Redexes(q) \neq \emptyset$ **then**
$openList := Insert(q, openList)$;

return $predicate$;

Figure 4.3: Algorithm for finding a predicate to split a node

function $RegLearn(\mathcal{E}, \rightarrowtail, P)$ **returns** a regression tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, $prune$ parameter;

$tree$:= $RegBuildTree(\mathcal{E}, \rightarrowtail, P)$
label each leaf node of $tree$ by the empirical mean of the regression values;
$tree$:= $RegPostprune(tree)$;
return $tree$;

Figure 4.4: Decision-tree learning algorithm

function $RegBuildTree(\mathcal{E}, \rightarrowtail, P)$ **returns** a regression tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, $prune$ parameter;

$tree$:= single node (with examples $\mathcal{E}$);
$p := RegPredicate(\mathcal{E}, \rightarrowtail, P)$;
if $Q_p = E_{\mathcal{E}}$ **then return** $tree$;
$tree.predicate := p$;
$\mathcal{E}_+ := \{ (x, y) \in \mathcal{E} \ : \ (p\ x) \}$;
$\mathcal{E}_- := \{ (x, y) \in \mathcal{E} \ : \ \neg(p\ x) \}$;
$tree.left := RegBuildTree(\mathcal{E}_+, \rightarrowtail, P)$;
$tree.right := RegBuildTree(\mathcal{E}_-, \rightarrowtail, P)$;
return $tree$;

Figure 4.5: Tree building algorithm

The top-down tree induction algorithm is given in Figures 4.4 and 4.5. In our implementation, the error complexity pruning algorithm of [32] is used for tree post-pruning. Any other post-pruning technique can be employed.

4.2.3 Others

4.2.3.1 Learning Regression Lists

The covering algorithm for learning decision lists can be extended straightforwardly to regression learning. At each iteration, the predicate that covers the subset of examples with the lowest squared error is picked. By (the proof of) Proposition 3.3.18, the resulting regression list is a linear model.

Predicate pruning does not work at all in the case of regression-list learning. Every predicate that covers a non-empty subset of examples can potentially be strengthened

to cover only a singleton set, which by definition has zero error. This means no (non-trivial) predicate can ever be pruned.

4.2.3.2 On-Line Learning

An on-line algorithm for regression-tree learning is presented in Chapter 5. It is similar in form to the on-line algorithm for classification-tree learning described briefly in §3.2.4.4. There is a function $AddExample$ that incorporates a new example into an existing tree and updates the meta-data in the tree. There is also a function $RemoveExample$ that removes an existing example from a tree. A function $Retrain$ can be used to bring the tree up-to-date with the current set of training examples.

4.3 Generalization Bounds

In the presence of sufficiently many examples, generalization is assured for finite function classes. This is a direct consequence of Hoeffding's inequality.

Definition 4.3.1. Let X be a set and P a probability distribution on $X \times [0,1]$. Given a function $f : X \to [0,1]$, we define the error $er_P(f)$ of f with respect to P by

$$er_P(f) = \mathbb{E}_{(x,y)\sim P}(f(x) - y)^2.$$

Given a sample $z = ((x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)) \in (X \times [0,1])^m$, we define the sample error $\hat{er}_z(f)$ of f on z by

$$\hat{er}_z(f) = \frac{1}{m}\sum_{i=1}^{m}(f(x_i) - y_i)^2.$$

Theorem 4.3.2 ([3]). *Let $\mathcal{F}$ be a finite class of functions mapping from a set X into $[0,1]$ and P a probability distribution on $X \times [0,1]$. For $0 < \epsilon < 1$ and m a positive integer, we have*

$$P^m\{|\hat{er}(f) - er_P(f)| \geqslant \frac{\epsilon}{2} \text{ for some } f \in \mathcal{F}\} \leqslant 2|\mathcal{F}|e^{-\epsilon^2 m/2}.$$

For infinite function classes, generalization can be shown to be governed by their pseudo-dimensions, a quantity analogous to VC dimensions in classification learning. Facts about pseudo-dimensions, in particular the way they can be used to bound the covering numbers of function classes, can be found in [3, Part III].

Definition 4.3.3. Let $\mathcal{F}$ be a set of functions from X to $\mathbb{R}$ and $S = \{x_1, x_2, \ldots, x_m\}$ a subset of X. Then S is *pseudo-shattered* by $\mathcal{F}$ if there are real numbers $r_1, r_2, \ldots, r_m$ such that for each $b = (b_1, b_2, \ldots, b_m) \in \{0,1\}^m$ there is a function f_b in $\mathcal{F}$ satisfying $sgn(f_b(x_i) - r_i) = b_i$ for $1 \leqslant i \leqslant m$. We say $r = (r_1, r_2, \ldots, r_m)$ *witnesses* the shattering.

Definition 4.3.4. Suppose that $\mathcal{F}$ is a set of functions from X to $\mathbb{R}$. Then $\mathcal{F}$ has *pseudo-dimension* d if d is the maximum cardinality of a subset S of X that is pseudo-shattered

by $\mathcal{F}$. If no such maximum exists, we say $\mathcal{F}$ has infinite pseudo-dimension. The pseudo-dimension of $\mathcal{F}$ is denoted $Pdim(\mathcal{F})$.

Unsurprisingly, the pseudo-dimensions of some common predicate classes definable using predicate rewrite systems can be large. The analysis is not all that different from the VC dimension analysis given in §3.4.3. We give one example to illustrate the kind of reasoning involved.

Proposition 4.3.5. *Let $\mathcal{F}_\exists = \{f_{i,x,y} : i \in \mathbb{N}, x, y \in [0,1]\}$ be the set of functions where each*

$$f_{i,x,y} : 2^{\mathbb{N}} \to [0,1]$$

is defined by

$$f_{i,x,y}(t) \;=\; \begin{cases} x & \textit{if } \exists z \in t.(z = i); \\ y & \textit{otherwise.} \end{cases}$$

Then $Pdim(\mathcal{F}_\exists) = \infty$.

Proof. It's sufficient to restrict attention to a subset of $\mathcal{F}_\exists$ defined by $\mathcal{F}'_\exists = \{f_{i,1,0} \in \mathcal{F}_\exists\}$. For each $n \in \mathbb{N}$, we construct a set $\{X_1, X_2, \ldots, X_n\}$ as in Proposition 3.4.18. For instance, when $n = 3$, we have the following:

$$\{X_1 = \{2,5,6,8\},\ X_2 = \{3,5,7,8\},\ X_3 = \{4,6,7,8\}\}.$$

This set is pseudo-shattered by $\mathcal{F}'_\exists$. The shattering is witnessed by $(0.5, 0.5, 0.5)$. □

4.4 Related Work

Both TILDE [23] and S-CART [118] (and its predecessor SRT [114]) support regression learning. The regression algorithm in S-CART is actually quite sophisticated: multiple linear step-wise regression is applied to fit a linear model to examples at each leaf node after the initial tree is grown. This extension to regression-tree induction was previously suggested in [32, §8.8] and, following [161], such a tree is now commonly called a model tree.

Another model-tree induction algorithm is Mr-SMOTI [4], which is essentially an upgrade of the SMOTI algorithm [132] to the multi-relational setting. Model trees induced by Mr-SMOTI are more complex than model trees induced by S-CART: linear regressions are not limitted to leaf nodes and can be performed at any stage during tree construction. Non-terminal regression nodes only have a single child and all training examples pass through to the child after appropriate transformations of some attribute values. Mr-SMOTI learns from multiple relations and uses a graphical language called selection graphs (which is less expressive than Prolog since it cannot represent recursive concepts) to partition examples at the splitting nodes.

A system that uses the covering approach to learn first-order regression models is FORS [98].

Chapter 5

Incremental Induction

5.1 Introduction

In applications where data are received in a continuous stream, being able to learn incrementally is important. The assumption is that it is in general a lot less expensive to revise existing knowledge than to start anew every time a new example is received. We study an on-line version of the top-down tree induction algorithm in this chapter. An on-line version of the covering algorithm can be constructed in a similar fashion.

The basic setting is as follows. At each time point,

1. the learner receives an individual x,
2. makes a prediction $\hat{h}(x)$ based on the current hypothesis $\hat{h}$, and
3. gets the correct label y for x and incurs a loss depending on the discrepancy between y and $\hat{h}(x)$.

The central problem here is the hypothesis-update step. For that, we need an algorithm that takes as input a tree and a training example and returns an updated tree for use at the next time point.

What can be reasonably demanded of the intermediate trees returned by the on-line algorithm? Preferably, one would like to have a good understanding of the usual properties associated with a tree, properties like predictive power, compactness, and comprehensibility. A simple way to achieve that understanding is to require the tree built by the on-line algorithm on seeing a sequence of examples be *functionally equivalent* to the tree built by the batch algorithm using the same set of examples. Functional equivalence is not easy to define. We can settle for a condition that is sufficient to guarantee functional equivalence, however it is defined, by demanding that the tree produced by the on-line algorithm on seeing a sequence of examples be identical to the tree produced by the batch algorithm for the same set of examples, and this is the approach we take here.

To focus the discussion and motivate some of the design decisions, we begin with some desiderata (adapted from [191]) for a good on-line tree-induction algorithm.

1. The incremental cost of updating the tree should be lower than the cost of building the tree from scratch using all the examples seen so far. The cumulative sum

of the incremental costs need not be less since we care only about the cost of being brought up to date.

2. If the underlying concept is static, the induced tree should stabilize after reaching optimality, that is, in the limit of infinitely long sequences of training examples, no revision should be needed.
3. The algorithm should be able to track drifting/changing concepts.
4. The memory requirement of the algorithm should not grow with the total number of examples seen so far.
5. The tree induced should be independent of the order in which the training examples are presented to the algorithm.
6. The tree should avoid overfitting noise.

To address the first point, we identify some sufficient conditions to detect whether a certain node in the (current) tree remains optimal with the addition or deletion of an example and need not be changed. To address the third point, and to a lesser degree, the fourth point, we employ a moving window to focus on a subset of examples at any time point. To address the sixth point, we use tree post-pruning.

We are thus lead to the following definition.

Definition 5.1.1. Let $E = [(x_1, y_1), (x_2, y_2), \ldots]$ be a sequence of examples, W a positive integer, and $E_{i,j}$ $(i \leqslant j)$ the subsequence of E starting from the i-th element and ending at the j-th element. (If i is less than 1, then $E_{i,j}$ is defined to be $E_{1,j}$.) Let $\mathcal{E}_{i,j}$ be the set formed using $E_{i,j}$. Let H_k be the hypothesis generated from $E_{1,k}$ by an incremental algorithm Alg_I. We say Alg_I is *lossless* with respect to a batch algorithm Alg_B if for all $k > 0, H_k = Alg_B(\mathcal{E}_{k-W+1,k})$.

In the definition, W is the size of the moving window.

In the following sections, we will give two incremental tree-induction algorithms, one for classification and one for regression. Each one is lossless with respect to its corresponding batch learning algorithm. Both algorithms share a common structure. We will start with the regression algorithm in Section 5.2. The somewhat easier classification algorithm is presented in Section 5.3.

5.2 Regression

The goal is to show that given the same sequence of examples, the tree produced by the on-line algorithm is identical to the one produced by the batch algorithm. We first state some properties of the batch algorithm in §5.2.1. This is followed by descriptions of the various component of the incremental algorithm in §5.2.2.

5.2.1 Properties of the Batch Algorithm

In what follows, we denote by $\mathcal{P}_p(\mathcal{E})$ the partition of a set $\mathcal{E}$ of examples induced by a predicate p.

Definition 5.2.1. Let $\mathcal{E}$ be a set of examples and $\mathcal{S}$ a set of predicates. A predicate p is a *best predicate* in $\mathcal{S}$ on $\mathcal{E}$ if for all $q \in \mathcal{S}$, $Q_{\mathcal{P}_p(\mathcal{E})} \leqslant Q_{\mathcal{P}_q(\mathcal{E})}$.

Note that, given a set of predicates $\mathcal{S}$, there can be multiple best predicates in $\mathcal{S}$ on a given set of examples.

Definition 5.2.2. Let $\mathcal{E}$ be a set of examples and $\rightarrowtail$ a predicate rewrite system. A predicate $p \in S_{\rightarrowtail}$ is *optimal* on $\mathcal{E}$ if it is a best predicate in $S_{\rightarrowtail}$ on $\mathcal{E}$, and it is the first one encountered in search by the $RegPredicate$ algorithm (Figure 4.3, page 97) using the default prune parameter.

Let T be a regression tree. Given a tree node t in T, we denote by $t.pred$ the predicate labelling t, and by $t.tset$ the set of examples in t.

Definition 5.2.3. Let $\mathcal{E}$ be a set of examples, $\rightarrowtail$ a predicate rewrite system, and T a regression tree built by some algorithm using $\mathcal{E}$ and $S_{\rightarrowtail}$. We define the *optimality* of nodes in T as follows.

1. A non-terminal node t in T is *optimal* if $t.pred$ is optimal on $t.tset$.
2. A terminal node t in T is *optimal* if there is no predicate in $S_{\rightarrowtail}$ that can split the node with error lower than $E_{t.tset}$.

We say T is *locally optimal* if every node in it is optimal.

The batch algorithm $RegLearn$ (Figure 4.4, page 98) without the final tree post-pruning step generates regression trees that are locally optimal.

5.2.2 The Incremental Algorithm

Our incremental algorithm, shown in Figure 5.1, is made up of three basic procedures: a function $AddExample$ for adding a new example to a tree, a function $RemoveExample$ for removing an existing example from a tree, and a function $Retrain$ for bringing a tree up to date. The basic idea is simple. In each iteration, an example is received from the data stream and added to the existing tree. If the window is full, the oldest example is first removed from the tree, then a call is made to update the tree. Note that the classification algorithm also has the same structure. In fact, we will reuse Figures 5.1, 5.2, 5.3 and 5.4 in Section 5.3, with appropriate instantiations of the code fragments that have been abstracted away. (The literate programming convention [109] is followed here.)

Every tree node has a boolean attribute called $dirty$. (When a tree node is first created by the batch algorithm, $dirty$ is initialized to $false$.) As examples are added and removed, parts of the tree become potentially suboptimal, and these are tagged as such by setting their $dirty$ attributes to $true$.

Definition 5.2.4. We say a node t in a tree T is *clean* if $t.dirty = false$ and for all nodes n in T, if n is an ancestor of t, then $n.dirty = false$.

```
function IncUpdate()

1. T := empty tree;
2. while true do
3.     (x, y) := getNextExample();
4.     if window buffer is full then
5.         (x', y') := findOldestExample(T);
6.         T := RemoveExample(T, (x', y'));
7.     T := AddExample(T, (x, y));
8.     T := Retrain(T);
```

Figure 5.1: Incremental update algorithm

The methods used for detecting potential suboptimality of tree nodes in both *AddExample* and *RemoveExample* rely heavily on the gap between the best and second best predicates on a set of examples. Intuitively, if the gap is large, then adding or removing an example will not affect the optimality of the best predicate. This motivates the next definition.

Definition 5.2.5. Let $\mathcal{E}$ be a set of examples and $\mathcal{S}$ a set of predicates. We define $Q2_{\mathcal{E}}$ to be $Q_{\mathcal{P}_q(\mathcal{E})}$, where q is a best predicate in $\mathcal{S} \setminus \{p\}$ on $\mathcal{E}$, and p is a best predicate in $\mathcal{S}$ on $\mathcal{E}$.

It is easy to show that $Q2$ is well-defined, in the sense that the value assigned is the same regardless of the choices made of p and q, if there are choices to be made at all.

Proposition 5.2.6. *Let $\mathcal{E}$ be a set of examples and $\mathcal{S}$ a set of predicates. Let p be a best predicate in $\mathcal{S}$ on $\mathcal{E}$. Then for all $q \in \mathcal{S}$, if $q \neq p$ then $Q_{\mathcal{P}_q(\mathcal{E})} \geqslant Q2_{\mathcal{E}}$.*

Proof. By Definition 5.2.5. □

Proposition 5.2.7. *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be sets of examples and let $\mathcal{S}$ be a set of predicates. If $\mathcal{E}_1 \subseteq \mathcal{E}_2$, then $Q2_{\mathcal{E}_1} \leqslant Q2_{\mathcal{E}_2}$.*

Proof. Let p be a best predicate in $\mathcal{S}$ on $\mathcal{E}_1$, q a best predicate in $\mathcal{S} \setminus \{p\}$ on $\mathcal{E}_1$, r a best predicate in $\mathcal{S}$ on $\mathcal{E}_2$, and s a best predicate in $\mathcal{S} \setminus \{r\}$ on $\mathcal{E}_2$. If $s \neq p$, then we have $Q2_{\mathcal{E}_1} = Q_{\mathcal{P}_q(\mathcal{E}_1)} \leqslant Q_{\mathcal{P}_s(\mathcal{E}_1)} \leqslant Q_{\mathcal{P}_s(\mathcal{E}_2)} = Q2_{\mathcal{E}_2}$ by Propositions 5.2.6 and 4.2.4. Otherwise, if $s = p$, then $r \neq p$ and we have

$$Q2_{\mathcal{E}_1} = Q_{\mathcal{P}_q(\mathcal{E}_1)} \leqslant Q_{\mathcal{P}_r(\mathcal{E}_1)} \leqslant Q_{\mathcal{P}_r(\mathcal{E}_2)} \leqslant Q_{\mathcal{P}_s(\mathcal{E}_2)} = Q2_{\mathcal{E}_2}$$

by Propositions 5.2.6 and 4.2.4. □

For housekeeping, every tree node keeps two variables *sqerror* and *sqerror2*. At all times, *sqerror* holds the value of $Q_{\mathcal{P}}$ where $\mathcal{P}$ is the partition of *tset* induced by *pred*, and *sqerror2* holds a value that is supposed to be lower or equal to $Q2_{tset}$.

```
function Retrain(T) returns an updated tree T';
input: T, a tree;

if (T.root.dirty = true) then
    T := rebuildTree(T.root);
    return T;

if isTerminal(T.root) then return T;
T.ltree := Retrain(T.ltree);
T.rtree := Retrain(T.rtree);
return T
```

Figure 5.2: Algorithm for retraining a tree

Semi-optimal Trees We now define the concept of semi-optimal trees. This is the invariant property with which we prove the lossless result. Intuitively, a semi-optimal tree is a tree that is close to, at varying degrees depending on the situation, being locally optimal, with appropriate augmentation to indicate the nodes that have become potentially sub-optimal.

Definition 5.2.8. A tree T is *semi-optimal* if for every node t in T,

1. if t is clean, then t is optimal;
2. if t is clean and non-terminal, then $t.sqerror2 \leqslant Q2_{t.tset}$.

A semi-optimal tree T with only clean nodes is locally optimal.

We want to show that the trees we have immediately after lines 6, 7 and 8 in Figure 5.1 are all semi-optimal. In the following, we give the $Retrain$, $AddExample$ and $RemoveExample$ functions, and show that they are all semi-optimality preserving, that is, given a semi-optimal tree as input, they all produce a semi-optimal tree as output.

The Retrain Procedure Figure 5.2 shows the $Retrain$ procedure, which works as follows. It traverses the input tree and rebuilds the first node down each branch whose $dirty$ attribute is $true$. Rebuilding is done using the function $rebuildTree$, which takes a node and discards the subtree rooted at the node, and then calls the batch learning algorithm to induce a new subtree based on the set of examples in the original node. This version of the batch learning algorithm performs the additional work of recording the values of $sqerror$ and $sqerror2$ during search, where the latter is set to $Q2_{t.tset}$ for every node t in the tree. This batch algorithm also sets to $false$ the $dirty$ attribute of every node it constructs.

Proposition 5.2.9. *Given as input a semi-optimal tree T, Retrain returns as output a semi-optimal tree T'. Furthermore, every node in T' is clean.*

```
function AddExample(T, (x, y)) returns an updated tree T';
inputs: T, a tree;
        (x, y), an example;

t := T.root;
while true do
    t.tset := t.tset ∪ {(x, y)};
    if t.dirty = true then break;
    if isTerminal(p) then
        ⟨⟨AddExample::Test for Terminal Nodes⟩⟩
        break;
    val := (t.pred x);
    ⟨⟨AddExample::Test for Non-Terminal Nodes⟩⟩
    if val = true then t := t.ltree; else t := t.rtree;
return T;
```

Figure 5.3: Algorithm for adding a new example to a tree

Proof. Denote by D the set of all the first nodes down any branch in T whose *dirty* attribute is *true*. T' is similar in structure to T except that all the subtrees rooted at one of the nodes in D get rebuilt using the batch algorithm. The nodes that T' and T share must still satisfy the two conditions of Definition 5.2.8. Furthermore, they are all clean. Clearly, the new nodes in T' generated by the batch algorithm are all clean and optimal. □

Adding An Example Figure 5.3 gives the algorithm for adding an example to a tree. The general idea of *AddExample* is this: given a new example, we push it down the tree, updating the set of examples at each node traversed, and return after setting to *true* the *dirty* attribute of the first node encountered that has the potential to become sub-optimal.

There are separate tests for checking the optimality of terminal and non-terminal nodes. The pseudo-code for these have been abstracted away in Figure 5.3. We now instantiate the missing pieces, starting with the optimality test for non-terminal nodes. The test is motivated by the following simple observation.

Observation 5.2.10. *Let $\mathcal{E}$ be a set of examples, (x, y) an example, $\rightarrowtail$ a predicate rewrite system, and $\mathcal{E}' = \mathcal{E} \cup \{(x, y)\}$. Further, let $p \in S_{\rightarrowtail}$ be the optimal predicate on $\mathcal{E}$, and let e be such that $e \leqslant Q2_{\mathcal{E}}$. If $Q_{\mathcal{P}_p(\mathcal{E}')} < e$, then p remains optimal on $\mathcal{E}'$.*

The statement follows from the fact that $Q2_{\mathcal{E}'}$ must be larger than or equal to $Q2_{\mathcal{E}}$ (by Proposition 5.2.7), and hence is strictly larger than $Q_{\mathcal{P}_p(\mathcal{E}')}$. Translating this condition into code, we obtain the following.

⟨⟨AddExample::Test for Non-Terminal Nodes⟩⟩≡
if $val = true$ **then** $subtree$:= $t.ltree$; **else** $subtree$:= $t.rtree$;
$t.sqerror := t.sqerror + E_{subtree.tset \cup \{(x,y)\}} - E_{subtree.tset}$;
if ($t.sqerror \geqslant t.sqerror2$) **then**
$t.dirty := true$;
break;

The first two lines compute $Q_{\mathcal{P}_p(\mathcal{E}')}$. Recall also that $t.sqerror2 \leqslant Q2_{\mathcal{E}}$ since t is clean.

The optimality test for terminal nodes is based on the following fact.

Observation 5.2.11. *Let $\mathcal{E}$ be a set of examples, (x, y) an example, and $\mathcal{E}' = \mathcal{E} \cup \{(x, y)\}$. Let $\mathcal{S}$ be a set of predicates, and $\bar{y}$ be $\frac{1}{|\mathcal{E}|}\sum_{(x,y)\in\mathcal{E}} y$. If $y = \bar{y}$ and there is no predicate in $\mathcal{S}$ that can partition $\mathcal{E}$ with error less than $E_{\mathcal{E}}$, then there is no predicate in $\mathcal{S}$ that can partition $\mathcal{E}'$ with error less than $E_{\mathcal{E}'}$.*

This is easy to see, since $E_{\mathcal{E}'} = E_{\mathcal{E}} = Q_{\mathcal{P}_p(\mathcal{E})} \leqslant Q_{\mathcal{P}_p(\mathcal{E}')}$ for all $p \in \mathcal{S}$ by Proposition 4.2.4. From the above, we get the following.

⟨⟨AddExample::Test for Terminal Nodes⟩⟩ ≡
if $y \neq average(t.tset \setminus \{(x, y)\})$ **then** $t.dirty := true$;

The function $average$ takes as input a set of examples and returns the average of the regression values in the set.

We now show $AddExample$ preserves semi-optimality of trees.

Proposition 5.2.12. *Given as input a semi-optimal tree T and an example (x, y), the function $AddExample$ returns as output a semi-optimal tree T'.*

Proof. Denote by N_{aff} the nodes of T traversed by (x, y), and by N_{rest} all the other nodes. Denote by n the last node in N_{aff} traversed. T' is identical in structure to T, except that the nodes in N_{aff} have their examples updated, and $n.dirty$ can now be $true$. We first note that all the nodes in N_{rest} remains untouched and satisfy the semi-optimality conditions by the semi-optimality of T.

We now show that the nodes in N_{aff} satisfy condition 1 of Definition 5.2.8. Every node $t \in N_{aff} \setminus \{n\}$ is a clean non-terminal node satisfying $t.sqerror < t.sqerror2$ and remains optimal by the semi-optimality of T and Observation 5.2.10. For node n, we need only consider the case where it is clean in T. If n is a non-terminal node, then $n.dirty$ must now be set to $true$. Otherwise, n is a terminal node. It can remain clean iff $y = average(n.tset \setminus \{(x, y)\})$, in which case it remains optimal by the semi-optimality of T and Observation 5.2.11.

We now show that the nodes in N_{aff} satisfy the second condition. Note that for all nodes $t \in N_{aff} \setminus \{n\}$, $t.sqerror2$ remains unchanged, giving $t.sqerror2 \leqslant Q2_{t.tset} \leqslant Q2_{t.tset\cup\{(x,y)\}}$, the second part following from Proposition 5.2.7. Node n need not concern us here because it is either a terminal node or a non-terminal node with $n.dirty = true$. □

```
function RemoveExample(T, (x, y)) returns an updated tree T';
inputs: T, a tree;
        (x, y), an example;

t := T.root;
while true do
    t.tset := t.tset \ {(x, y)};
    if t.dirty = true then break;
    if isTerminal(p) then
        ⟨⟨RemoveExample::Test for Terminal Nodes⟩⟩
        break
    val := (t.pred x);
    ⟨⟨RemoveExample::Test for Non-Terminal Nodes⟩⟩
    if val = true then t := t.ltree; else t := t.rtree;
return T
```

Figure 5.4: Algorithm for deleting a new example from a tree

Removing An Example Figure 5.4 gives the algorithm for removing an example from a tree. It is similar in form to $AddExample$. Given a tree and an example to remove, it goes down the branch of the tree containing the chosen example, updates the set of examples at each node traversed, and mark the first node encountered that has the potential to become sub-optimal.

We now give the optimality tests, beginning with that for non-terminal nodes. The main statement is Observation 5.2.15.

Definition 5.2.13. Let $\mathcal{E}$ be a (non-empty) set of examples, (x, y) an example in $\mathcal{E}$, and $\mathcal{E}' = \mathcal{E} \setminus \{(x, y)\}$. Let p be a predicate, and let $\mathcal{P}_p(\mathcal{E}) = (\mathcal{E}_{p1}, \mathcal{E}_{p2})$ and $\mathcal{P}_p(\mathcal{E}') = (\mathcal{E}'_{p1}, \mathcal{E}'_{p2})$ be the partitions of $\mathcal{E}$ and $\mathcal{E}'$ induced by p. Define the *gain* of p with the removal of (x, y) from $\mathcal{E}$ as

$$\delta_p(\mathcal{E}, (x, y)) = Q_{\mathcal{P}_p(\mathcal{E})} - Q_{\mathcal{P}_p(\mathcal{E}')}.$$

By straightforward algebraic manipulation, one can show that

$$\delta_p(\mathcal{E}, (x, y)) = \max \{E_{\mathcal{E}_{p1}} - E_{\mathcal{E}'_{p1}}, E_{\mathcal{E}_{p2}} - E_{\mathcal{E}'_{p2}}\}.$$

One of the two terms must be zero because either $\mathcal{E}_{p1} = \mathcal{E}'_{p1}$ or $\mathcal{E}_{p2} = \mathcal{E}'_{p2}$. For convenience, we denote the non-equal pair by $\mathcal{E}_{p3}$ and $\mathcal{E}'_{p3}$ and write $\delta_p(\mathcal{E}, (x, y)) = E_{\mathcal{E}_{p3}} - E_{\mathcal{E}'_{p3}}$.

Proposition 5.2.14. *Let $\mathcal{E}$ be a (non-empty) set of examples, (x, y) and example in $\mathcal{E}$ and $\mathcal{E}' = \mathcal{E} \setminus \{(x, y)\}$. Let $\mathcal{S}$ be a set of predicates. The maximum gain with the removal of (x, y) from $\mathcal{E}$*

of any predicate in $\mathcal{S}$, denoted Δ, is bounded from above by $\max\{2(y-y_{max})^2, 2(y-y_{min})^2\}$, where y_{max} is the largest regression value in $\mathcal{E}$, and y_{min} the smallest.

Proof.

$$\begin{aligned}
\Delta &= \max_{p\in\mathcal{S}} \{\delta_p(\mathcal{E}, (x,y))\} \\
&= \max_{p\in\mathcal{S}} \{E_{\mathcal{E}_{p3}} - E_{\mathcal{E}'_{p3}}\} \\
&= \max_{p\in\mathcal{S}} \{(\bar{y}_3 - y)^2(1 + |\mathcal{E}_{p3}|^{-1})\} \\
&\leqslant \max_{p\in\mathcal{S}} \{\max\{2(y_{min} - y)^2, 2(y_{max} - y)^2\}\} \\
&= \max\{2(y_{min} - y)^2, 2(y_{max} - y)^2\}
\end{aligned}$$

The third step is a straightforward algebraic rewriting where $\bar{y}_3 = \frac{1}{|\mathcal{E}_{p3}|}\sum_{(x,y)\in\mathcal{E}_{p3}} y$. □

The following is a straightforward consequence of Proposition 5.2.14 that provides the optimality test for non-terminal nodes.

Observation 5.2.15. *Let $\mathcal{E}$ be a (non-empty) set of examples, (x,y) an example in $\mathcal{E}$ and $\mathcal{E}' = \mathcal{E} \setminus \{(x,y)\}$. Let $\rightarrowtail$ be a predicate rewrite system and suppose $p \in S_{\rightarrowtail}$ is the optimal predicate on $\mathcal{E}$. Let e be such that $e \leqslant Q2_{\mathcal{E}}$, and let $\Delta = \max\{2(y_{max} - y)^2, 2(y_{min} - y)^2\}$. If $e - \Delta > Q_{\mathcal{P}_p(\mathcal{E}')}$, then p remains optimal on $\mathcal{E}'$.*

Translating this into pseudo-code, we obtain the following. The first two lines compute the value of $Q_{\mathcal{P}_p(\mathcal{E}')}$.

```
⟨⟨RemoveExample::Test for Non-Terminal Nodes⟩⟩ ≡
    if val = true then subtree := t.ltree; else subtree := t.rtree;
    t.sqerror := t.sqerror − (E_{subtree.tset} − E_{subtree.tset\{(x,y)}});
    t.sqerror2 := t.sqerror2 − Δ;
    if (t.sqerror2 ⩽ t.sqerror) then
        t.dirty := true;
        break;
```

The approximation to Δ given in Proposition 5.2.14 is rather crude. Improvements can be made if we are willing to store more information, including the y_{min} and y_{max} for each node, and the size of the smallest subsets of $\mathcal{E}$ obtained from the partitionings. Other completely different approaches may be needed to fix this weak spot in the algorithm.

We next state the optimality test for terminal nodes.

Definition 5.2.16. A set of examples $\mathcal{E}$ is *pure* if $\forall(x_1,y_1),(x_2,y_2) \in \mathcal{E},\ y_1 = y_2$.

Observation 5.2.17. *Let $\mathcal{E}$ be a (non-empty) set of examples, (x, y) an example in $\mathcal{E}$, and $\mathcal{E}' = \mathcal{E} \setminus \{(x, y)\}$. Let $\mathcal{S}$ be a set of predicates. If $\mathcal{E}$ is pure and there is no predicate in $\mathcal{S}$ that can partition $\mathcal{E}$ with error less than $E_{\mathcal{E}}$, then there is no predicate in $\mathcal{S}$ that can partition $\mathcal{E}'$ with error less than $E_{\mathcal{E}'}$.*

To see this, observe that $\mathcal{E}'$ must be pure, and from that we have $\forall p \in \mathcal{S},\ Q_{\mathcal{P}_p(\mathcal{E}')} = 0 = E_{\mathcal{E}'}$. This gives us the next code fragment.

$\langle\langle$RemoveExample::Test for Terminal Nodes$\rangle\rangle \equiv$
if $isPure(t.tset \cup \{(x, y)\}) = false$ **then** $t.dirty := true$;

The condition given here appears quite weak. One might conjecture that the stronger condition $y \neq average(t.tset)$ can be used. It is easy to construct counter-examples to show that this is not true.

The function $RemoveExample$ can be shown to preserve semi-optimality of trees.

Proposition 5.2.18. *Given a semi-optimal tree T and an example in $T.root.tset$ as input, $RemoveExample$ returns as output a semi-optimal tree T'.*

Proof. This can be shown with an analogous argument as that used in Proposition 5.2.12, this time using Observations 5.2.15 and 5.2.17. To show that all the nodes in N_{aff} satisfy the second condition of Definition 5.2.8, use Proposition 5.2.14. □

It is now easy to show that the $IncUpdate$ algorithm is lossless with respect to the batch algorithm $Learn$ in the regression setting.

Proposition 5.2.19. *Let $RegLearn2$ be the algorithm which is the same as $RegLearn$ (Figure 4.4, page 98) but without the final tree postpruning step. $IncUpdate$ is lossless with respect to $RegLearn2$.*

Proof. Denote by T_k the tree generated by $IncUpdate$ after seeing $E_{1,k}$. We prove the stronger statement that $\forall k \geqslant 1, T_k = RegLearn2(\mathcal{E}_{k-W+1,k})$ and T_k is semi-optimal. To show $T_k = RegLearn2(\mathcal{E}_{k-W+1,k})$, we need to show that $T_k.tset = \mathcal{E}_{k-W+1,k}$ and T_k is locally optimal.

The proof proceeds by induction on k. In the base case when $k = 1$, clearly both algorithms produce identical trees that are semi-optimal. Next consider the inductive case. Depending on whether the window buffer is full, we have either

$$T_{k+1} = Retrain(AddExample(T_k, (x_{k+1}, y_{k+1})))$$

or

$$T_{k+1} = Retrain(AddExample(RemoveExample(T_k, (x_{k-W+1}, y_{k-W+1})), (x_{k+1}, y_{k+1}))).$$

By the inductive hypothesis, $T_k = RegLearn2(\mathcal{E}_{k-W+1,k})$ and T_k is semi-optimal. In the first case, we have $T_{k+1}.tset = T_k.tset \cup \{(x_{k+1}, y_{k+1})\} = \mathcal{E}_{1,k+1} = \mathcal{E}_{k+1-W+1,k+1}$.

Further, by Proposition 5.2.12, $AddExample(T_k, (x_{k+1}, y_{k+1}))$ is semi-optimal. Then by Proposition 5.2.9, T_{k+1} is semi-optimal and every node in it is clean, which in turn implies that T_{k+1} is locally optimal.

In the second case, we have

$$\begin{aligned} T_{k+1}.tset &= T_k.tset \setminus \{(x_{k-W+1}, y_{k-W+1})\} \cup \{(x_{k+1}, y_{k+1})\} \\ &= \mathcal{E}_{k-W+2,k+1} = \mathcal{E}_{k+1-W+1,k+1}. \end{aligned}$$

By Proposition 5.2.18, $RemoveExample(T_k, (x_{k-W+1}, y_{k-W+1}))$ is semi-optimal. Further, by Proposition 5.2.12,

$$AddExample(RemoveExample(T_k, (x_{k-W+1}, y_{k-W+1})), (x_{k+1}, y_{k+1}))$$

is semi-optimal. Finally, by Proposition 5.2.9, T_{k+1} is semi-optimal and every node in it is clean, which in turn implies that T_{k+1} is locally optimal. □

Remark. In practice, we usually have no need to retrain the tree after every example. Suppose retraining is only done at certain time points, one can reason as in Proposition 5.2.19 to show that the trees produced straight after retraining at those time points are identical to the trees produced by the batch algorithm on the same sets of examples.

The optimality tests we identified in $AddExample$ and $RemoveExample$ are all dependent on the gap between the best and second best predicates in the search space. For any particular tree node, the lack of a clear winning predicate will cause the $dirty$ attribute of the node to be set to $true$ very often, and hence rebuilt very often. If this happens at the root node, which is not unusual given the instability of the greedy top-down induction algorithm, then the incremental cost is equal to the cost of building the tree from scratch. This is the worst case scenario, and there is not much one can do given that the instability is in the batch algorithm, and our aim is to follow it. The unfortunate fact is that even when there is a clear winner, one will observe (unnecessary) periodic rebuilding of tree nodes whose cycle is dependent on the gap between the best and second best predicates.

The tests are clearly quite weak, but there is little one can do. We can strengthen the tests by remembering all the second best predicates and calculating the gaps precisely, but this is too much work for too little benefit. The first issue is memory consumption. Nodes low down in the tree have a small number of examples, and most of the predicates in the search space can be equally good on them. This also happens in the early stages of learning, when the number of examples is still small. The second, more problematic, issue is that the list of second best predicates becomes stale after only one iteration. Storing a list of the n best predicates can only alleviate the problem.

5.3 Classification

The incremental algorithm for classification has the same overall structure as the regression algorithm. We now give the optimality tests for classification, filling the gaps

abstracted away in Figures 5.3 and 5.4. We simply list the observations leading to the code fragments. Correctness can be established easily.

5.3.1 Properties of the Batch Algorithm

Definition 5.3.1. Let $\mathcal{E}$ be a set of examples and $\mathcal{S}$ a set of predicates. A predicate p is a *best predicate* in $\mathcal{S}$ on $\mathcal{E}$ if for all $q \in \mathcal{S}$, $A_{\mathcal{P}_p(\mathcal{E})} \geqslant A_{\mathcal{P}_q(\mathcal{E})}$ and $EN_{\mathcal{P}_p(\mathcal{E})} \leqslant EN_{\mathcal{P}_q(\mathcal{E})}$.

Definition 5.3.2. Let $\mathcal{E}$ be a set of examples and $\rightarrowtail$ a predicate rewrite system. A predicate $p \in S_{\rightarrowtail}$ is *optimal* on $\mathcal{E}$ if it is a best predicate in $S_{\rightarrowtail}$ on $\mathcal{E}$, and it is the first one encountered in search by the *Predicate* algorithm (Figure 3.1, page 32) using the default prune parameter.

Definition 5.3.3. Let $\mathcal{E}$ be a set of examples, $\rightarrowtail$ a predicate rewrite system, and T a classification tree built by some algorithm using $\mathcal{E}$ and $S_{\rightarrowtail}$. We define the *optimality* of nodes in T as follows.

1. A non-terminal node t in T is *optimal* if $t.pred$ is optimal on $t.tset$.
2. A terminal node t in T is *optimal* if for all $p \in S_{\rightarrowtail}$, we have $A_p = A_{t.tset}$ and $EN_p = EN_{t.tset}$.

We say T is *locally optimal* if every node in it is optimal.

The batch algorithm *Learn* (Figure 3.2, page 34) without the final tree post-pruning step generates classification trees that are locally optimal.

5.3.2 The Incremental Algorithm

Definition 5.3.4. Let $\mathcal{E}$ be a set of examples and $\mathcal{S}$ a set of predicates. A $p \in \mathcal{S}$ is *safe* if for all q other than p in $\mathcal{S}$, $A_{\mathcal{P}_p(\mathcal{E})} > A_{\mathcal{P}_q(\mathcal{E})}$.

Observation 5.3.5. *Let $\mathcal{E}$ be a set of examples, (x, y) an example and $\mathcal{E}' = \mathcal{E} \cup \{(x, y)\}$. Let $\rightarrowtail$ be a predicate rewrite system. Suppose $p \in S_{\rightarrowtail}$ is optimal on $\mathcal{E}$. If p is safe and $A_{\mathcal{P}_p(\mathcal{E}')} > A_{\mathcal{P}_p(\mathcal{E})}$, then p remains optimal on $\mathcal{E}'$.*

Safety of predicates is needed here because there could be multiple best predicates in the search space, and the batch algorithm picks the first one found during search.

⟨⟨AddExample::Test for Non-Terminal Nodes⟩⟩ ≡
  if $t.pred$ is not safe **then**
    $t.dirty$:= $true$;
    break;
  if $(val \wedge y \neq maj(t.ltree.tset)) \vee (\neg val \wedge y \neq maj(t.rtree.tset))$ **then**
    $t.dirty$:= $true$;
    break;

The function maj takes a set $\mathcal{E}$ of examples and returns the majority class of $\mathcal{E}$. If there are two or more majority classes, the function returns the one that was the majority class before the addition of the last example.

Observation 5.3.6. *Let $\mathcal{E}$ be a set of examples and $\mathcal{S}$ a set of predicates. If $\mathcal{E}$ is pure, then for all $p \in \mathcal{S}$, $A_{\mathcal{P}_p(\mathcal{E})} = A_{\mathcal{E}}$ and $EN_{\mathcal{P}_p(\mathcal{E})} = EN_{\mathcal{E}}$.*

⟨⟨AddExample::Test for Terminal Nodes⟩⟩ ≡
if $isPure(t.tset) = false$ **then** $t.dirty := true$;

Observation 5.3.7. *Let $\mathcal{E}$ be a set of examples, (x, y) an example in $\mathcal{E}$ and $\mathcal{E}' = \mathcal{E} \setminus \{(x, y)\}$. Let $\rightarrowtail$ be a predicate rewrite system. Suppose $p \in S_{\rightarrowtail}$ is the optimal predicate on $\mathcal{E}$. If p is safe and $A_{\mathcal{P}_p(\mathcal{E}')} > A_{\mathcal{P}_p(\mathcal{E})}$, then p remains optimal on $\mathcal{E}'$.*

The condition in the observation is realized if the example to be removed is incorrectly classified by the tree node in question. The negation of this condition is used in the following.

⟨⟨RemoveExample::Test for Non-Terminal Nodes⟩⟩ ≡
if $t.pred$ is not safe **then**
$t.dirty := true$;
break;
if $(val \wedge y = maj(t.ltree.tset)) \vee (\neg val \wedge y = maj(t.rtree.tset)))$ **then**
$t.dirty := true$;
break;

We again use Observation 5.3.6 to formulate the optimality test for terminal nodes.

⟨⟨RemoveExample::Test for Terminal Nodes⟩⟩ ≡
if $isPure(t.tset) = false$ **then** $t.dirty := true$;

Using a similar argument, one can show that the incremental algorithm for classification is lossless with respect to the batch algorithm.

Proposition 5.3.8. *Let Learn2 be the algorithm which is the same as Learn (Figure 3.2, page 34) but without the final tree postpruning step. IncUpdate is lossless with respect to Learn2.*

5.4 Discussion

We now evaluate the incremental algorithms against the desiderata given earlier.

The incremental cost of updating the tree should be lower than the cost of building the tree from scratch using all the examples in the window.

At any particular time point, the cost of updating the current tree can be, in the worst case, equal to the cost of building the tree from scratch. In general, however, some

Dataset	Window Size	Batch	On-line	Saving
Mutagenesis	188	2889.68	2439.97	15.56%
Musk-1	92	88.87	30.81	65.33%
Mutagenesis	125	2414.44	2242.36	7.13%
Musk-1	60	79.42	59.53	25.04%

Table 5.1: The efficiency of the incremental regression algorithm

Dataset	Window Size	Batch	On-line	Saving
Mutagenesis	188	7577.71	3472.52	54.17%
Musk-1	92	85.64	54.62	36.22%
Mutagenesis	125	6596.28	6016.26	8.79%
Musk-1	60	64.06	52.45	18.12%

Table 5.2: The efficiency of the incremental classification algorithm

kind of saving can be expected. To get an indication of their performance, we tested the algorithms on two benchmark problems: Mutagenesis [186] and Musk-1 [63]. Both are binary classification problems.

The datasets were used 'as is' to test the classification algorithm. To test the regression algorithm, we converted the two datasets into regression problems by relabelling individuals from the same class with random numbers chosen from the same subinterval of the real line. This setup allows us to gain some understanding of the relative performance of the two incremental algorithms.

We use essentially the same hypothesis languages given in [130, §6.2] for the two problems. To reduce the computational effort involved, we simplify the hypothesis language for Musk-1 to use only one condition inside $setExists_1$. To learn a suitable multiple-instance hypothesis, the stump algorithm is used for Musk-1. This mandates a small change to the $Retrain$ algorithm to rebuild, when necessary, only the root node of the tree under revision.

We compared the incremental algorithms against the naïve approach of batch learning the current set of training examples in each iteration. Two different window sizes, 100% and 60% of the size of the training set, for each problem were used. (When the window size is equal to the size of the training set, windowing does not come into play and $RemoveExample$ is never called.) The results are given in Tables 5.1 and 5.2. The running times are in seconds.

As shown, significant reductions in computation can be achieved when there is no windowing. When windowing is in effect, each iteration involves two changes to the set of examples, a deletion and an addition. It takes a predicate with a clear winning margin to survive two optimality tests at each tree node. This, together with the (infamous) instability of the top-down induction algorithm, means that non-trivial savings in computation time are often hard to achieve in windowing mode.

If the underlying concept is static, the induced tree should stabilize after reaching optimality.

In the special case when the target function is a stump and the examples are noise-free, the incremental classification algorithm does satisfy this requirement: no revision is necessary after the target function is first acquired. In general, however, the algorithms in their current forms do not address this point. One simple modification is to stop retraining after a tree with acceptably high accuracy is learned. (Reliable estimation of accuracy should be easy since there is a continuous supply of data.)

The algorithm should be able to track drifting/changing concepts.

The windowing approach should work well for tracking time-varying concepts. One unanswered question is how big should the window size be? Or should the window size change at run time? The answer is probably dependent on the application, but a development of some general guidelines would be desirable. Some ideas from [199] can clearly be useful in our context.

The memory requirement of the algorithm should be modest.

The windowing approach uses a fixed amount of memory.

The tree induced should be independent of the order in which the examples are presented.

By design, the trees induced are independent of the order the examples are presented.

The tree should avoid overfitting noise.

Overfitting can be countered by tree postpruning. We use the error and cost complexity pruning algorithms of CART [32] here. Both methods require an independent validation set. Towards that end, we modify the $IncUpdate$ algorithm to maintain two sets of examples. This can be done by, for example, allocating the odd-numbered examples in the data stream to the training set, and the even-numbered examples to the validation set. After retraining, the postpruning algorithm is called to compute the 'right-sized' tree.

A problem with this scheme is that we may lose the lossless property (suitably redefined to take into consideration the assignment of examples to the validation set) of $IncUpdate$ with respect to the batch algorithms. This is because we cannot guarantee that the pruned subtrees of T_k will be regrown and repruned in the same fashion in T_{k+1}. To resolve this issue, we adopt the virtual pruning mechanism of Utgoff. We introduce into each tree node a field called $pruned$ that has value $true$ if the subtree rooted at that node has been pruned. The postpruning algorithm is modified as follows. Given a tree, the $pruned$ fields of all the nodes are set to $false$ initially. The algorithm then proceeds in the usual fashion, except that instead of actually cutting off an unwanted subtree, it simply sets the $pruned$ field of the root node of the unwanted subtree to $true$. When using the tree to make a prediction, the evaluation algorithm uses the nodes with $pruned = true$ as if they are terminal nodes. In both the classification and regression settings, one can show that the modified algorithm is lossless with respect to the batch algorithms.

Other alternatives to error/cost complexity pruning exist. For example, the approach taken in [192], which is based on the minimum description length principle, can be adopted by ALKEMY.

5.5 Related Work

Incremental tree-induction algorithms The best known approach to incremental tree induction is probably the family of algorithms due to Utgoff *et al*; see [189], [190], [191] and [192]. Unfortunately, these algorithms cannot be adapted for use with ALKEMY because they all make fairly strong assumptions about the size of the predicate search space. Specifically, they all assume that the search space is small and that it is feasible to compute the best predicate cheaply every time a new example is received. This is generally impossible to do in our setting. Indeed, if the predicate search space is so small, one can (and probably should) tap into the rich body of work on Winnow-like algorithms [124], [28] in the on-line learning literature. A paper along that line of investigation that holds particular interest here is [152].

The G-algorithm introduced in [40] is an incremental tree-learning algorithm designed for Q-learning. It was incorporated into TILDE and used for relational reinforcement learning in [66] and [64]. There are experimental results showing that the algorithm works reasonably well in practice, but not a lot is known about it, especially the properties of the trees induced at intermediate stages. It is also not clear whether the algorithm is useful outside the context of reinforcement learning.

The work closest in spirit to the one reported here is [52]; the algorithm proposed there was claimed to be in some sense lossless with respect to the CART system, but no actual proof of correctness was given in the paper.

Incremental learning in ILP Incremental learning has been studied for a long time in ILP. In fact, two of the earliest ILP systems MARVIN ([173], [175]) and MIS [180] are both incremental learners. Other noteworthy incremental ILP systems include CIGOL [149], CLINT [56], MOBAL [106], FORTE [167], and MINERVA [188]. A typical system of this kind works as follows. A current hypothesis in the form of a logic program is maintained at every time point. As each new example is received, the system checks whether the new example is 'explained' by the current hypothesis. If not, a process is triggered to revise the current hypothesis. Revision typically involves specializations and/or generalizations of (parts of) the current hypothesis, sometimes aided by declarative diagnosis techniques like contradiction backtracing and missing answer diagnosis ([126], [188, §3.7]). Systems like MARVIN and MINERVA can also formulate conjectures and questions and actively seek answers to them.

The primary difference between on-line ALKEMY and the incremental ILP systems (and some of the on-line tree-induction algorithms stated earlier) is the overall design goal. In the case of ALKEMY, the current hypothesis is solely a function of the current set of examples, whereas in the case of the incremental ILP systems, the current hypothesis is a function of the current set of examples *and* the previous hypothesis. In

other words, the algorithms of this chapter are essentially optimized batch-learning-at-every-time-step algorithms. In contrast, the incremental ILP systems perform true knowledge revision.

It is probable that some of the sophisticated diagnosis and search techniques employed in incremental ILP systems can be, with appropriate modifications, incorporated into ALKEMY to speed up learning. We may need to weaken the notion of losslessness, however. Extending ALKEMY to do active learning is a particularly exciting prospect.

Chapter 6

Applications

Few things are impracticable in themselves;
and it is for want of application, rather than of means,
that men fail to succeed.
Francois De La Rochefoucauld

6.1 Introduction

In this chapter, we describe in some detail four different applications of the system, each chosen to illustrate principles and techniques that the author finds particularly useful in applying ALKEMY.

The first is a standard binary classification task: the Musk problem introduced in [63]. It is of interest to us here because its solution (using ALKEMY) involves the use of different techniques for handling massive predicate search spaces, and these have obvious applications elsewhere. Indeed, it's worth pointing out that many of the search algorithms and optimization procedures presented in Chapters 2 and 3 were originally conceived in the process of tackling this problem.

The second is a knowledge discovery task in predictive toxicology. We treat a dataset of molecules, each classified as carcinogenic or not, as a database and use ALKEMY as a kind of query language to mine interesting patterns in the underlying molecules. The emphasis here is on the expressiveness of the representation language and how it can be used to precisely form very specific conditions on molecules as conjectured by the data analyst.

The third and fourth applications are presented here to demonstrate the usefulness of ALKEMY as a learning component embedded inside a bigger system. Each of these is an exercise in the design and construction of intelligent agents using symbolic learning. In Section 6.4, we use ALKEMY to do function approximation within the framework of reinforcement learning. This is done in the familiar blocks world domain, a problem simple in appearance but (surprisingly) challenging in nature. In Section 6.5, we use ALKEMY to personalize an infotainment system to the preferences and habits of users, concentrating on a TV program recommender. In both cases, the main difficulty is in coping with the different constraints introduced by other components in the system. These include the need

1. to learn incrementally to achieve acceptable efficiency in a real-time environment;
2. to track concepts that change over time at different frequencies;
3. to handle (an abundance of) low-quality training examples in the case of reinforcement learning; and
4. to cope with a scarcity of (high-quality but mostly negative) training examples in the case of the TV agent.

Some of these problems can be solved (or at least alleviated) using carefully-crafted hypothesis languages that exploit important background knowledge about the application domains. As we shall see, the symbolic nature of ALKEMY can be a useful asset in the formulation of these solutions.

6.2 Multiple-Instance Learning — Musk

The first application is the Musk problem introduced in [63]. The task is to determine whether or not a molecule has a musk odour. It is a standard binary classification problem, with the slightly unusual twist that each molecule is represented as a set of its conformations. In a sense, the training examples are ambiguous. A molecule can be classified as positive (or otherwise) when only one of its many possible instances satisfy a certain property. There have been many attempts made on this benchmark multiple-instance learning problem; see, for example, [9], [137], [164], [82], and [203].

The section is organized as follows. The setup of the problem is described in §6.2.1. We review in §6.2.2 an early solution to the problem presented in [31]. We then point out some undesirable properties of that early effort and give a better solution in §6.2.3. We finish in §6.2.4 with a lesson we learned about the proper use of statistical tests for systems like ALKEMY.

6.2.1 Representation of Individuals

The representation issues for this problem has been described in brief in §3.4.4.2. Here we give a few more specific details. Every molecule in the dataset is represented as a set of feature tuples, each denoting one of the many possible low-energy conformations the molecule can take. Each conformation is a tuple of 166 floating-point numbers, where the first 162 of these represent the distance in angstroms from some origin in the conformation out along a radial line to the surface of the conformation and the remaining 4 numbers represent the position of a specific oxygen atom in the molecule. The attributes are discretized as follows. For each attribute, we calculated the mean μ and the standard deviation σ of the values occurring in the data. We then built up intervals centred on the mean, taking the width of each interval to be one standard deviation, and assigned integral labels to the intervals, so that interval 0 centred on the mean is $[\mu - \sigma/2, \mu + \sigma/2]$, interval 1 is $[\mu + \sigma/2, \mu + 3\sigma/2]$, interval -1 is $[\mu - 3\sigma/2, \mu - \sigma/2]$, and so on. We chose to use 13 intervals in total, labelled -6 through

6. The outermost intervals, -6 and 6, were extended below and above respectively to cover any outlying points. These lead to the following type declarations:

$$-6, -5, \ldots, 5, 6 : \mathit{Distance}$$
$$\mathit{Conformation} = \mathit{Distance} \times \cdots \times \mathit{Distance}$$
$$\mathit{Molecule} = \{\mathit{Conformation}\}.$$

Molecules with a musk odour are labelled true, and those without are labelled false. The function $musk$ we want to learn thus has signature $musk : \mathit{Molecule} \to \Omega$.

Admittedly, the representation just proposed is not the best possible one. There are alternatives and potential improvements one can make to it, but we will not pursue these in this section.

There are two datasets: Musk-1 and Musk-2. Here, we focus on the Musk-2 dataset, which contains 102 molecules (6598 conformations in total), 39 of which are positive examples. The Musk-2 dataset is the larger of the two datasets. But from the point of learning, it is also the harder of the two. This is because it is larger not in the sense that it contains significantly more molecules (Musk-1 has 92 molecules), but larger in the sense that every molecule has many more conformations.

6.2.2 An Early Effort

We will start by recapping the result reported in [31]. This early result was jointly obtained by Xiaobing Wu, John W. Lloyd and the author. Xiaobing Wu implemented the parallel algorithm using MPI (Message Passing Interface) on Bunyip, a 192-processor Beowulf cluster and carried out the experiments; John W. Lloyd and the author contributed to the high-level design of the parallel search algorithm.

The following is the hypothesis language used.

$$\mathit{top} \rightarrowtail \mathit{setExist}_1\ (\wedge_3\ \mathit{top}\ \mathit{top}\ \mathit{top})$$
$$\mathit{top} \rightarrowtail \mathit{proj}_i \circ \mathit{top} \ \text{ where } \ i \in \{1, 2, \ldots, 166\}$$
$$\mathit{top} \rightarrowtail (= j) \ \text{ where } \ j \in \{-6, -5, \ldots, 6\}$$
$$\mathit{top} \rightarrowtail (\neq k) \ \text{ where } \ k \in \{-6, -5, \ldots, 6\}$$

Three seemed to be the right number of conjuncts to use; we observed underfitting using only $\wedge_2$, but had to struggle with overfitting when using $\wedge_4$. The use of predicates of the form $\mathit{proj}_i \circ (\neq j)$ captures disjunctions of intervals conveniently and cheaply. This concept is actually needed here to push accuracy above 80%.

We used the parallelization technique described in §3.5.1.3 to partition the search space into $\binom{166}{2}$ = 13,695 subspaces, each rooted by a predicate of the form

$$\mathit{setExists}_1\ (\wedge_3\ (\mathit{proj}_i \circ \mathit{top})\ (\mathit{proj}_j \circ \mathit{top})\ \mathit{top}) \ \text{ where } \ i, j \in \{1, 2, \ldots, 166\} \text{ and } i < j.$$

Search was conducted using the SeenSet algorithm. (The LR search algorithm had not been invented then.) The cutout parameter was set at 25,000. This means that within

each subspace, the algorithm would terminate the search after examining successively 25,000 predicates without finding one that strictly improves the (current best) accuracy. The computation is an almost "embarrassingly parallel" task, but there are overlaps between the subspaces.

Using 160 processors on the cluster, each an Intel processor running at 500MHz, the learner took about 30 hours to do three 10-fold cross validations and recorded an average accuracy of 82.2%. The following hypothesis was produced for the whole dataset:

$$
\begin{aligned}
&musk\ m = \\
&\quad if\ \ setExists_1\ (\wedge_3\ (proj_{29} \circ (\neq -4))\ (proj_{119} \circ (= 1))\ (proj_{132} \circ (= -2)))\ m \\
&\quad then\ 1 \\
&\quad else\ \ 0.
\end{aligned} \tag{6.1}
$$

There is some regularity to the hypotheses produced in the cross validations. In fact, the exact same hypothesis as (6.1) was produced in 25 out of the 30 folds computed. In the remaining 5 folds, the hypotheses induced retained the two constraints on the 119th and 132nd features.

6.2.3 Doing it without Bunyip

We now take up where we left off in [31]. The most negative aspect of the result presented in the previous section is the prohibitive computational cost incurred. It would be interesting to try and replicate the result on a standard desktop computer.

The main difficulty with doing this is achieving consistent performance. In 10 fold cross validations, ALKEMY will often obtain good accuracy in the region of 80-100% on the validation set in 7-8 out of the 10 folds, and performs poorly (40-60% accuracy) in the remaining ones, ruining the overall result. This problem can be traced back to the large discrepancy between the (small) number of training examples and the (incredibly rich) hypothesis language used. There is little one can do about the size of the training set. One can try to reduce the 'capacity' of the hypothesis language, however.

Predicate Rewrite Pruning The size of the search space is a useful piece of information to have. There are a total of 4317 predicates (including top) defined on type $Conformation$. In the case of $k = 3$ conjuncts, the size of the search space is the rather discouraging figure of 13,418,273,519.

The number of predicates defined on $Conformation$ can be substantially cut down using the predicate rewrite pruning technique described in §3.5.1.3. Setting the target accuracy at an optimistic 99%, 2330 of the 4317 $Conformation$ predicates can be pruned. For $k = 3$, the size of the new search space is 1,309,476,714, a quite spectacular factor of 10 reduction. Alas, that is still too large for ALKEMY to handle. Is there a way to make it smaller?

Predicate Rewrite Junking A run down of the surviving 1987 $Conformation$ predicates reveals that they are of two kinds, exemplified by the following two predicates.

$$setExists_1\ (\wedge_2\ (proj_1 \circ (\neq -6))\ top)$$
$$l : (39, 63)\ r : (0, 0)\ A_p = 63\ B_p = 102$$

$$setExists_1\ (\wedge_2\ (proj_{143} \circ (= 1))\ top)$$
$$l : (39, 39)\ r : (0, 24)\ A_p = 63\ B_p = 102$$

Here l and r give the class distributions of examples in the left and right subtrees. The value A_p is the accuracy of the predicate and B_p its refinement bound. Notice that the accuracy and the refinement bound values failed to distinguish between the two predicates in any way. To the human eye, however, the second predicate is a lot more interesting than the first. It actually did something, moving 24 molecules to the right.

Building on this observation, we remove from the remaining predicate rewrites all those whose body failed to move at least $l = 3$ (obtained by trial-and-error) molecules to the right. (This is unsafe, but it turns out that no vital information relevant to the classification function is lost in the process.) Only 318 predicate rewrites survived this selection process, resulting in a new search space with only 5,461,280 predicates in it.

An exhaustive search of the predicate space is now within reach, but this is still expensive and, in fact, unnecessary. It is actually now possible to reproduce the result reported in [31] using the SeenSet search algorithm in conjunction with the cutout search strategy. Setting $cutout$ to 10,000, we performed three separate 10-fold cross validations and recorded an average accuracy of 81.94%. Each cross validation took slightly less than half an hour on a computer with an Intel 700 MHz processor. The same final hypothesis as (6.1) was induced on the whole dataset.

It turns out that the result can be improved using the LR search algorithm. Setting $cutout$ to 5000, three 10-fold cross validations yielded an average accuracy of 83.6%. Each cross validation took about 5 minutes on the same 700 Mhz computer. Interestingly, the following slightly different hypothesis was induced on the whole dataset.

$$\begin{array}{l} musk\ m = \\ \quad if\ \ setExists_1\ (\wedge_3\ (proj_{119} \circ (= 1))\ (proj_{122} \circ (\neq -4))\ (proj_{132} \circ (= -2)))\ m \\ \quad then\ 1 \\ \quad else\ \ 0. \end{array} \tag{6.2}$$

A quick check confirms that it behaves identically to (6.1) on the dataset. The higher average accuracy obtained here is due to better consistency across validation folds. In fact, the hypothesis (6.2) was induced in 29 out of the 30 folds. This better consistency can be attributed to the relative stability of the LR search algorithm.

The moral of the story, in my view, is summed up succinctly in this quote taken from the 1975 Turing lecture by Allen Newell and Herbert Simon [153].

> search is a fundamental aspect of a symbol system's exercise of intelli-

gence in problem solving but that amount of search is not a measure of the amount of intelligence being exhibited. What makes a problem a problem is not that a large amount of search is required for its solution, but that a large amount *would* be required if a requisite level of intelligence were not applied.

6.2.4 A Pitfall in Learning with ALKEMY

We end the section with an (interesting) lesson learned from an honest mistake the author made in tackling the Musk problem.

In pursuit of a better result, we can build on the hypothesis (6.1) obtained earlier, repeated here with information on the distribution of examples.

$$\begin{aligned} &\mathit{setExists}_1\ (\wedge_3\ (\mathit{proj}_{29} \circ (\neq -4))\ (\mathit{proj}_{119} \circ (= 1))\ (\mathit{proj}_{132} \circ (= -2))) \\ &l : (32, 9)\ r : (7, 54)\ A_p = 86\ B_p = 95 \end{aligned} \tag{6.3}$$

There are two weaknesses in the hypothesis language used to obtain (6.3). The first is in the representation, where we fixed a priori the intervals on the attributes. The second is in the predicate rewrite system, where we only look at one interval at a time. We can offset these negatives after the event by perturbing the chosen conditions on the three attributes appearing in (6.3). It turns out that the condition on the 119th attribute can't be changed, but the range on the 29th attribute can be narrowed down to $[-3, 2]$ (without any effect on the class distributions), and the range on the 132nd attribute can be weakened to $[-2, -1]$ (with helpful changes in the class distributions). From this small experiment, we obtain the following new hypothesis.

$$\begin{aligned} &\mathit{setExists}_1\ (\wedge_3\ (\mathit{proj}_{29} \circ \wedge_2\ (> -4)\ (< 3)) \\ &\qquad\qquad (\mathit{proj}_{132} \circ \wedge_2\ (> -3)\ (< 0))\ (\mathit{proj}_{119} \circ (= 1))) \\ &l : (38, 15)\ r : (1, 48)\ A_p = 86\ B_p = 101 \end{aligned} \tag{6.4}$$

The distributions of examples and the refinement bounds both suggest rather strongly that (6.4) is to be preferred over (6.3). Further, it seems a good idea to try and strengthen (6.4) in order to move some of the remaining negative examples in the left subtree to the right. Using (6.4) as a basis, we constructed the following predicate rewrite system and experimented with it.

$$\begin{aligned} &\mathit{top} \rightarrowtail \mathit{setExists}_1\ (\wedge_4\ (\mathit{proj}_{29} \circ \wedge_2\ (> -4)\ (< 3))\ (\mathit{proj}_{132} \circ \wedge_2\ (> -3)\ (< 0)) \\ &\qquad\qquad (\mathit{proj}_{119} \circ (= 1))\ \mathit{top}) \\ &\mathit{top} \rightarrowtail \mathit{proj}_i \circ (= j)\ \text{ where }\ i \in \{1, 2, \ldots, 166\} \text{ and } j \in \{-6, -5, \ldots, 6\} \\ &\mathit{top} \rightarrowtail \mathit{proj}_i \circ (\neq j)\ \text{ where }\ i \in \{1, 2, \ldots, 166\} \text{ and } j \in \{-6, -5, \ldots, 6\} \end{aligned}$$

A comparable 10-fold cross-validation result was obtained (see Table 6.1 under entry 4 conjuncts). The hypotheses induced in the individual validation folds all have the same characteristic as predicate (6.4), with good accuracy and high refinement bound.

This prompted us to collect together all the predicates that turned up as the hypothesis in any of the folds and use them to construct a new predicate rewrite system by adding one conjunct to each of them in the same fashion as was done to (6.4). We repeated the process until we reached ten conjuncts, at which point no further improvements in training accuracy can be achieved using the extra top. Table 6.1 shows the 10-fold cross-validation results obtained for the successive iterations.

Conjuncts	4	5	6	7	8	9	10
Train Acc.	87.26	89.33	91.29	92.48	93.57	94.66	94.44
Test Acc.	82.45	84.45	86.36	87.36	87.36	86.36	90.09

Table 6.1: The training and test set accuracies for different numbers of conjuncts.

We have obtained a very impressive cross-validation result (comparable to some of the best efforts on the Musk-2 dataset) with minimal search. But is the result valid? This is debatable. The technique is alright, but surely the accuracy estimation so-computed cannot be trusted. Why?

Consider this next scenario. ALKEMY is used to find a hypothesis with high accuracy, say 98%, on the whole training set. We then proceed to use a predicate rewrite system that consists of only this predicate and do an n-fold cross validation. Voila, we have a world record! This is clearly unacceptable, but it is really not that different from the experiment we have just described. In fact, the first six iterations can be thought of as a systematic search for high-accuracy predicates, and the last iteration the cheating step, where instead of one, we put several hundred high-accuracy predicates in the search space, thinly disguised by many other low-accuracy predicates that we know won't play a big distracting role in search.

The problem is that in ALKEMY, the hypothesis language is an important parameter that can be tuned. Further, by design, we have great control over its exact form. This flexibility opens up many possibilities for one to commit errors in the use of statistical tests to estimate true accuracy. To perform an n-fold cross validation, the correct procedure in this case is to first partition the data into n subsets, and then repeat in each fold the crafting of the predicate rewrite system, all the time using only the current set of training examples available. The test set must not be used in any (indirect) way in the fine-tuning of the hypothesis language.

In fact, our observation here invalidates the numbers reported in the previous section. The results given in [31], however, stand untainted.

It's worth noting that similar (and independent) observations of such malpractices have been reported in other areas of machine learning. See, for example, [1].

6.2.5 An Observation

We end the section with a more positive observation.

Learning with ALKEMY is an iterative process. Given a problem, we first decide on a representation and a hypothesis language based on some initial understanding

of the domain. As more and more experiments are done, and lessons learned, the hypothesis language or even the representation can change. At any one particular iteration, the representation and predicate rewrite system reflects one's understanding of the problem being solved.

6.3 Knowledge Discovery — Predictive Toxicology

In this section, we present a case study in predictive toxicology to bring out the kind of support provided by ALKEMY for knowledge discovery. The main learning problem is introduced in §6.3.1. After a brief discussion on the representation of molecules in §6.3.2, we describe two simple experiments conducted and what we found in each case. An evaluation of the approach is given at the end of the section in §6.3.6.

6.3.1 The 2000-1 Predictive Toxicology Challenge (PTC)

The Challenge is to obtain models that predict the outcome of biological tests for the toxicity of chemicals using information related to chemical structures alone.

There are 417 molecules in the dataset, each labelled with a classification for four different chemical bioassays conducted on male rats (MR), female rats (FR), male mice (MM) and female mice (FM). The dataset was built up over many years from experiments conducted at the US National Toxicology Program. For that reason, there are some inconsistencies in the labellings used. For each of the four types of experiments, a molecule can have any one of the following labels: P (Positive), N (Negative), E (Equivocal), CE (Clear Evidence), SE (Some Evidence), EE (Equivocal Evidence), NE (No Evidence) and IS (Inadequate Study). Early experiments use the P, N and E labels; later experiments refined the classification scheme to CE, SE, EE, and NE.

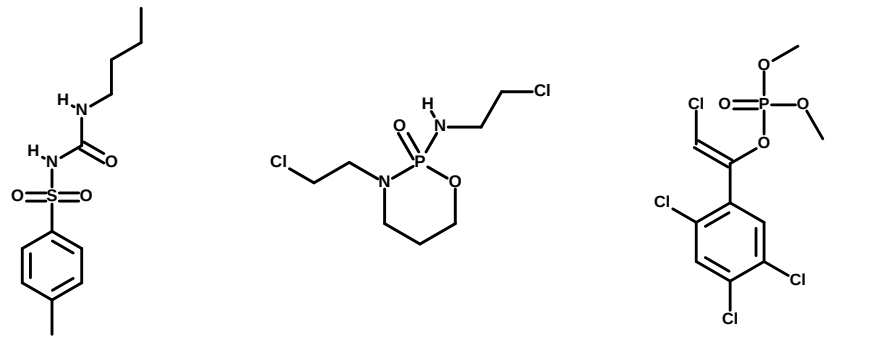
Structural descriptions of the molecules are available in various standard encodings. In addition, seven sets of feature vectors containing chemical descriptors of various kinds are available from the initial data engineering phase of the challenge. Figure 6.1 shows three typical molecules in the dataset together with their classifications.

For the purpose of evaluation, an independent test set of 185 molecules was selected from data collected by the US Food and Drug Administration. Participants were required to make predictions for these molecules based on what they learned from the training data.

The challenge is a largely unsolved problem. The best efforts so far are documented in [21] and [157]. More information about PTC, including the very humbling results produced, can be found in [94], [93] and at

`http://www.predictive-toxicology.org/ptc`.

In fact, the author was a participant in that challenge. The material presented in this section builds on lessons learned from that early attempt [155].



Name: TR031
MR=N, FR=N, MM=N, FM=N

Name: TR032
MR=N, FR=P, MM=N, FM=P

Name: TR033
MR=N, FR=P, MM=P, FM=P

Figure 6.1: Three typical molecules in the PTC dataset

6.3.2 Representation of Individuals

We used the Female Rat dataset for our experiments here as it is the only one that looks remotely learnable from previous attempts by various research groups. The molecules are relabelled as follows: molecules with P, CE and SE labels are classified as positive; molecules with N and NE labels are classified as negative; the remaining molecules with E, EE and IS labels are discarded. The resulting dataset contains 351 molecules, with 121 positive examples and 230 negative examples.

To focus on the structure of the chemicals, we choose not to use the feature vectors in this set of experiments. (This is primarily for convenience; there are many hundreds of features available – most of them uninformative – and knowing which ones to use is a challenging learning task in itself.) We note the limitation of this choice, as pointed out by many authors (see, for example, [166]), and understand that, to generate good models, it is essential that all the available information be used somehow.

As is the usual convention for such problems, we represent each molecule as an undirected graph (see Section 2.5 for the type declaration of graphs).

$$Molecule = Graph\ Element\ Bond$$

We have the following constants for the elements and the bonds.

$$As, Au, B, Ba, Br, C, Ca, \ldots, O, P, Pb, S, Sn, Te, Zn : Element$$
$$S, D, T, R : Bond$$

The four constants for $Bond$ represent, respectively, single bond, double bond, triple bond and resonant bond.

The following is the representation of molecule TR032 (see Figure 6.1).

$$\begin{aligned}TR032 = (\{&(1,C),(2,C),(3,Cl),(4,N),(5,C),(6,C),(7,C),(8,O),(9,P),\\ &(10,O),(11,N),(12,C),(13,C),(14,Cl),(15,H),(16,H),(17,H),\\ &(18,H),(19,H),(20,H),(21,H),(22,H),(23,H),(24,H),(25,H),\\ &(26,H),(27,H),(28,H),(29,H)\},\\ \{&(\langle 11,12\rangle,S),(\langle 12,13\rangle,S),(\langle 13,14\rangle,S),(\langle 9,11\rangle,S),(\langle 9,10\rangle,D),\\ &(\langle 1,15\rangle,S),(\langle 1,16\rangle,S),(\langle 2,17\rangle,S),(\langle 2,18\rangle,S),(\langle 5,19\rangle,S),\\ &(\langle 1,2\rangle,S),(\langle 11,25\rangle,S),(\langle 12,26\rangle,S),(\langle 12,27\rangle,S),(\langle 13,28\rangle,S),\\ &(\langle 13,29\rangle,S),(\langle 5,20\rangle,S),(\langle 6,21\rangle,S),(\langle 6,22\rangle,S),(\langle 7,23\rangle,S),\\ &(\langle 7,24\rangle,S),(\langle 2,3\rangle,S),(\langle 1,4\rangle,S),(\langle 4,5\rangle,S),(\langle 5,6\rangle,S),\\ &(\langle 6,7\rangle,S),(\langle 7,8\rangle,S),(\langle 4,9\rangle,S),(\langle 8,9\rangle,S)\})\end{aligned}$$

The notation $\langle s,t\rangle$ is used as a shorthand for the multiset that takes the value 1 on each of s and t, and is 0 elsewhere, i.e., $\langle s,t\rangle$ is essentially an unordered pair.

The task is to learn a function with the signature $carcinogenic : Molecule \rightarrow \Omega$. We have cast the basic problem as a binary classification task. But the solution to it requires intelligent feature engineering from structural descriptions of molecules, and *that* is a knowledge discovery problem.

In the following, we will liberally use the usual transformations associated with graphs, sets and multisets. The reader is referred to Section 2.5 for their definitions.

6.3.3 A First Experiment

A fairly natural question to ask is whether there exists a correlation between the size of a molecule and its carcinogenicity. Here, we measure the size of a molecule by the number of edges in it. (In an earlier experiment, the number of vertices in a molecule was used; nothing of interest was found in that experiment.) Using the following predicate rewrite system

$$top \rightarrowtail edges \circ domCard\ (top) \circ (=i) \ \text{ where } \ i \in \{1,2,\ldots,50\},$$

we plotted the distribution of positive and negative examples with respect to the number of edges, shown here in Figure 6.2.

From the graph, one is inclined to conclude that, in general, the number of edges in a molecule does not have a strong correlation with its carcinogenicity. There are, however, several interesting observations that can be made from the plot. For example, there is a cluster of positive examples around 9 and 10, and two different groups of negative examples around 13 and 36. It turns out that many interesting patterns and potential structural alerts can be found from a close inspection of these clusters.

For an illustration of the process of discovery, we zoom in onto the group of molecules with exactly ten edges, shown here in Figure 6.3. Notice that the chemicals TR028, TR206 and TR384 all share a bat-shaped structure with group seven atoms

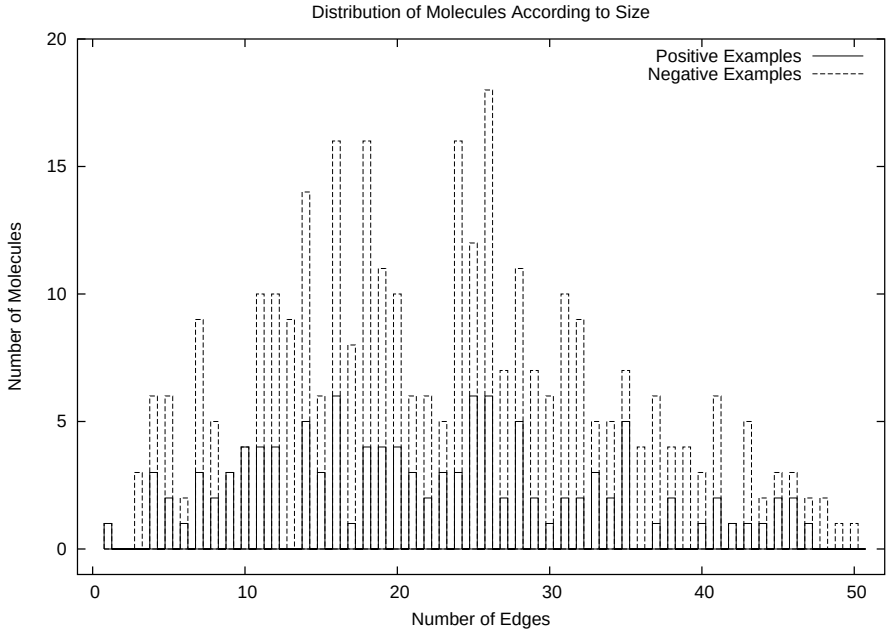


Figure 6.2: Distribution of positive and negative molecules with respect to size

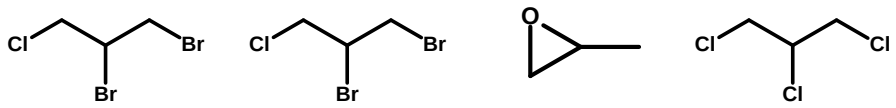


Figure 6.3: From left to right: TR028, TR206, TR267, TR384.

attached to the ends. For ease of reference, we will call this structure the G7-propane structure. It seems plausible to speculate that other similar-looking molecules in the dataset are also carcinogens. To test this conjecture, we formulated the following:

$$
\begin{aligned}
&top \rightarrowtail \wedge_2\ (vertices \circ domCard\ (top) \circ (= 11))\\
&\quad ((subgraphs\ 6) \circ domCard\ (edges \circ \wedge_2\\
&\quad\ (domCard\ (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C))\ (vertex \circ (= C)))\\
&\qquad\qquad (edge \circ (= S))) \circ (= 2))\\
&\quad\ (domCard\ (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C))\\
&\qquad\qquad\qquad\qquad (vertex \circ \vee_2\ (= Br)\ (= Cl)))\\
&\qquad\qquad (edge \circ (= S))) \circ (= 3))) \circ (= 1)).
\end{aligned}
\tag{6.5}
$$

The predicate in the body of (6.5) checks for the existence of two properties that approximate the G7-propane structure:

1. the number of atoms (vertices) in a molecule is exactly eleven; and
2. there exists exactly one connected subgraph of size six in the molecule with two edges connecting two carbons in a single bond, and three edges connecting a carbon to either a bromine or a chlorine in a single bond.

Running ALKEMY with (6.5) shows that only the same three molecules, shown earlier in Figure 6.3 and also in the top row of Figure 6.4, satisfy the predicate. The condition is too strong. Weakening parts of it we obtained the following two predicate rewrites.

$$
\begin{aligned}
&top \rightarrowtail \wedge_2\ (vertices \circ domCard\ (vertex \circ (= Br)) \circ (> 0))\\
&\quad ((subgraphs\ 6) \circ setExists_1\ (edges \circ \wedge_2\\
&\quad\ (domCard\ (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C))\ (vertex \circ (= C)))\\
&\qquad\qquad (edge \circ (= S))) \circ (= 2))\\
&\quad\ (domCard\ (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C))\\
&\qquad\qquad\qquad\qquad (vertex \circ \vee_2\ (= Br)\ (= Cl)))\\
&\qquad\qquad (edge \circ (= S))) \circ (= 2))))
\end{aligned}
\tag{6.6}
$$

$$
\begin{aligned}
&top \rightarrowtail \wedge_2\ (vertices \circ domCard\ (vertex \circ (= Br)) \circ (> 0))\\
&\quad ((subgraphs\ 6) \circ setExists_1\ (edges \circ \wedge_2\\
&\quad\ (domCard\ (connects \circ msetExists_2\ (vertex \circ (= C))\\
&\qquad\qquad\qquad\qquad (vertex \circ (= C))) \circ (= 2))\\
&\quad\ (domCard\ (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C))\\
&\qquad\qquad\qquad\qquad (vertex \circ \vee_2\ (= Br)\ (= Cl)))\\
&\qquad\qquad (edge \circ (= S))) \circ (= 2))))
\end{aligned}
\tag{6.7}
$$

The predicate in the body of (6.6) aims to capture the existence of the G7-propane

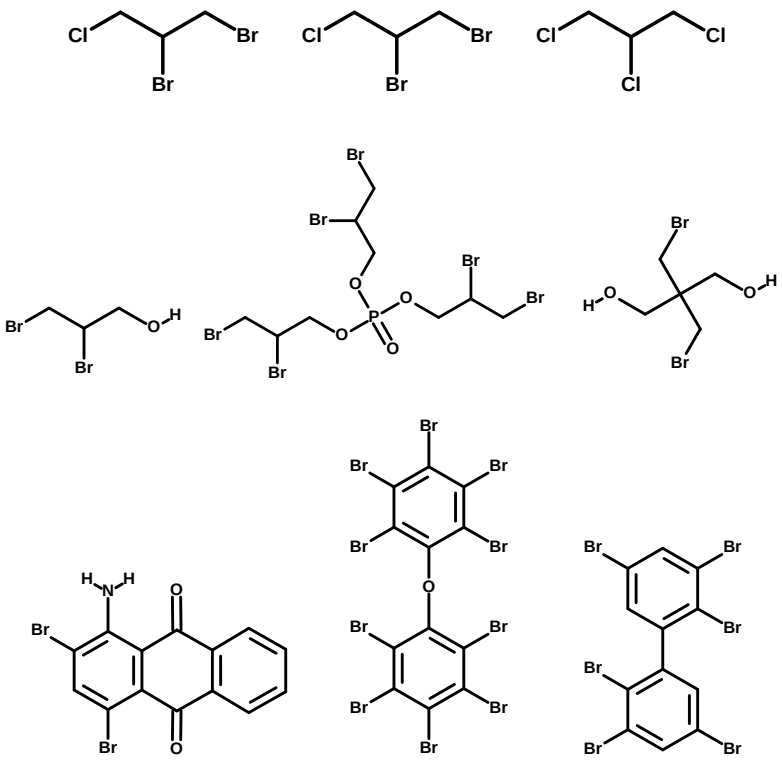


Figure 6.4: From left to right, top to bottom: TR028, TR206, TR384, TR400, TR076, TR452, TR383, TR309, TR398. All these molecules are positive.

structure as a subgraph in larger molecules. We restricted ourselves to molecules with at least one bromine because TR384 appears as a subgraph in many molecules in the dataset, both negative and positive, which is probably evidence that it is not an active substructure. We also relaxed the requirement on the number of edges connecting a carbon to either a bromine or a chlorine to two because demanding the existence of three such edges produces no increased coverage compared to (6.5). Five molecules satisfy this second predicate, including the first two molecules in the top row and the three molecules in the second row in Figure 6.4.

The predicate in the body of (6.7) is similar, but now we relax the type of bond on the edges connecting two carbon atoms. All the molecules in Figure 6.4, except of course TR384, satisfy this predicate. In particular, compared to the second predicate, we gain coverage of the three molecules in the bottom row.

The molecules at the bottom row of Figure 6.4 looks rather different from those in the first row. In a sense, as we proceed from top to bottom, we lose some confidence in our conjecture that the G7-propane structure is the active cancer-causing (sub)structure. The molecules in the bottom row may well be carcinogenic for a different reason; for instance, it is known that molecules that contain bromine(s) have a

high chance of being carcinogenic. Without expert chemical knowledge, a judgement cannot be made on this point. But from a knowledge discovery point of view, we can be quite happy with our progress.

A note on the process of discovery. Notice how we start from a predicate describing specific properties of a few chemicals we observed in the initial experiment and are able to gradually relax the conditions to find other molecules with similar structures, all of which turned out to be carcinogens. This kind of predicate perturbation, whether the aim is to generalize or to specialize an existing condition, is a common and useful operation in knowledge discovery, and is well-supported in ALKEMY.

6.3.4 A Second Experiment

The aim of this experiment is to uncover chemical structures characterized by the co-existence of N different atoms. We show what we found in the case of $N = 3$. The following predicate rewrite system was used.

$$
\begin{aligned}
&top \rightarrowtail vertices \circ \wedge_3\ (setExists_1\ top)\ (setExists_1\ top)\ (setExists_1\ top)\\
&top \rightarrowtail vertex \circ (= As)\\
&top \rightarrowtail vertex \circ (= Au)\\
&\ldots\\
&top \rightarrowtail vertex \circ (= Te)\\
&top \rightarrowtail vertex \circ (= Zn)
\end{aligned}
$$

A lot of information can be gleaned from the outcome of the experiment. We give here two eye-catching combinations. The first is characterized by the co-existence of an oxygen, a phosphorus and a sulfur in a molecule. Figure 6.5 shows the molecules satisfying this property. All of them are negative examples.

By inspection of Figure 6.5, a natural conjecture is that the existence of a substructure with a phosphorus in the middle and four atoms, at least one of which is a sulfur, connected to it, renders a molecule non-carcinogenic.

The second combination is characterized by the co-existence of a chlorine, an oxygen and a sulfur. Figure 6.6 shows the molecules satisfying this property. All of them are negative examples.

There are several interesting patterns in this collection of molecules. Molecules in the first row share the property that each has an O=S=O component. A simple rewrite system can be used to find all the molecules having that substructure in the dataset; they are shown in Figure 6.7. All the molecules except the four in the bottom row are negative.

Another interesting pattern we found in Figure 6.6 is associated with molecule TR015. It is natural to guess that the substructure responsible in this case is the small T-structure with three chlorine atoms hanging off it. Generalizing chlorine to common

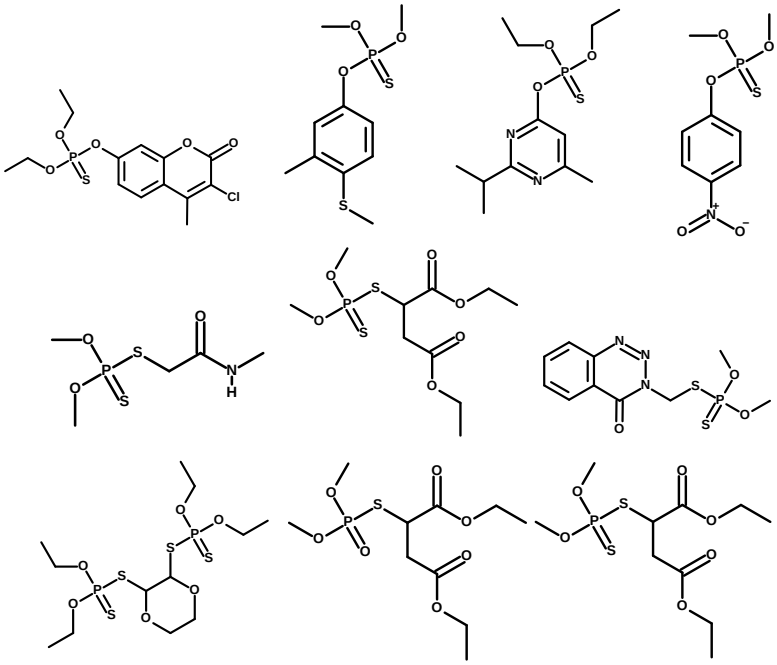


Figure 6.5: From left to right, top to bottom: TR096, TR103, TR137, TR157, TR004, TR024, TR069, TR125, TR135, TR192. All the molecules are negative.

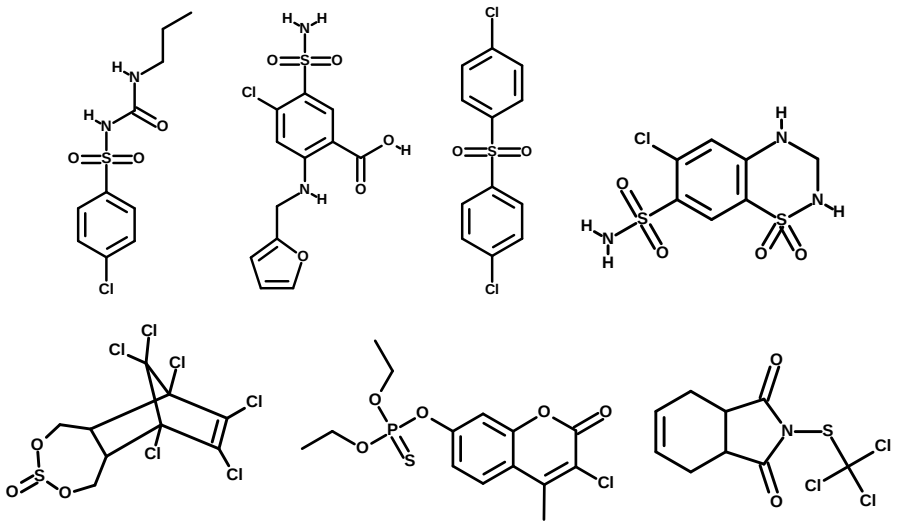


Figure 6.6: From left to right, top to bottom: TR045, TR356, TR501, TR357, TR062, TR096, TR015. All the molecules are negative.

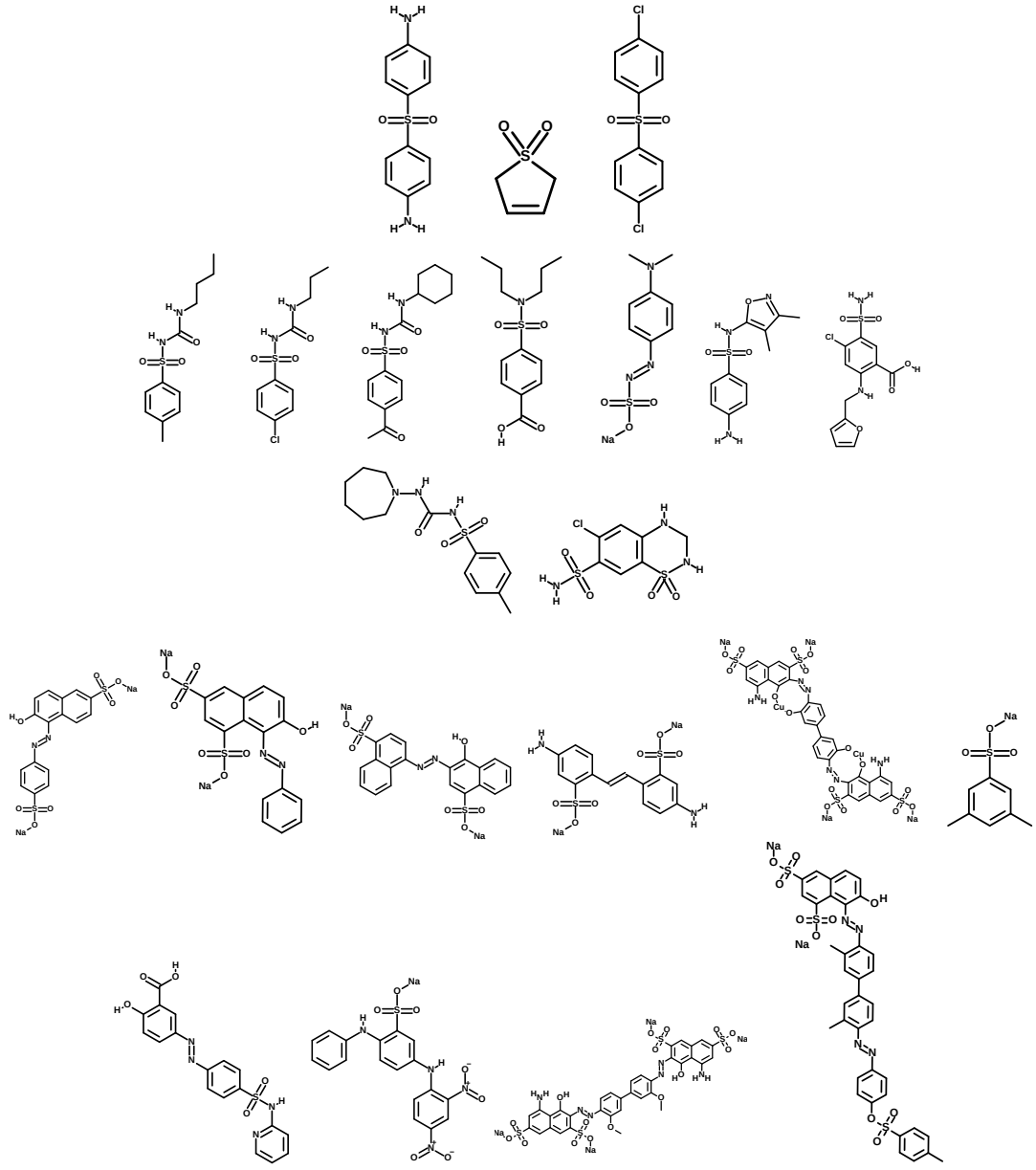


Figure 6.7: From left to right, top to bottom: TR020, TR102, TR501, TR031, TR045, TR050, TR395, TR101, TR138, TR356, TR051, TR357, TR208, TR211, TR220, TR412, TR430, TR464, TR457, TR335, TR397, TR405.

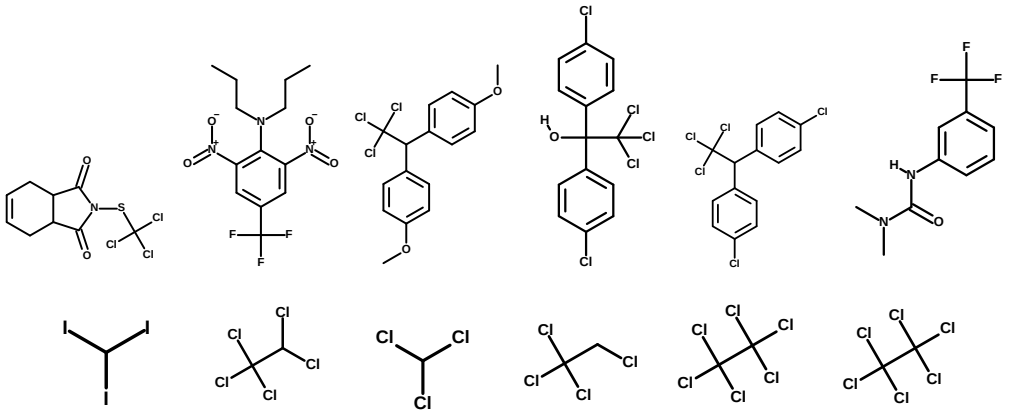


Figure 6.8: From left to right, top to bottom: TR015, TR034, TR035, TR090, TR131a, TR195, TR110, TR232, TR000, TR237, TR361, TR068

group seven atoms, we get the following predicate rewrite system:

$$\begin{aligned} top \rightarrowtail\ & (subgraphs\ 4) \circ setExists_1\ (edges \circ domCard \\ & (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C)) \\ & (vertex \circ \vee_3\ (= Cl)\ (= F)\ (= I))) \\ & (edge \circ (= S))) \circ (= 3))). \end{aligned} \tag{6.8}$$

The molecules satisfying the body of (6.8) are shown in Figure 6.8. Interestingly, but perhaps unsurprisingly, they are all negative examples.

6.3.5 Other Features

The two experiments described above serve as good illustrations of the process of knowledge discovery. Figure 6.9 shows seven groups of positive molecules that share a particular structure. These patterns were discovered over a period of three to four weeks, and are the result of very many experiments with different ideas using the same process. The suspected structure in each case should be obvious.

6.3.6 Evaluation

To evaluate our approach to PTC, we collected all the features identified in one predicate rewrite system, given in full in §6.3.7, and learned a decision list that obtains 70.9% accuracy on the training set. The decision list is used to make predictions on the independent test set as follows: if a molecule falls through the list and ended up in the default node, no prediction is made; otherwise, a prediction is made in the usual fashion. Out of the 185 molecules in the test set, ALKEMY was confident enough to make a prediction only on 28 molecules; but out of the 28 predictions made, 24 were

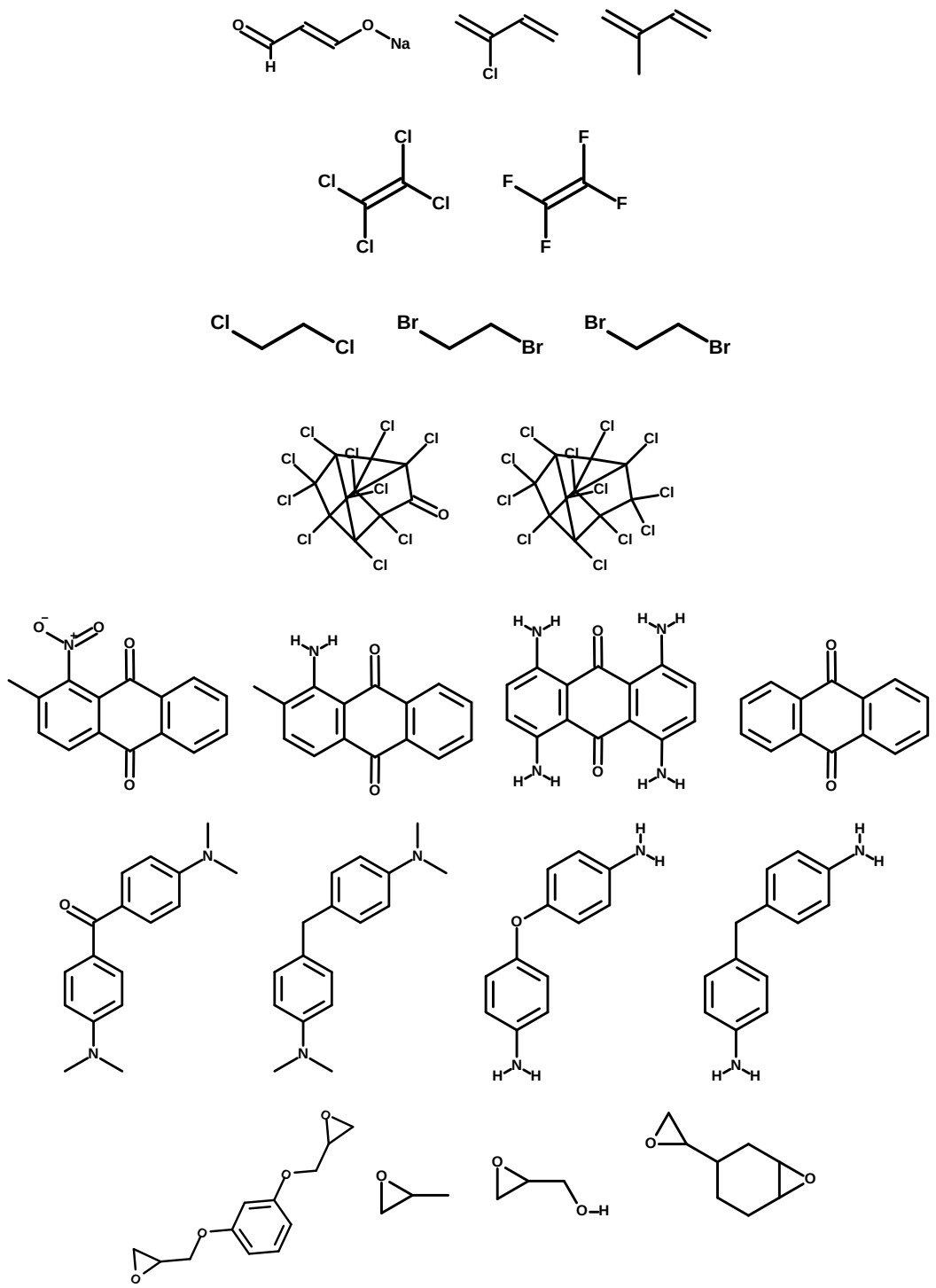


Figure 6.9: From left to right, top to bottom: TR331, TR467, TR486, TR311, TR450, TR055, TR086, TR210, TR001, TR313, TR029, TR111, TR299, TR494, TR181, TR186, TR205, TR248, TR257, TR267, TR374, TR362

correct. In terms of accuracy, that is 85%, a respectable result by any measure on this very difficult problem. The poor coverage can be put down to the fact that the test set is (totally) unrepresentative of the training set, as pointed out in [15].

To properly assess the value of this knowledge discovery exercise, the meaningfulness of the structural features and the carcinogenicity theory we found, from a bio-chemistry perspective, has to be established. Having little knowledge in chemistry, we consulted an expert from the US Environmental Protection Agency. Given the known difficulty in explaining non-carcinogenicity, our discussion focused on the carcinogens found. It turns out that almost all the substructures we have identified in positive examples are indeed long-recognized structural features of potent carcinogens, and most of them can be explained mechanistically. We quote some of the explanations given by the domain expert here. More complete descriptions of the mechanisms can be found in the book series [5].

Row 1 in Figure 6.4 and row 3 in Figure 6.9 Chlorine and bromine are good leaving groups, particularly if they are terminal and mono. Disubstitution on adjacent carbons (vicinal substitution) further enhances carcinogenic potential because of the crosslinking potential and additional metabolic activation by gluthathione conjugation pathway.

Row 1 in Figure 6.9 The first chemical is alpha, beta-unsaturated aldehyde, which is reactive. The double bond helps to modulate its reactivity. The other two have terminal double bonds that can be metabolically activated to epoxides as alkylating agent.

Row 2 in Figure 6.9 These are halogenated ethylene that can be epxodized as a metabolic activation pathway.

Row 4 in Figure 6.9 These are polyhalogenated hydrocarbons, known to be moderately active carcinogens.

Row 5 in Figure 6.9 The first three molecules are amino or nitro anthraquinones. The amino/nitro group can be metabolically activated to nitrenium ion to bind to DNA. The anthraquinone moiety may intercalate into DNA. The last chemical is expected to be much weaker; it can only intercalate.

Row 6 in Figure 6.9 These are typical aromatic amines, the best-known class of carcinogens, with favourable position and substitution.

Row 7 in Figure 6.9 They are epoxides, a well-known class of carcinogens.

This is clearly a very positive result, but one that requires qualification. At best, this set of experiments can be interpreted as a proof of concept that trying to learn carcinogenicity theories from examples is not a completely meaningless exercise, and as we have shown here, in fact, some simple theories can be found. But simple theories they are, nothing more than that. Compared to expert systems like DEREK [176] and OncoLogic [201], inductive learning systems still have some way to go in terms of level of sophistication, especially when it comes to explanatory power and the incorporation of the vast available chemical knowledge in making predictions.

The discovery process we have just described is admittedly ad hoc. One can of

course be more systematic about it. For example, it is easy to generalize association rule mining algorithms from data mining (see, for example, [90]) to our knowledge representation setting for use in feature construction. In fact, this idea has been explored in ILP, see for example [108]. Other feature selection methods can be used as well. The expectation, however, is that automated approaches can only reveal statistical information. Educated guesses of structural alerts and their precise formulations still require human input. This is not an undesirable feature of the system. After all, the expressiveness of the knowledge representation formalism is designed with human usage and consumption in mind in the first place.

6.3.7 A Predicate Rewrite System for PTC

$$top \rightarrowtail vertices \circ domCard\ (vertex \circ (= Br)) \circ (> 0)$$

$$top \rightarrowtail vertices \circ domCard\ (vertex \circ (= Br)) \circ (> 1)$$

$$top \rightarrowtail vertices \circ domCard\ (vertex \circ (= Br)) \circ (> 2)$$

$$\begin{aligned} top \rightarrowtail\ & \wedge_2\ (vertices \circ domCard\ (vertex \circ (= Br)) \circ (> 1)) \\ & ((subgraphs\ 6) \circ setExists_1\ (vertices \circ \wedge_2 \\ & \quad (domCard\ (vertex \circ (= C)) \circ (= 3)) \\ & \quad (domCard\ (vertex \circ \vee_2\ (= Br)\ (= Cl)) \circ (= 2)))) \end{aligned}$$

$$\begin{aligned} top \rightarrowtail\ & \wedge_2\ (vertices \circ domCard\ (vertex \circ (= Br)) \circ (> 1)) \\ & ((subgraphs\ 4) \circ setExists_1\ (edges \circ \wedge_2 \\ & \quad (domCard\ (\wedge_2 \\ & \qquad (connects \circ msetExists_2\ (vertex \circ (= C))\ (vertex \circ (= C))) \\ & \qquad (edge \circ (= S))) \circ (= 1)) \\ & \quad (domCard\ (\wedge_2 \\ & \qquad (connects \circ msetExists_2\ (vertex \circ (= C)) \\ & \qquad\qquad (vertex \circ \vee_2\ (= Cl)\ (= Br))) \\ & \qquad (edge \circ (= S))) \circ (= 2)))) \end{aligned}$$

$$\begin{aligned} top \rightarrowtail\ & \wedge_2\ (vertices \circ \wedge_2\ (domCard\ (vertex \circ (= Br)) \circ (> 0)) \\ & \qquad (domCard\ (vertex \circ (= C)) \circ (= 1))) \\ & (edges \circ domCard\ (\wedge_2 \\ & \quad (connects \circ msetExists_2\ (vertex \circ (= C)) \\ & \qquad\qquad (vertex \circ \vee_2\ (= Cl)\ (= Br))) \\ & \quad (edge \circ (= S))) \circ (= 3)) \end{aligned}$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ domCard\ (top) \circ (=8)) \\
\quad (subgraphs\ 8) \circ setExists_1\ (edges \circ \wedge_3 \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (=C))\ (vertex \circ (=C))) \\
\quad\quad\quad (edge \circ (=S))) \circ (=1)) \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (=C))\ (vertex \circ (=H))) \\
\quad\quad\quad (edge \circ (=S))) \circ (=4)) \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (=C)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (vertex \circ \vee_2\ (=Cl)\ (=Br))) \\
\quad\quad\quad (edge \circ (=S))) \circ (=2))))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ domCard\ (vertex \circ (=Cl)) \circ (=6)) \\
\quad ((subgraphs\ 3) \circ setExists_1\ (edges \circ \wedge_2 \\
\quad\quad (domCard\ (connects \circ msetExists_2\ (vertex \circ (=C)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (vertex \circ (=C))) \circ (=1)) \\
\quad\quad (domCard\ (connects \circ msetExists_2\ (vertex \circ (=C)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (vertex \circ (=O))) \circ (=2))))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ domCard\ (vertex \circ (=Cl)) \circ (=0)) \\
\quad ((subgraphs\ 3) \circ setExists_1\ (edges \circ \wedge_2 \\
\quad\quad (domCard\ (connects \circ msetExists_2\ (vertex \circ (=C)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (vertex \circ (=C))) \circ (=1)) \\
\quad\quad (domCard\ (connects \circ msetExists_2 (vertex \circ (=C)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (vertex \circ (=O))) \circ (=2))))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail (subgraphs\ 3) \circ setExists_1\ (edges \circ \wedge_2 \\
\quad\quad (domCard\ (connects \circ msetExists_2\ (vertex \circ (=C)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (vertex \circ (=C))) \circ (=1)) \\
\quad\quad (domCard\ (connects \circ msetExists_2\ (vertex \circ (=C)) \\
\quad\quad\quad\quad\quad\quad\quad\quad (vertex \circ (=O))) \circ (=2)))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ setExists_1\ (vertex \circ \vee_3\ (=F)\ (=Cl)\ (=I))) \\
\quad ((subgraphs\ 4) \circ setExists_1\ (edges \circ domCard\ (\wedge_2 \\
\quad\quad (connects \circ msetExists_2 \\
\quad\quad\quad (vertex \circ (=C))\ (vertex \circ \vee_3\ (=Cl)\ (=F)\ (=I))) \\
\quad\quad (edge \circ (=S))) \circ (=3)))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ domCard\ (vertex \circ (=C)) \circ\ \wedge_2\ (\geqslant 14)\ (\leqslant 17)) \\
\qquad (edges \circ \wedge_3 \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=C))\ (vertex \circ (=C))) \\
\qquad\qquad (edge \circ (=S))) \circ\ \vee_2\ (=4)\ (=5)) \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=C))\ (vertex \circ (=C))) \\
\qquad\qquad (edge \circ (=R))) \circ (=12)) \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=C))\ (vertex \circ (=O))) \\
\qquad\qquad (edge \circ (=D))) \circ (=2)))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ setExists_1\ (vertex \circ (=P))) \\
\qquad ((subgraphs\ 5) \circ setExists_1\ (edges \circ \wedge_2 \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=P))\ (vertex \circ (=O))) \\
\qquad\qquad (edge \circ (=S))) \circ (=3)) \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=P))\ (vertex \circ (=S))) \\
\qquad\qquad (edge \circ (=D))) \circ (=1))))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ setExists_1\ (vertex \circ (=P))) \\
\qquad ((subgraphs\ 5) \circ setExists_1\ (edges \circ \wedge_3 \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=P))\ (vertex \circ (=O))) \\
\qquad\qquad (edge \circ (=S))) \circ (=2)) \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=P))\ (vertex \circ (=S))) \\
\qquad\qquad (edge \circ (=S))) \circ (=1)) \\
\qquad\quad (domCard\ (\wedge_2 \\
\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (=P)) \\
\qquad\qquad\qquad\qquad\qquad (vertex \circ\ \vee_2\ (=O)\ (=S))) \\
\qquad\qquad (edge \circ (=D))) \circ (=1))))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_3\ (vertices \circ setExists_1\ (vertex \circ (= P))) \\
\quad ((subgraphs\ 5) \circ setExists_1\ (edges \circ \wedge_2 \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (= P))\ (vertex \circ (= O))) \\
\quad\quad\quad (edge \circ (= S))) \circ (= 3)) \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (= P))\ (vertex \circ (= O))) \\
\quad\quad\quad (edge \circ (= D))) \circ (= 1)))) \\
\quad\quad (edges \circ setExists_2 \\
\quad\quad\quad (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C))\ (vertex \circ (= C))) \\
\quad\quad\quad\quad (edge \circ (= S))) \\
\quad\quad\quad (\wedge_2\ (connects \circ msetExists_2\ (vertex \circ (= C))\ (vertex \circ (= Cl))) \\
\quad\quad\quad\quad (edge \circ (= S))))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_2\ (vertices \circ setExists_1\ (vertex \circ (= P))) \\
\quad ((subgraphs\ 5) \circ setExists_1\ (edges \circ \wedge_2 \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (= P))\ (vertex \circ (= O))) \\
\quad\quad\quad (edge \circ (= S))) \circ (= 3)) \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (= P))\ (vertex \circ (= O))) \\
\quad\quad\quad (edge \circ (= D))) \circ (= 1))))
\end{array}
$$

$$
\begin{array}{l}
top \rightarrowtail \wedge_3\ (vertices \circ domCard\ (top) \circ (\leqslant 10)) \\
\quad (edges \circ domCard\ (edge \circ (= D)) \circ (= 2)) \\
\quad ((subgraphs\ 4) \circ setExists_1\ (edges \circ \wedge_2 \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (= C))\ (top)) \\
\quad\quad\quad (edge \circ (= D))) \circ (= 2)) \\
\quad\quad (domCard\ (\wedge_2 \\
\quad\quad\quad (connects \circ msetExists_2\ (vertex \circ (= C))\ (vertex \circ (= C))) \\
\quad\quad\quad (edge \circ (= S))) \circ (= 1))))
\end{array}
$$

$$
\begin{aligned}
&top \rightarrowtail \wedge_2\ (vertices \circ setExists_1\ (vertex \circ (= S))) \\
&\qquad ((subgraphs\ 5) \circ setExists_1\ (edges \circ \wedge_2 \\
&\qquad\quad (domCard\ (\wedge_2 \\
&\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (= O))\ (vertex \circ (= S))) \\
&\qquad\qquad (edge \circ (= D))) \circ (= 2)) \\
&\qquad\quad (domCard\ (\wedge_2 \\
&\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (= C))\ (vertex \circ (= S))) \\
&\qquad\qquad (edge \circ (= S))) \circ (= 2))))
\end{aligned}
$$

$$
\begin{aligned}
&top \rightarrowtail \wedge_2\ (vertices \circ setExists_1\ (vertex \circ (= S))) \\
&\qquad ((subgraphs\ 4) \circ setExists_1\ (edges \circ \wedge_2 \\
&\qquad\quad (domCard\ (\wedge_2 \\
&\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (= O))\ (vertex \circ (= S))) \\
&\qquad\qquad (edge \circ (= D))) \circ (= 2)) \\
&\qquad\quad (domCard\ (\wedge_2 \\
&\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (= N))\ (vertex \circ (= S))) \\
&\qquad\qquad (edge \circ (= S))) \circ (= 1))))
\end{aligned}
$$

$$
\begin{aligned}
&top \rightarrowtail \wedge_2\ (vertices \circ setExists_1\ (vertex \circ (= S))) \\
&\qquad ((subgraphs\ 4) \circ setExists_1\ (edges \circ \wedge_2 \\
&\qquad\quad (domCard\ (\wedge_2 \\
&\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (= O))\ (vertex \circ (= S))) \\
&\qquad\qquad (edge \circ (= D))) \circ (= 2)) \\
&\qquad\quad (domCard\ (\wedge_2 \\
&\qquad\qquad (connects \circ msetExists_2\ (vertex \circ (= O))\ (vertex \circ (= S))) \\
&\qquad\qquad (edge \circ (= S))) \circ (= 1))))
\end{aligned}
$$

6.4 Relational Reinforcement Learning — Blocks World

In this section, we explore the suitability of reinforcement learning [187] as a basis for an adaptive agent architecture. Specifically, we take up the approach of relational reinforcement learning proposed in [68] and [69]. The key idea of this approach is to use a symbolic learning system (instead of a non-symbolic system like a neural network) for function approximation. There are several advantages in doing this. To begin with, a symbolic learning system provides a convenient way to incorporate domain knowledge that can be used to greatly reduce the size of the search problems associated with reinforcement learning. Furthermore, the approximated functions, being in symbolic form, are comprehensible and amenable to explicit manipulation.

An outline of the section is as follows. We present the basic framework and the

agent algorithm in §6.4.1. The blocks world domain in which we study the agent architecture is described in §6.4.2. A few interesting experiments are presented in §6.4.3. We end with some discussions in §6.4.4.

The material presented here, first reported in [49], is joint-work with Joshua Cole and John W. Lloyd. Joshua Cole implemented the agent and carried out the experiments. Extensions and modifications to ALKEMY needed in support of this application, including regression-tree learning and on-line learning, were supplied by the author. We built on ideas presented in [183] in the design of the hypothesis language for blocks world. The hypothesis language used is significantly more expressive and relevant than the one adopted in [69].

6.4.1 The Basic Framework

The agent architecture is based on Markov decision processes. We assume discrete time, so there is a set T of time steps of the form $\{0, 1, 2, \ldots\}$.

Definition 6.4.1. A *Markov decision process* consists of the following:

1. a finite set S_t of states, for each $t \in T$.
2. a finite set A_t of actions, for each $t \in T$.
3. for each state $s_t \in S_t$ and each action $a_t \in A_t$, a transition probability distribution $p_t(\cdot \mid s_t, a_t)$, for each $t \in T$.
4. a reward function $r_t : S_t \times A_t \to \mathbb{R}$, for each $t \in T$.

Note that the states, actions, transition probability distribution, and the reward function are all indexed by the time t. At time t, the agent perceives the current state $s_t \in S_t$ and then chooses from amongst the legal actions an action $a_t \in A_t$. The next percept from the environment gives the reward $r_t(s_t, a_t)$ to the agent and the next state is $s_{t+1} \in S_{t+1}$ with probability $p_t(s_{t+1} \mid s_t, a_t)$. It is assumed that the agent does not know the transition probability distribution nor the reward function.

A solution to a Markov decision process is a *policy sequence* of the form

$$\pi = [\pi_0, \pi_1, \pi_2, \ldots]$$

where each π_t is a function called a *policy* from S_t to A_t. Given the current state $s_t \in S_t$ at time t, the action prescribed by the policy sequence is $a_t = \pi_t(s_t)$. To find a good policy sequence, we need a way to evaluate their quality. We define the *discounted total reward* $V^\pi(s)$ by following a policy sequence π from an arbitrary initial state s as

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t) \mid \pi, s_0 = s\right],$$

where γ is a discount factor satisfying $0 \leq \gamma < 1$. Given an initial state s, an optimal policy sequence π_s^* can then be defined as

$$\pi_s^* = \arg\max_\pi V^\pi(s).$$

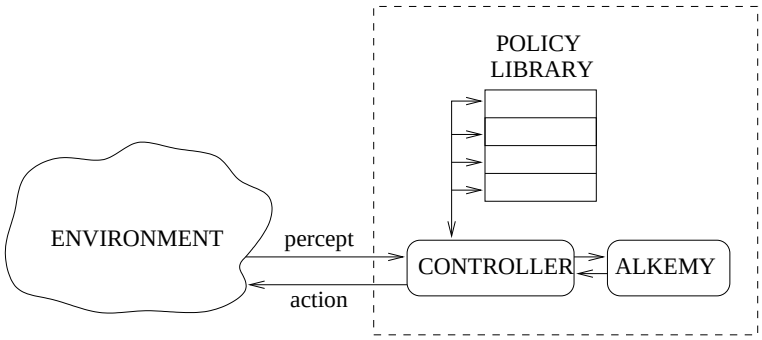


Figure 6.10: The agent architecture

Thus the agent attempts to find a policy that maximizes its discounted total reward given some initial state.

We next present a preliminary design of an adaptive agent architecture to solve the problem. The underlying adaptation algorithm is a form of Q-learning [197] with function approximation [18]. The approach taken is motivated by the work on relational reinforcement learning in [68], [69] and [64]. We represent states, actions, and value functions symbolically. For function approximation, we use (the on-line algorithms of) ALKEMY.

An important innovation introduced in [69], which we adopt in our architecture, is P-learning. The basic idea is that one can associate with a Q function a boolean-valued policy function P defined as follows:

$$P(s,a) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q(s,a'); \\ 0 & \text{otherwise.} \end{cases}$$

The main observation is that the Q function explicitly encodes the path length to a goal from a given state, and this is complex and specific to particular worlds. By computing P from Q, one obtains a more compact representation of the policy, and this has obvious implications for the stability and predictive power of the policy. Furthermore, by carefully crafting the hypothesis languages for P and Q, it is possible to achieve generalization across problem instances. In other words, given a class of tasks of similar nature, one can pick an instance from the class, train the agent on that instance to obtain a P function via Q-learning, and then (re)use P as a generic policy to solve other problems in the same class. This phenomenon will be elaborated further in §6.4.3.

Figure 6.10 shows an overview of the proposed agent architecture. The agent is equipped with a policy library. There is a policy for each kind of task the agent can perform. Each policy is encoded using an Alkemic decision tree, coupled with an hypothesis search space and constraints on how the tree can be modified. The agent interacts with the world in the usual manner, by perceiving the environment

and performing actions. From these interactions, training examples are generated which ALKEMY uses to update the policies in the library to improve its performance. The revision of policies is guided by the learning of Q functions, which provide vital information about the quality of policies otherwise unavailable to the agent.

Figure 6.11 gives the agent algorithm. Given a task $\mathcal{T}$, the agent selects from the library a policy that matches the problem and uses it to initialize P and Q. It then goes into a loop, performing actions by a trade-off between exploitation of P and exploration of the state space, collecting rewards and observing the effects of its actions. Training examples are generated to update the P and Q functions in each iteration. Note that action selection and the two update functions are parameters in the algorithm. Depending on the situation, one may prefer to do more or less exploration. Further, one may choose different ways and frequencies of updating the P and Q functions. In §6.4.3, we explore different instantiations of these functions for different scenarios.

6.4.2 Blocks World

We use blocks world, a simple yet sufficiently rich domain, as a test bed for the proposed agent architecture. Here are some declarations suitable for this domain.

$$
\begin{aligned}
&B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, \mathit{Floor} : \mathit{Object}\\
&\mathit{Stack} = \mathit{List}\ \mathit{Object}\\
&\mathit{World} = \{\mathit{Stack}\}\\
&\mathit{OnState} = \mathit{Object} \times \mathit{Object}\\
&\mathit{Intention} = \{\mathit{OnState}\}\\
&\mathit{Action} = \mathit{Object} \times \mathit{Stack}\\
&\mathit{State} = \mathit{World} \times \mathit{Intention}\\
&\mathit{Individual} = \mathit{State} \times \mathit{Action}.
\end{aligned}
$$

The number of blocks in the world varies from time to time, but we never use more than ten blocks. A blocks world is modelled as a set of stacks of blocks. The agent's intention is modelled as a set of on-states, where an *on-state* is a pair of objects, the first of which is intended to be immediately on top of the second. An action specifies that some block that is clear should be put on top of some stack. The empty stack is allowed and is another representation of the floor, so that moving a block to the empty stack is actually moving the block to the floor. A state is a pair consisting of a blocks world and an intention. The interpretation is that the agent lives in the world and intends to achieve the state specified by the intention. In our experiments, the intentions are provided externally by percepts that the agent immediately accepts as intentions. Finally, an individual is a pair consisting of a state and an action. Several functions below whose domain is the set of individuals are the subject of learning.

A block in the world component of an individual is *misplaced* if its position in the world is inconsistent with the on-states specified in the intention component of

Algorithm $Agent(\mathcal{T})$
input: $\mathcal{T}$, a task;

$t :=$ current time;
$s_t :=$ current state;
$P_t := policyLibrary[\mathcal{T}].P$;
$Q_t := policyLibrary[\mathcal{T}].Q$;
while $\mathcal{T}$ not done

$a_t := selectAction(s_t, P_t)$;
perform a_t;
observe $r = r_t(s_t, a_t)$ and s_{t+1};
$x := ((s_t, a_t),\ r + \gamma \max_{a \in A_{t+1}} Q_t(s_{t+1}, a))$;
$Q_{t+1} := updateQ(Q_t, \{x\})$;
$X := \emptyset$;
for each $a \in A_t$ **do**

if $a = \arg\max_{a' \in A_t} Q_{t+1}(s_t, a')$
then $x := ((s_t, a), 1)$;
else $x := ((s_t, a), 0)$;
$X := X \cup \{x\}$;

$P_{t+1} := updateP(P_t, X)$;
$t := t + 1$;

$policyLibrary[\mathcal{T}].P := P_t$;
$policyLibrary[\mathcal{T}].Q := Q_t$;

Figure 6.11: The agent algorithm

the individual. An action is *constructive* if after the move of the block specified by the action, neither the block nor any block underneath it is misplaced and the move achieves an on-state in the intention. Once a block has been moved constructively, it need not be moved again in the course of achieving the (rest of the) intention. An individual is *deadlocked* if no constructive move is possible with respect to the world and intention components of that individual.

Each Q_t function has signature

$$Q_t : \mathit{Individual} \rightarrow \mathbb{R}.$$

The hypothesis language for each Q_t function has a single domain-specific transformation that has signature

$$\mathit{estimatedPathLength} : \mathit{Individual} \rightarrow \mathit{Int}$$

and is defined as follows. Suppose, in the individual, the action is to move block A on top of some stack that has block B at the top. Then the value of $\mathit{estimatedPathLength}$ for that individual is given by

$$2 \times \text{number of misplaced blocks in the world}$$
$$+ \begin{cases} -1 & \text{if } A \text{ is intended to be on } B \\ +1 & \text{if } A \text{ is intended to be on } C\,(\neq B) \text{ or} \\ & C\,(\neq A) \text{ is intended to be on } B \\ 0 & \text{otherwise} \end{cases}$$
$$+ \begin{cases} -1 & \text{if } A \text{ is misplaced} \\ 0 & \text{otherwise} \end{cases}$$
$$+ \begin{cases} +1 & \text{if } B \text{ is misplaced} \\ 0 & \text{otherwise.} \end{cases}$$

The function $\mathit{estimatedPathLength}$ is intended to provide an estimate of the shortest path to a goal (that is, a state satisfying the intention) from the state that results by applying the action to the current state. Such an estimate is needed to approximate each Q_t function. Note that using a transformation such as $\mathit{estimatedPathLength}$ implies that the agent has some limited knowledge of the effects of its actions. Here is the predicate rewrite system for the Q_t hypothesis language:

$$\mathit{top} \rightarrowtail \mathit{estimatedPathLength} \circ (= i) \;\text{ where }\; i \in \{0, 1, \ldots, 21\}.$$

Each Q_t function for a world in which there are N blocks, represented as a regres-

sion tree, typically has the following general structure.

$$
\begin{array}{l}
Q\ x = \mathit{if}\ \mathit{estimatedPathLength} \circ (= 0)\ x \\
\quad \mathit{then}\ q_0 \\
\quad \mathit{else\ if}\ \mathit{estimatedPathLength} \circ (= 1)\ x \\
\quad\quad \mathit{then}\ q_1 \\
\quad\quad\quad \ldots \\
\quad\quad \mathit{else\ if}\ \mathit{estimatedPathLength} \circ (= 2N+1)\ x \\
\quad\quad\quad \mathit{then}\ q_{2N+1} \\
\quad\quad\quad \mathit{else}\ q_{otherwise},
\end{array}
$$

where q_0, q_1, and so on, are the various regression values.

Now we turn attention to the policy function. Instead of directly learning a policy function

$$\pi_t : \mathit{State} \rightarrow \mathit{Action},$$

a policy *relation* having signature

$$\mathit{policy}_t : \mathit{Individual} \rightarrow \Omega$$

is learned. To determine the policy function π_t from policy_t, we proceed as follows. Let s be a state. For each action a, determine the value of $\mathit{policy}_t\,(s, a)$. If there is at least one a for which $\mathit{policy}_t\,(s, a) = 1$, choose an a arbitrarily amongst all such actions. Otherwise, choose a arbitrarily amongst all possible actions. Depending on the nature of policy_t, the function π_t may thus be non-deterministic.

The hypothesis language for the policy relation requires a number of domain-specific transformations.

The transformation

$$\mathit{extractAction} : \mathit{Individual} \rightarrow \mathit{Object} \times \mathit{Stack} \times \mathit{Individual}$$

takes an individual as input and returns the triple consisting of the block in the action, the stack in the action, and the individual.

The transformation

$$\mathit{projA} : \mathit{Object} \times \mathit{Stack} \times \mathit{Individual} \rightarrow \mathit{Object} \times \mathit{Individual}$$

takes as input a triple consisting of a block, a stack, and an individual, and returns the pair consisting of the block and the individual.

The transformation

$$\mathit{projB} : \mathit{Object} \times \mathit{Stack} \times \mathit{Individual} \rightarrow \mathit{Stack} \times \mathit{Individual}$$

takes as input a triple consisting of a block, a stack, and an individual, and returns the

pair consisting of the stack and the individual.

The transformation

$$noMisplacedNotOnFloorBlock : Individual \to \Omega$$

takes as input an individual and returns true, if there is no block in the world component of the individual that is misplaced and not on the floor; and returns false, otherwise.

The transformation

$$isDeadlocked : Individual \to \Omega$$

takes as input an individual and returns true, if there is no constructive move possible in the world component of the individual; and returns false, otherwise.

The transformation

$$isMisplaced : Object \times Individual \to \Omega$$

takes as input a pair consisting of a block and an individual and returns true, if the block is misplaced in the world component of the individual; and returns false, otherwise.

The transformation

$$isFloor : Stack \times Individual \to \Omega$$

takes as input a pair consisting of a stack and an individual and returns true, if the stack is empty; and returns false, otherwise.

The transformation

$$isConstructive : Object \times Stack \times Individual \to \Omega$$

takes as input a triple consisting of a block, a stack, and an individual, and returns true if moving the block to the stack in the world component of the individual is constructive; and returns false, otherwise.

The transformations we have introduced thus far will be used to form predicate rewrite systems for P learning in §6.4.3. Here we see how they can be used to code up some simple but effective policies for blocks world. We consider two policies, US and $GN1$, that were studied in [183], and a more restricted one, $simple$.

The $simple$ policy either makes a constructive move or else moves a block to the

floor. Here is the *simple* policy as an Alkemic decision tree.

$$\begin{aligned} policy_{simple}\ x = {} & \mathit{if}\ extractAction \circ isConstructive\ x \\ & \mathit{then}\ 1 \\ & \mathit{else\ if}\ \ extractAction \circ projB \circ isFloor\ x \\ & \quad \mathit{then}\ 1 \\ & \quad \mathit{else}\ \ 0. \end{aligned}$$

The *US* (Unstack-Stack) policy puts all misplaced blocks on the floor first and then builds the goal state by constructive moves. Here is the *US* policy as a decision tree.

$$\begin{aligned} & policy_{US}\ x = \\ & \quad \mathit{if}\ \wedge_2\ (noMisplacedNotOnFloorBlock)\ (extractAction \circ isConstructive)\ x \\ & \quad \mathit{then}\ 1 \\ & \quad \mathit{else\ if}\ \wedge_2\ (extractAction \circ projA \circ isMisplaced)\ (extractAction \circ projB \circ isFloor)\ x \\ & \qquad \mathit{then}\ 1 \\ & \qquad \mathit{else}\ \ 0. \end{aligned}$$

The *GN1* policy is as follows: if there is a constructive move, then do it; else arbitrarily choose a misplaced block and move it to the floor. Here is the *GN1* policy as a decision tree.

$$\begin{aligned} policy_{GN1}\ x = {} & \mathit{if}\ \ extractAction \circ isConstructive\ x \\ & \mathit{then}\ 1 \\ & \mathit{else\ if}\ \wedge_3\ (isDeadlocked)\ (extractAction \circ projA \circ isMisplaced) \\ & \qquad\qquad (extractAction \circ projB \circ isFloor)\ x \\ & \quad \mathit{then}\ 1 \\ & \quad \mathit{else}\ \ 0. \end{aligned}$$

6.4.3 Experiments and Results

This subsection contains discussions of a few experiments carried out and the results obtained. We start with a description of the basic experimental setup.

In the description of the agent algorithm, the three functions $selectAction$, $updateQ$ and $updateP$ referred to in Figure 6.11 are left unspecified. We now state how they are instantiated in the experiments below.

The function $selectAction$ is used to control the trade-off between adequate exploration of the state space and sufficient exploitation of the (current) policy. An *exploration factor* N can be specified such that the agent selects a random action $N\%$ of the time, and follows the current policy the rest of the time.

The function $updateQ$ uses the on-line algorithm of ALKEMY (see §4.2.3.2) to retrain the current Q regression tree at the end of each episode or after 100 moves since the

last retrain, whichever occurs first.

The function $updateP$ behaves in a similar manner. It also has a few other mechanisms in place. In particular, it can choose to ignore training examples collected in the early phase of learning when the predictive power of the Q regression tree is still poor. Further, it can delay the retraining of the policy in two different ways: an initial delay and an accuracy-moderated delay. The first is used to delay retraining until sufficient data is available. The second is used to avoid unnecessary retraining when the current policy is still good. This works as follows. The empirical error of the current P decision tree on the current set of training examples is computed every time a retraining request is issued, and retraining is not performed unless a pre-specified error threshold is breached.

For Q-learning, the window size for the on-line learning algorithm was set at 200; for P-learning, the window size was set at 1000. These are reasonable numbers since there are about five times as many P training examples as Q training examples.

The predicate rewrite system for the Q hypothesis language used in the experiments is as given in §6.4.2. The predicate rewrite system for the P hypothesis language is as follows.

$$top \rightarrowtail \wedge_3\ top\ top\ top$$
$$top \rightarrowtail extractAction \circ projA \circ isMisplaced$$
$$top \rightarrowtail extractAction \circ projB \circ isFloor$$
$$top \rightarrowtail extractAction \circ isConstructive$$
$$top \rightarrowtail isDeadlocked$$

Graphs showing the results obtained from the three experiments described next are shown at the end of the section in Figures 6.12–6.14. For each experiment two graphs are plotted, the first depicting cumulative moves versus episodes, and the second depicting extra moves versus episodes. Extra moves are moves additional to what the policy US predicts for a given episode. We have chosen US as a benchmark because it is near optimal for a small number of blocks [183] and simple to compute. Note that a consequence of this is that sometimes the number of extra moves plotted in the second type of graph is a small negative number, when the agent solves an episode in fewer moves than US.

6.4.3.1 Experiments 1 and 2 — Initialization

These two experiments consider the initialization problem for the agent. In blocks world, the number of states rises rapidly as the number M of blocks increases. Even for small values of M, finding a goal state by randomly exploring the state space is impossible. Some sort of guidance for the agent is thus required. The symbolic nature of ALKEMY allows us to specify an initial policy for the agent.

Experiment 1 is conducted in a blocks world containing 5 blocks. At each episode the agent is supplied with a new task of achieving a set of 5 on-states. It receives a reward at the end of each episode after achieving the specified task.

In the first instance, the agent is supplied with no initial policy and the $selectAction$ function specifies a zero exploration factor. The effect of this is that the agent initially randomly explores the 5-block state space, fully exploiting its (initially empty) current policy that becomes refined through retraining over time.

In the second instance, the agent is supplied with the initial policy $policy_{simple}$. The $selectAction$ function again specifies a zero exploration factor. The $updateP$ function ignores the training data collected in the first 20 episodes, and delays the retraining of the initial policy decision tree until after episode 40. The retraining error threshold is set at 15%.

In both instances the agent eventually converges to a good policy $policy_{GN1}$. This occurs after 52 episodes without guidance and after 71 episodes with guidance. Fewer overall moves were made, however, with guidance.

Experiment 2 is conducted in a blocks world containing 8 blocks where the task is to achieve a set of 8 on-states. The number of states in an 8 blocks world is 394,353 compared to 501 for a 5 blocks world (see [183]). In both cases, there is only a single goal state. Unguided random exploration does not reach the goal state within a reasonable time for such a big state space, so guided exploration is mandatory. This experiment shows the agent converging to $policy_{GN1}$ after 57 episodes, starting from $policy_{simple}$.

The $selectAction$ function for this experiment specifies an exploration factor of 50% until after episode 20 when it drops (immediately) to zero. The $updateP$ function ignores the training data collected in the first 10 episodes, and delays the retraining of the initial policy until after episode 20. The retraining error threshold is set at 15%.

6.4.3.2 Experiment 3 — Adapting to a changing environment

This experiment shows the agent moving between different blocks worlds with minimal changes in policy. The experiment is conducted in a blocks world initially containing 5 blocks. At some episode, a new block is added to the environment and, at a subsequent episode, yet another block is added. The task of the agent is also incremented in the richer environments from sets of 5 on-states to sets of 6 on-states, and finally to sets of 7 on-states.

The agent is initialized with $policy_{simple}$ and converges to the better policy $policy_{GN1}$ after episode 21. At episode 30, a 6th block is added and the task incremented to 6 on-states. The agent continues to achieve its harder tasks by applying the policy it learned in the simpler world. At episode 37, an inconsistency between the incoming training examples and the current policy is detected to be higher than the error threshold and a retraining of the current working policy is triggered. This results in a series of suboptimal, but not disastrous, policies until episode 46 at which point they reconverge to $policy_{GN1}$.

At episode 60, a 7th block is added and the task complexity incremented to 7 on-states. The policy $policy_{GN1}$ is maintained until episode 128 when the incoming training data again exceeds the retraining error threshold, triggering a further retraining of the policy. This results in a suboptimal policy, which is refined over the course of

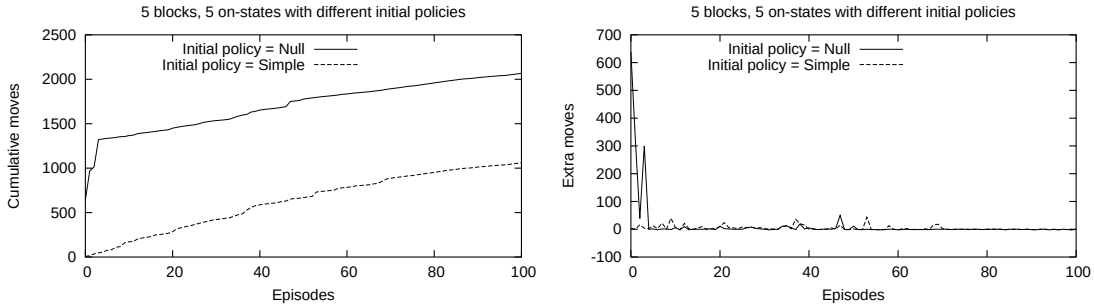


Figure 6.12: Experiment 1

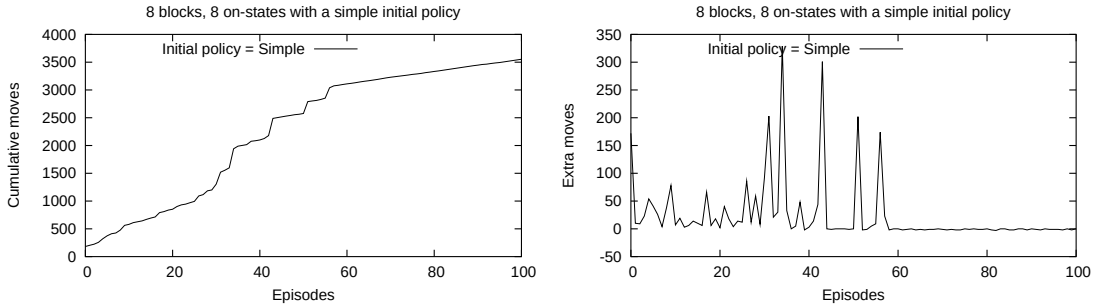


Figure 6.13: Experiment 2

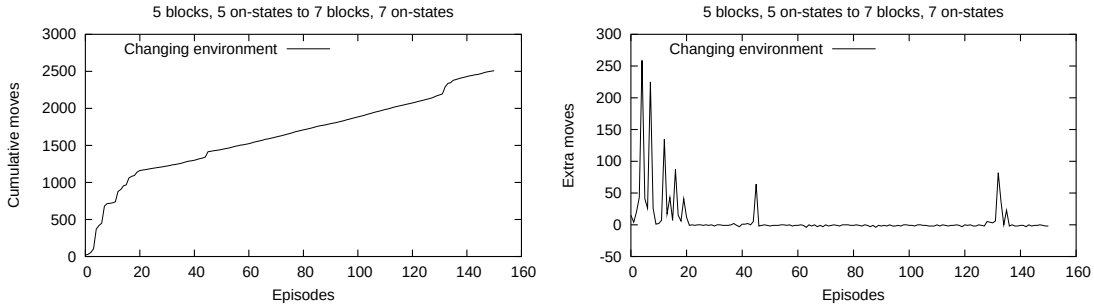


Figure 6.14: Experiment 3

the next few episodes, converging to $policy_{GN1}$ after episode 136.

The $selectAction$ function for this experiment specifies an exploration factor of 50% until after episode 20 when it drops to zero. The $updateP$ function ignores the training data collected in the first 10 episodes, and delays the retraining of the initial policy until after episode 20. The retraining error threshold is set at 15%.

6.4.4 Discussions

We now draw together some general remarks that can be made on the basis of the experiments.

The hypothesis language for the Q function is quite different to the hypothesis language for the policy function. The reason is that the Q function encodes the path length to the goal, while the policy distinguishes between good and bad actions – they are different functions that therefore require different hypothesis languages.

The Q function does not have to be perfect in order to get a good policy. This is partly because the policy is general, that is, is independent of the size of the state, so small errors in Q can be absorbed by the policy.

Domain knowledge needs to be incorporated to ease the usual search problems associated with reinforcement learning in large state spaces. The predicate rewrite system provides a convenient way for encoding this knowledge.

For large search spaces, a good initial policy is highly advantageous in that it can help the agent find rewards more easily (or, even, at all). The symbolic nature of ALKEMY helps here as it makes it possible for the agent designer to easily code up initial policies for this purpose. The idea of integrating guidance into relational reinforcement learning is not new, of course; see [65].

Discovering a good policy requires not only a good hypothesis language but also good training data. In other words, much attention has to be paid to issues such as adequate exploration of the state space and the size of training windows for the online learning algorithms. Too small a window can lose valuable data, while too large a window can restrict the agent's ability to react to a changing environment.

Finally, we remark that methods other than Q learning can be used to learn policies. For example, a model of the environment could be learned and then planning techniques employed. This idea is explored in [33] and [88].

6.5 Personalization — An Infotainment Agent

In this section, we study the applicability of ALKEMY to the task of building user agents that facilitate interaction between a user and the Internet. Specifically, we concentrate on the topic of personalization in which the agent adapts its behaviour according to the interests and preferences of the user.

The research is set in the context of an infotainment agent, which is a multi-agent system that contains a number of agents with functionalities for recommending TV programs, movies, music and the like, as well as information agents with functionalities for searching for information on the Internet. We concentrate on the TV recom-

mender as a typical such agent and show how a high degree of personalization can be achieved. The techniques can be ported comparatively easily to other agents of the system, thus providing a fully personalized infotainment system.

An outline of the section is as follows. In §6.5.1, we describe the infotainment agent and the TV recommender in more detail. §6.5.2 contains a short discussion of the learning algorithm used. We state the main function subject to learning and give a hypothesis language for it in §6.5.3. Some experimental results are presented in §6.5.4. We conclude with lessons learned and possible future work in §6.5.5.

The material presented here, first reported in [48], is joint-work with Joshua Cole, Matthew Gray and John W. Lloyd. Joshua Cole and Matthew Gray implemented the different user agents in the system.

6.5.1 An Infotainment Agent

The infotainment agent is a multi-agent system that combines a number of related functionalities concerning information search and entertainment. The agents that comprise the system include a TV recommender, a movie recommender, a music recommender, a news agent, a search agent, and a diary agent. In addition, there is a coordinator agent that has the responsibility of handling interactions between the user and the various agents in the system.

A detailed description of the architecture of the TV recommender is now given. The architecture of the other agents in the infotainment agent is similar. What functionality do we want the TV recommender to have? When the user first begins to use the TV recommender it clearly has no knowledge of the interests or preferences of the user. The aim is to design an adaptive architecture for the TV recommender so that within a comparatively short time, perhaps several weeks, it is able to make helpful recommendations to the user. Furthermore, it should improve its performance over longer periods and accurately track changing user interests and preferences. To get started, the agent presents a short questionnaire to the user the first time it is used. The purpose of the questionnaire is to acquire, with as little effort as possible on the part of the user, some initial idea of the user's interests and preferences. After that, the agent collects training examples by observing the user's activities. Over time, the agent is expected to be able to make recommendations for programs in specified time periods (say, 'next week' or 'tonight') that the user finds helpful.

A detailed description of the most pertinent aspects of the design of the TV recommender is now given. Three domain-specific types, $Channel$, $Genre$, and $Classification$, will be needed. Here are the data constructors for these types.

$$
\begin{aligned}
&ABC, Adventure\_1, Animal\_Planet, Arena,\\
&Biography, BBC\_World, Cartoon\_Network,\\
&\ldots\\
&Sky\_News, TCM, Tech\_TV, Travel,\\
&TV1, UK\_TV, W, World\_Movies : Channel
\end{aligned}
$$

$$Action, ActionAdventureGroup, Adult, Animals,$$
$$Animated, Art, ArtsMusicLiving, Auto,$$
$$\ldots$$
$$Volleyball, War, Watersports, Weather,$$
$$Western, WesternGroup, Wrestling : Genre$$
$$Y7, Y, G, MA, M14, M, NA : Classification.$$

There are 49 channels, 115 genres and 7 classifications.

We introduce the following type synonyms.

$$Date = Day \times Month \times Year$$
$$Time = Hour \times Minute$$
$$Title = String$$
$$Subtitle = String$$
$$Duration = Minute$$
$$Synopsis = String$$
$$Program = Title \times Subtitle \times Duration \times (List\ Genre) \times Classification \times Synopsis$$
$$Year = Nat$$
$$Month = Nat$$
$$Day = Nat$$
$$Hour = Nat$$
$$Minute = Nat$$
$$Text = List\ String.$$

The agent has access via the Internet to a TV guide (for the next week or so) for all channels. This database is represented by a function tv_guide having signature

$$tv\_guide : Date \times Time \times Channel \to Program.$$

Here the date, time and channel information uniquely identifies the program and the value of the function is (information about) the program itself. The TV guide consists of (thousands of) facts like the following one.

$$((tv\_guide\ ((20, 7, 2004), (20, 30), ABC)) =$$
$$(\text{“}The\ Bill\text{”}, \text{“”}, 50, [Drama], M,$$
$$\text{“}Sun\ Hill\ continues\ to\ work\ at\ breaking\ the\ people\ smuggling\ operation\text{”})).$$

This fact states that the program on 20 July 2004 at 8.30pm on channel ABC has title “The Bill”, no subtitle, a duration of 50 minutes, genre drama, a classification for mature audiences, and synopsis “Sun Hill continues to work at breaking the people

smuggling operation".

6.5.2 Adaptation

For adaptation, we use the decision-list learning algorithm presented in §3.2.3, with the parameter K set to ∞. As shown in §3.5.3, this algorithm behaves well when there is little noise in the training data, which is what we have in this application.

We keep a current set of training examples for the function we wish to learn. Since a user's interests and preferences may change over time, there are times when the current set of training examples becomes inconsistent, in the sense that the same individual have two or more distinct class labels. A common (indirect) solution to this problem is to keep a moving window, removing the oldest training examples as new ones are received to keep to a limit on the window size. In this application, we prefer the approach of returning the training set to consistency, even to the point of asking the user to resolve conflicts, if necessary. This way, a training example could stay in the training set for a very long time and would only drop out if it contradicted another training example that was somehow confidently known to be correct. The current implementation uses a simple algorithm to check for consistency.

In the experiments reported in §6.5.4, the decision lists induced are used to make predictions on new individuals in the usual manner, but we abstain from making a prediction if the individual reaches the default node, lacking confidence. This means that the coverage of the learner is usually not 100%.

6.5.3 Personalization of the TV Recommender

As for all learning tasks, the main issue is deciding on a suitable hypothesis language. This is discussed for the TV recommender now.

The key function that needs to be learned is the function $user\_likes\_tv\_program$ which takes a TV program as input and returns true if the agent considers the program to be worth recommending to the user; otherwise, it returns false. Thus the belief base of the TV agent contains the function $user\_likes\_tv\_program$ that has signature

$$user\_likes\_tv\_program : Program \rightarrow \Omega$$

and a definition that is a decision list of the form

$$\begin{array}{l} user\_likes\_tv\_program\ x = \\ \quad if\ (p_1\ x)\ then\ 1 \\ \quad else\ if\ (p_2\ x)\ then\ 0 \\ \qquad \vdots \\ \quad else\ if\ (p_n\ x)\ then\ 1 \\ \quad else\ \ 0, \end{array}$$

where $p_1, \ldots, p_n$ are predicates on programs.

We now discuss the hypothesis language used by ALKEMY that contains these predicates $p_1, \ldots, p_n$ and is used to learn the definition of $user\_likes\_tv\_program$. First, a collection of transformations suitable for use in this application is presented.

We begin with two transformations whose definitions come from user interests and preferences, and so provide a way of personalizing the hypothesis language for $user\_likes\_tv\_program$. One of these is the function $genre$. To define this, we introduce the type synonym

$$Preference = Int,$$

where it is understood that only numbers in $\{-2, -1, 0, 1, 2\}$ are to be used as constants of type $Preference$. Then

$$genre : Genre \rightarrow Preference$$

is the function that maps each genre into an integer in the range -2 to 2, depending on how strong a preference the user has for that particular genre. Here is a typical definition of $genre$.

$$\begin{array}{l} genre\ x = \\ \quad if\ ((= Animals)\ x)\ then\ 1 \\ \quad else\ if\ ((= Animated)\ x)\ then\ -2 \\ \qquad\qquad \vdots \\ \quad else\ if\ ((= Wrestling)\ x)\ then\ -2 \\ \quad else\ 0. \end{array}$$

This definition states that the user has a modest liking for animal programs, a strong dislike of animation programs, and so on. The information in this definition is obtained by an initial questionnaire completed by the user and by belief update, if the user later changes his/her preferences. Our experiments showed that the learner was able to make good use of the information given by the function $genre$.

Another transformation obtained from the user is

$$classification : Classification \rightarrow Preference$$

that gives information about the user's liking for programs having a certain classification. A typical definition of $classification$ could be as follows.

$$\begin{array}{l} classification\ x = \\ \quad if\ ((= Y7)\ x)\ then\ -2 \\ \quad else\ if\ ((= Y)\ x)\ then\ -2 \\ \quad else\ 0. \end{array}$$

The information in this definition is also obtained by an initial questionnaire.

The transformation

$$StringToText : String \rightarrow Text$$

takes a string as input and returns the list of words in the order that they occur in the string (discarding white space between words). Furthermore, words in the output text are stemmed. Thus

$$(StringToText\ \text{“}High\ \ Technology\text{”}) = [\text{“}high\text{”}, \text{“}technolog\text{”}].$$

The transformation

$$listExists_1 : (String \rightarrow \Omega) \rightarrow Text \rightarrow \Omega$$

is defined by

$$listExists_1\ p\ t = \exists x.((p\ x) \wedge (member\ x\ t)).$$

The predicate $(listExists_1\ p)$ checks whether some text (that is, a list of strings) contains a string that satisfies p.

The predicate rewrite system for the function $user\_likes\_tv\_program$ is given in Figure 6.15. The actual strings S used in rewrites of the form

$$top \rightarrowtail (= S)$$

are the titles and subtitles of all the programs in the (current) set of training examples. In rewrites of the form

$$top \rightarrowtail (listExists_1\ (=\ S)),$$

the strings S appearing are computed as follows. First, the set of all stemmed words appearing in titles, subtitles or synopses of programs in the (current) set of training examples that are not stop-words is formed. Then for each word in this set we compute the ratio of the number of positive training examples (plus one) in which it appears divided by the number of negative training examples (plus one) in which it appears. The set of words is decreasingly ordered by this ratio and the top 100 are used in the rewrites. The intuition is that these 100 words are good for discriminating between positive and negative examples. Note that the predicate rewrite system is constantly changing as new training examples arrive.

Here are typical predicates that appear in induced decision lists for the function $user\_likes\_tv\_program$.

$$projTitle \circ (=\ \text{“}English\ \ Premier\ \ League\text{”})$$
$$projSynopsis \circ StringToText \circ (listExists_1\ (=\ \text{“}wide\text{”}))$$
$$projClassification \circ (= M)$$

$$top \rightarrowtail projTitle \circ top$$
$$top \rightarrowtail projTitle \circ StringToText \circ top$$
$$top \rightarrowtail projSubtitle \circ top$$
$$top \rightarrowtail projSubtitle \circ StringToText \circ top$$
$$top \rightarrowtail projGenre \circ (listExists_1\ top)$$
$$top \rightarrowtail projGenre \circ (listExists_1\ genre \circ top)$$
$$top \rightarrowtail projClassification \circ top$$
$$top \rightarrowtail projClassification \circ classification \circ top$$
$$top \rightarrowtail projSynopsis \circ StringToText \circ top$$

$$top \rightarrowtail (=\ \text{“}The\ Bill\text{”})$$
$$top \rightarrowtail (=\ \text{“}South\ Park\text{”})$$
$$\ldots$$
$$top \rightarrowtail (=\ \text{“}Seinfeld\text{”})$$
$$top \rightarrowtail (=\ \text{“}The\ Cosby\ Show\text{”})$$

$$top \rightarrowtail (listExists_1\ (=\ \text{“}adventur\text{”}))$$
$$top \rightarrowtail (listExists_1\ (=\ \text{“}coverag\text{”}))$$
$$\ldots$$
$$top \rightarrowtail (listExists_1\ (=\ \text{“}technolog\text{”}))$$
$$top \rightarrowtail (listExists_1\ (=\ \text{“}war\text{”}))$$

$$top \rightarrowtail (=\ -2)$$
$$\ldots$$
$$top \rightarrowtail (=\ 2)$$

$$top \rightarrowtail (=\ Action)$$
$$\ldots$$
$$top \rightarrowtail (=\ Wrestling)$$

$$top \rightarrowtail (=\ Y7)$$
$$\ldots$$
$$top \rightarrowtail (=\ NA)$$

Figure 6.15: The predicate rewrite system for the TV recommender

$projGenre \circ (listExists_1\ (= Children))$
$projGenre \circ (listExists_1\ genre \circ (= -1))$
$projTitle \circ (=\ \text{“}Edge\ of\ the\ Universe\text{”})$
$projSynopsis \circ StringToText \circ (listExists_1\ (=\ \text{“}scientist\text{”}))$
$projGenre \circ (listExists_1\ (= Business))$

Such decision lists usually contain over a hundred decision nodes.

Users also have some level of direct control over the function $user\_likes\_tv\_program$ since it is possible to add user-defined rules to the learned definition of the function. For example, a user can add a rule such as:

> If the title is “Rugby Union” and the word “Australia” is in the synopsis, then true.

(This rule assumes the predicate rewrite system is enriched by also allowing conjunctions of conditions.) In general, the predicate in a rule can be any one that is obtainable from the predicate rewrite system. Since the user-defined rules are checked before the learned part of the definition, TV programs that satisfy these conditions are guaranteed to be classified in the way the user desires.

6.5.4 Experiments and Results

Here we present the results of a number of experiments measuring the performance of the TV recommender. The four authors of [48] used the system over a two-week period, training it to personalize to their individual viewing preferences. One person trained the system twice, in different ways. All experiments were carried out on a 10-channel subset of the full TV guide, chosen by each user.

6.5.4.1 Experiment 1 — Learning under favourable conditions

The first set of experiments was designed to test whether the TV recommender could personalize to different users when good training data was made available. Two users used the system in a rigid way, collecting training examples for two hours from the TV guide each day for two weeks. They provided feedback on every program in those two hours.

Charts (a) and (b) in Figure 6.16 are learning curves showing the performance of the system as more training examples were supplied. 10-fold cross-validation experiments were performed after every 10 examples up to 50, and every 25 examples thereafter. Cover, recall, precision and accuracy were calculated in the usual way. The last three values were calculated on the covered examples only. A Bezier curve of best fit was plotted on the resulting data points. The charts show a rapid rise in performance up to approximately 100 examples and a gradual improvement as further examples were added. The performance on all measures is near 90% after 500 examples, for both users.

6.5.4.2 Experiment 2 — Learning under real conditions

The second set of experiments tested personalization to different users under more realistic conditions. Three users used the system on two weeks of TV programming. They requested recommendations from any time slot and provided training examples to improve the correspondence between these recommendations and their viewing preferences. Users generally supplied corrective training examples for incorrect recommendations or programs for which the TV recommender indicated it was unsure. Reinforcing training examples could also be supplied.

Charts (d)–(f) in Figure 6.16 are learning curves plotted for each user as in the previous set of experiments. They show a steep rise in performance after early training and a more gradual improvement towards the end of the training period. The number of examples is generally smaller for each user than in the previous experiment. The final performance is generally lower than in the previous experiment, although still broadly improving at this point.

6.5.4.3 Experiment 3 — Comparison of learning algorithms

This set of experiments evaluates the decision-list algorithm against more sophisticated learning algorithms.

The table in Figure 6.16 records the final number of examples (Ex) collected for each user as well as their positive (Ex+) and negative (Ex−) breakdowns. Also shown are decision-list 10-fold cross-validation results for cover (DL Cov), recall (DL Rec), precision (DL Prec) and accuracy (DL Acc) on the final datasets.

For the purpose of comparison, the boosting algorithm AdaBoost [78] was used. AdaBoost is arguably the best off-the-shelf algorithm available, and its performance gives a good indication of the kind of accuracy attainable by other state-of-the-art learners.

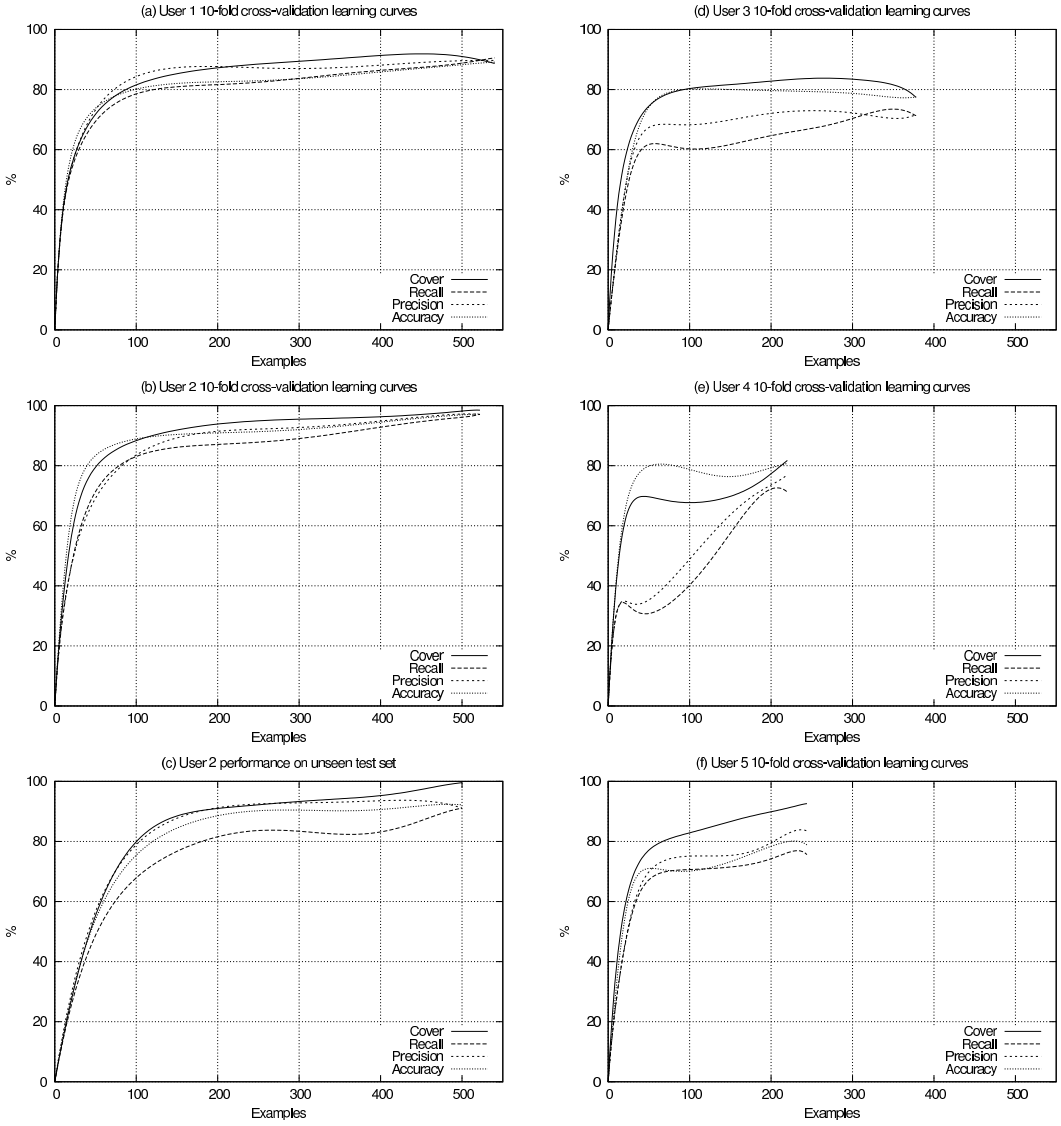
We used single predicates defined by the predicate rewrite system given in Figure 6.15 as base classifiers. The number of iterations was set at 400 after some experimentation.

10-fold cross validation results on the final datasets generated by each user are reported in the table in Figure 6.16 alongside the corresponding results achieved by the decision-list algorithm. In general terms, the numbers suggest that AdaBoost performs slightly better, but the decision-list algorithm is not far behind. (However this comparison is unfair to AdaBoost which makes a prediction on every example.)

This confirms that in this particular application, the use of a symbolic learning algorithm does not incur a significant cost in terms of accuracy.

6.5.4.4 Discussion

In all the experiments the performance of the system was evaluated using 10-fold cross validations. One could also use an independent test set to collect performance statistics. This was in fact done for each user and the results closely correspond. As an example we have included the test-set results for user 2 in Chart (c).



User	Ex	Ex+	Ex−	DL Cov	DL Rec	Boost Rec	DL Prec	Boost Prec	DL Acc	Boost Acc
1	540	283	257	88.67	90.57	80.24	89.69	91.16	89.13	90.12
2	520	232	288	98.46	97.15	92.67	97.02	94.64	97.27	94.23
3	377	146	231	77.43	71.37	54.98	71.44	79.09	77.40	74.79
4	220	68	152	81.75	71.21	63.48	77.05	81.97	80.79	84.55
5	248	126	122	92.60	75.63	82.50	83.52	84.63	78.85	83.03

Figure 6.16: Charts and table of results

The first set of experiments suggest that it is possible for the TV recommender to personalize well to different users, given sufficient training data. This is due in part to the nature of the problem. Television programming tends to be rather repetitive. Programs with the same title are scheduled cyclically at daily, weekly and sometimes hourly periods. In these circumstances instance-based learning is expected to perform well. This form of learning is captured implicitly by the use of predicates that test for equality of program titles in the decision lists. This works well in combination with more conventional rule-based learning that exploits more general conditions in the hypothesis language.

The second set of experiments suggest that real-world learning is more problematic. Collecting sufficient training data is difficult. Users in general prefer to give minimal feedback, mostly correcting mistakes as they occur. They also expect the TV recommender to make useful predictions on times of the day not previously trained on.

The results for user 3 show that even after a moderately large period of training, the TV recommender can fail to perform to expectation. This user chose two specialist movie channels from the TV guide and noted that the system performed poorly on movies compared with the more day-to-day TV programming of other channels. One reason for this is that there is in general an insufficient overlap between a person's preference for movies and TV programs for it to be possible to learn a theory that models both simultaneously. This suggests the need for a separate movie recommender with a more appropriate representation of movie individuals. We also note that the meta-data for movies in the TV guide was not very rich. For example, the genre for many movies was often simply designated as 'movie'.

Users 4 and 5 reported that subjectively the system performed well enough, and that further training to improve performance seemed unnecessary.

6.5.5 Conclusions and Future Work

In this section, we have described the application of ALKEMY to personalization of an infotainment agent, concentrating particularly on a TV recommender. The results suggest that a high level of personalization can be achieved.

One promising technique for improving the performance of the agent is that of active learning. The idea of this is that the agent should proactively seek training examples for the predictions it is most unsure about as measured by some confidence factor. In practice, this would mean that the TV recommender would occasionally directly ask the user about some particular program. This approach would relieve the user of some of the responsibility of giving good training examples to the agent.

6.6 Other Applications

ALKEMY has been used in various other applications. In [31], we applied the system to the benchmark problem of mutagenicity prediction [186] and obtained good

results. (The basic findings of that exercise are summarized in §7.2.2.1.) We have also incorporated ALKEMY into an intelligent email sorter [51] with some success.

Other researchers have also found ALKEMY useful. The system was used to learn classification rules for XML documents in [202]. It has also been used in the context of relational reinforcement learning in [33] and [88]. In [160], Alkemy was applied to the problem of constructing adaptive BDI agents.

One pound of learning requires ten pounds
of common sense to apply it.
Persian Proverb

Chapter 7

Comparison and Evaluation

Your true value depends entirely on
what you are compared with.
Bob Wells

This chapter presents a comparative study of ALKEMY in relation to closely related work in the literature, mainly in the field of Inductive Logic Programming (ILP). A qualitative analysis of the practical differences between learning in first-order logic and learning in our higher-order setting is presented in Sect. 7.1. A quantitative analysis of the performance of ALKEMY compared to other learning systems is then given in Sect. 7.2.

7.1 First-order vs Higher-order Learning: Practical Differences

The comparison is centred around the following three aspects of symbolic learning:

1. the underlying programming language used;
2. the way individuals are represented; and
3. the way hypothesis languages are constructed.

In each case, the approach taken in ALKEMY is compared with representative solutions employed in standard ILP systems. To draw out the essential differences, we will occasionally use as case studies applications discussed in Chapter 6.

7.1.1 Programming Language

From a user's perspective, the most obvious and fundamental difference between learning in first-order logic and learning in our higher-order setting is the need to work with two rather distinct declarative styles of programming. Just about every ILP system is built on Prolog. ALKEMY, in turn, is built on Escher [129].

The basic computational mechanisms lying at the heart of the two languages are different. The computational mechanism underlying Prolog is first-order resolution theorem proving [127], whereas that underlying Escher is equational reasoning [130, Chap. 5]. The basic programming construct in Prolog is a Horn clause, which is a

disjunction of literals that contains at most one positive literal. These are used to define facts, rules, and queries. The basic construct in Escher is a statement, which is a term of the form $h = b$ where h has the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function symbol f. Statements are used to define functions.

The logical foundations of the two languages are rather different, but from a practical programming perspective, Escher and Prolog have two important things in common. Like most other declarative languages, the concepts of pattern matching and function definition through basic recursion feature prominently in the two languages. The definition of many common functions encountered in programming problems can be written down in largely similar form in the two languages. To give an example, we show how quicksort is written in the two languages. Here is quicksort written in Prolog.

```
qsort( [],[] ).
qsort( [X | Tail], Sorted) :-
        split( X, Tail, Small, Big),
        qsort( Small, SortedSmall),
        qsort( Big, SortedBig),
        concat( SortedSmall, [X | SortedBig], Sorted).

split( _, [], [], []).
split( X,[Y | Tail], [Y | Small], Big) :-
        X > Y, !
        split( X, Tail, Small, Big).
split( X, [Y | Tail], Small, [Y | Big] ) :-
        split( X, Tail, Small, Big).

concat([],L,L).
concat( [X | L1], L2, [X | L3]) :- concat( L1, L2, L3).
```

Here is quicksort written in Escher.

```
qsort :: List a -> List a
qsort [] = []
qsort x:y = concat (qsort (filter (<= x) y),
                    x:(qsort (filter (> x) y)))

filter :: (a -> Bool) -> List a -> List a
filter p [] = []
filter p x:y = if (p x) then x:(filter p y) else (filter p y)

concat :: List a * List a -> List a
concat ([], x) = x
concat (u:x, y) = u:(concat (x, y))
```

As can be seen, the two definitions look very much alike.

In the context of learning, there are two important practical differences between Prolog and Escher, however. First of all, the basic entities we define and manipulate in Prolog are relations; in Escher, they are functions. The methods available for

combining basic entities to form complex concepts are therefore different in the two languages. To compose separate relations in Prolog, we use common variables as arguments to different predicates. This style of programming is available in Escher (see [130, §5.3]), but the main methods for combining functions in Escher are functional composition and the use of functions as arguments to (higher-order) functions. This latter style of programming is not available in Prolog, since the two operations are essentially higher-order concepts that cannot be defined in first-order logic. The two different styles of programming described account for all the differences in the two definitions of quicksort given above. As we shall see in §7.1.3, these differences have implications in the context of learning.

The second important difference between the languages is that Escher is strongly typed whereas Prolog is only (very) weakly typed. Strong typing is an essential part of almost all modern programming languages, declarative or not. As we will see below, the fact that Prolog is more-or-less untyped is an inconvenience that ILP systems have to deal with all the time.

7.1.2 Representation of Individuals

Two distinct approaches to representation of individuals are taken in first-order and higher-order learning. ALKEMY represents individuals by basic terms, adhering to the general principle in logic that (closed) terms denote individuals. In contrast, most ILP systems represent individuals using databases of facts. To illustrate the difference, we show how a molecule in the PTC dataset (see §6.3.1) is represented under the two schemes.

A typical ILP system will represent the molecule as follows.

```
atom('118','118_1').  element('118_1',c).
atom('118','118_2').  element('118_2',c).
atom('118','118_3').  element('118_3',i).
...
atom('118','118_19'). element('118_19',h).
atom('118','118_20'). element('118_20',h).
atom('118','118_21'). element('118_21',h).

bond('118','118_1_2'). bond_type('118_1_2',S).
connected('118_1','118_2','118_1_2').
connected('118_2','118_1','118_1_2').
...
bond('118','118_9_21'). bond_type('118_9_21',S).
connected('118_9','118_21','118_9_21').
connected('118_21','118_9','118_9_21').
```

The individual is named '118' and we have a database of facts about it. Each of its constituent atoms needs to be explicitly named so that information about the atom can be stored and a link to the constant '118' can be made. The same is true for each of its bonds. Notice how the predicates are used to mimic a type system, which is not available in Prolog.

The same molecule can be represented in ALKEMY in the following way.

```
Mol118 :: Graph Element Bond
Mol118 = ({ (1,c), (2,c), (3,i), (4,c), (5,c), (6,o), (7,c),
            (8,c), (9,o), (10,o), (11,h), (12,h), (13,h),
            (14,h), (15,h), (16,h), (17,h), (18,h), (19,h),
            (20,h), (21,h) },
          { ({ (4,1), (10,1) }, S), ({ (7,1), (10,1) }, S),
            ({ (1,1), (11,1) }, S), ({ (1,1), (12,1) }, S),
            ({ (1,1), (13,1) }, S), ({ (2,1), (14,1) }, S),
            ({ (4,1), (15,1) }, S), ({ (5,1), (16,1) }, S),
            ({ (5,1), (17,1) }, S), ({ (7,1), (18,1) }, S),
            ({ (8,1), (19,1) }, S), ({ (1,1), (2,1) }, S),
            ({ (8,1), (20,1) }, S), ({ (9,1), (21,1) }, S),
            ({ (2,1), (3,1) }, S), ({ (2,1), (4,1) }, S),
            ({ (4,1), (5,1) }, S), ({ (5,1), (6,1) }, S),
            ({ (6,1), (7,1) }, S), ({ (7,1), (8,1) }, S),
            ({ (8,1), (9,1) }, S) })
```

This representation is compact and all information about the individual is contained in one place. There is also no need to name every constituent part of the molecule. In the author's view, this is a more direct way of capturing the same information.

Aesthetically, the higher-order approach to representation of individuals is arguably more satisfactory. From a practical perspective, however, there is little to choose between the two. Whatever can be done with one can be done with the other. In fact, each one can always be recovered from the other, if necessary.

7.1.3 Construction of Hypothesis Languages

The predicate construction mechanism employed in ALKEMY is closely related to the body of work on refinement operators in ILP. Starting from the most general predicate *top*, ALKEMY conducts a general-to-specific search in a space of predicates defined by a predicate rewrite system. This is similar to top-down ILP systems that start from some initial theory and repeatedly apply (downward) refinement operators to specialize the current theory in its search for a good hypothesis. Applying a predicate rewrite in our setting thus corresponds to applying a refinement operator in top-down ILP systems.

We now compare the specifics of the two mechanisms, starting from the concept of generality employed in the two approaches. The downward refinement operators used in top-down ILP systems are built around a syntactic notion of generality called θ-subsumption. Given a clause c, such operators can perform the following two basic operations to construct (logically stronger) new clauses from c:

1. apply a substitution to the clause; or
2. add a literal to the body of the clause.

While this syntactic θ-subsumption-based notion of generality is convenient from a computational perspective, it is also narrow in scope and captures only a small part of semantic generality.

In contrast, the concept of generality employed in predicate rewrite systems is much more general and less well-defined. (This is a conscious design decision.) Given a predicate rewrite system $\rightarrowtail$, one can enrich it in arbitrary ways by adding another predicate rewrite $p \rightarrowtail q$ as long as one can show that

1. the type of p is more general than the type of q;
2. $\forall x.((p\ x) \leftarrow (q\ x))$; and
3. the predicate q is monotone with respect to $\rightarrowtail$.

The latter two conditions are not always easy to check, but this mechanism does allow one the flexibility of being able to use generality concepts that are much more powerful than simple syntactic notions like θ-subsumption.

Complex hypotheses are formed by composing simpler constructs. This is done differently in the two logical settings. We now compare them.

As mentioned earlier, the basic entities in ILP systems are relations and these are linked together by the use of common variables. Unfortunately, these variables are actually quite hard to control. There are three basic problems. First of all, they are untyped and the designers of many ILP systems saw the need to introduce ad hoc type systems to limit their scope. For example, TILDE has an (extra-logical) mechanism for specifying types; see [22, §6.4]. Many other systems also have similar facilities.

The second problem with the use of variables is that if one is not careful about their placements and contexts when adding new literals, lots of meaningless hypotheses can get generated. A few mechanisms like mode declarations, used in both TILDE and S-CART, have been invented in ILP to control the use of variables, but they are not very satisfactory solutions.

The third problem with variables is that their careless use can result in semantic difficulties. We give an example here. Variables are shared across decision nodes in TILDE. For that reason, a first-order tree induced by TILDE cannot actually be interpreted in the same way as a propositional decision tree; to understand what it means, a correspondence with a Prolog program need to be made; see [23, §5.3]. In practical terms, this means that the class of hypotheses considered by TILDE for a particular application may not actually correspond to what the user expects. Take for instance the Musk problem considered earlier in §6.2.4. TILDE trees of depth higher than one cannot precisely capture the strict multiple-instance concept intended for the problem. The concept can only be captured if the learning algorithm is restricted to learn stumps or lists. (See, for more details, [25].) This is the kind of subtlety that will surely be lost on a naïve user.

In the case of ALKEMY, predicates are formed by composing simpler functions called transformations. This method banishes the use of variables altogether from predicate rewrites and the predicates constructed – variables are only needed to define the transformations, after that they do not appear. One should note that this approach is not available to first-order ILP systems because higher-order functions are needed to make these work.

We now move on to consider the merits of predicate rewrite systems as a language bias compared to related approaches in ILP. Predicate rewrite systems can be thought of as a grammar for generating predicates. Similar grammatical constructs have been used in ILP for a long time. Examples of these include S-CART's schemata [115, Chap. 6], ICL's DLab templates [58] and Cohen's antecedent description grammars [42]. The last of these in fact bears close resemblance to predicate rewrite systems; see [31, §9] for a comparison. There is nothing to choose between these different ways of specifying grammars. They are all useful for the purpose they are intended to serve.

We end the discussion with some references. Refinement operator theories are covered extensively in [144, §5], [122, Chap. 3] and [156, Chap. 17]. Good discussions on declarative language bias can be found in [144, §7] and [115, Chap. 6].

Honest differences are often a healthy sign of progress.
Mahatma Gandhi

7.2 Quantitative Performance Comparisons

In this section, we give a quantitative analysis of ALKEMY's performance compared to related learning systems. §7.2.1 studies ALKEMY's performance on attribute value data; §7.2.2 examines its performance on structured data.

7.2.1 ALKEMY's Performance on Attribute Value Data

The first experiment is aimed at establishing a baseline for ALKEMY. ALKEMY is designed from the outset to deal with structured data, but one hopes that in the degenerate case of attribute value data, its performance is no worse than conventional propositional learners. In a sense, this experiment is just a sanity check.

We will concentrate on comparing the performance of C4.5 and ALKEMY here. We can focus on C4.5 because there is already a great body of work comparing C4.5 to other propositional learners. Once we understand the relative performance of ALKEMY and C4.5, we can tap into the literature to understand where ALKEMY sits with respect to other propositional learning algorithms.

7.2.1.1 Experimental Setup

As is usual in the field of machine learning, we will compare the two systems on a reasonable collection of datasets from the UCI repository. The datasets chosen for this particular study are described in Table 7.1. The first six datasets listed were employed in a similar study on FOIL reported in [39]. (The other datasets used in [39] are no longer available in the UCI repository.) The remaining three datasets were picked at random from the repository.

Domain	Instances	Discrete Attributes	Continuous Attributes
Monk-1	124	6	0
Monk-2	169	6	0
Monk-3	122	6	0
Promoters	106	57	0
Iris	100	0	4
Credit	690	9	6
Votes	435	16	0
Mushroom	8124	22	0
Audiology	200	69	0

Table 7.1: Domain descriptions

We next discuss the setup of learning parameters. C4.5 defines its own set of default tests given the description of instances in a dataset. To learn with ALKEMY, we need to first decide on a default hypothesis language for attribute value data. The following is used. For each discrete attribute A with n possible values, we construct n predicates of the form

$$proj_A \circ (= X)$$

where X takes on all possibles values of the attribute. Continuous attributes are handled in the usual fashion as in CART. For each such attribute (call it B), we first sort all the values that occur in the dataset for that attribute to obtain a list $[v_1, v_2, \ldots, v_m]$, removing duplicates as necessary. We then construct $m - 1$ predicates of the form

$$proj_B \circ (= j_i)$$

where j_i is defined to be $(v_i + v_{i+1})/2$. One should note that the hypothesis language for ALKEMY just described is a strict subset of the kind of predicates that would be considered by CART; see [32, p. 29].

For learning, the default options for C4.5 are used. Both C4.5 trees and C4.5 rules were induced for each dataset. ALKEMY is set up to learn decision trees only. Tree post-pruning is switched on with an appropriate percentage of training examples set aside for validation in each case.

For error estimation, leave-one-out cross validation is used whenever it is computationally feasible. Ten-fold cross validation is used otherwise.

7.2.1.2 Results

The results of the experiment are presented in Table 7.2. The error rate (in percentage) is shown in each entry.

We first compare the performance of Alkemic trees with C4.5 trees. As can be

Domain	C4.5		ALKEMY
	Trees	Rules	
Monk-1	24.3	0.0	8.8
Monk-2	35.0	34.7	11.57
Monk-3	2.8	3.7	2.8
Promoters	17.0	10.4	17.0
Iris	8.0	6.0	4.0
Credit	15.1	15.5	13.6
Votes	3.2	3.9	4.1
Mushroom	0.0	0.1	0.0
Audiology	22.1	22.5	26.5

Table 7.2: Results obtained by C4.5 and ALKEMY on some UCI datasets

seen, there is not a great deal of difference between the two in most cases, although ALKEMY seems to have a significant edge on Monk-1 and Monk-2.

We next examine the relative performance between Alkemic trees and C4.5 rules. In cases where the (heuristic) rules-generation algorithm of C4.5 actually works (this clearly happens in the case of Monk-1 and Promoters), C4.5 rules enjoys better performance compared to Alkemic trees. In all other cases, C4.5 rules performs at the same level as C4.5 trees and Alkemic trees. This suggests that ALKEMY can probably benefit from the incorporation of C4.5's rules-generation algorithm.

7.2.1.3 Discussion

As far as such experiments can tell us, we can conclude that the predictive performance of C4.5 and ALKEMY are mostly comparable. This shows that ALKEMY does not perform worse than propositional decision-tree learners when the data degenerate to simple feature vectors. In a way, this conclusion is not surprising since one can, with appropriate set up of learning parameters, make ALKEMY behave *exactly* like CART.

A related question is whether there is an inherent computational cost associated with the use of a rich language setting in ALKEMY. While C4.5 runs noticeably faster than ALKEMY in all cases, one should note that all the experiments ran in seconds, often less than a second. I don't believe there is an inherent computational cost associated with the use of a rich language setting in ALKEMY. The C4.5 system runs faster simply because it is a much more optimized learner.

7.2.2 ALKEMY's Performance on Structured Data

In this section, we compare ALKEMY against some standard ILP systems on structured data. The comparative results are drawn from our experience with two benchmark datasets: Mutagenesis and Musk.

Material presented in this section first appeared in [31]. The Mutagenesis experiments reported there were all performed by the author. The Musk experiments reported were jointly performed by Xiaobing Wu and the author.

7.2.2.1 Mutagenesis

The Mutagenesis dataset has been studied extensively in the field of ILP. The basic problem involves learning a theory to predict whether a chemical molecule is mutagenic or not. The problem has been described many times in the literature; the reader is referred to [186] and [107] for details on this important and interesting problem.

To solve the problem using ALKEMY, we first need to decide on a representation scheme for molecules. Each molecule in the dataset is described by four attributes, its physical structure and three chemical descriptors called I_1, I_a and ϵ_{LUMO}. The meaning of the latter three are explained in [107].

An (undirected) graph is used to model the structure of a molecule. The type $Element$ is the type of the (relevant) chemical elements.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

Here are the type declarations.

$$I_1 = \Omega$$
$$I_a = \Omega$$
$$\epsilon_{LUMO} = Float$$
$$AtomType = Nat$$
$$Charge = Float$$
$$Atom = Element \times AtomType \times Charge$$
$$Bond = Nat$$
$$Structure = Graph\ Atom\ Bond$$
$$Molecule = I_1 \times I_a \times \epsilon_{LUMO} \times Structure.$$

The set of 188 regression-friendly molecules is used in our experiment. We want to learn the definition of the function $mutagenic$ having the following signature.

$$mutagenic : Molecule \to \Omega.$$

The hypothesis language contains the following transformations.

$$(= 1) : I_1 \to \Omega$$
$$(= 0) : I_1 \to \Omega$$
$$(= 1) : I_a \to \Omega$$
$$(= 0) : I_a \to \Omega$$
$$(\leq -3.718) : \epsilon_{LUMO} \to \Omega$$

$(\leq -3.368) : \epsilon_{LUMO} \rightarrow \Omega$
$(\leq -3.168) : \epsilon_{LUMO} \rightarrow \Omega$
$(\leq -3.018) : \epsilon_{LUMO} \rightarrow \Omega$
$(\leq -2.668) : \epsilon_{LUMO} \rightarrow \Omega$
$(\leq -2.418) : \epsilon_{LUMO} \rightarrow \Omega$
$(\leq -0.718) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -3.418) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -3.218) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -3.068) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -2.918) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -2.618) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -2.368) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -0.668) : \epsilon_{LUMO} \rightarrow \Omega$
$(= Br) : Element \rightarrow \Omega$
$\vdots$
$(= S) : Element \rightarrow \Omega$
$(= 1) : AtomType \rightarrow \Omega$
$(= 3) : AtomType \rightarrow \Omega$
$\vdots$
$(= 232) : AtomType \rightarrow \Omega$
$(\geq -0.781) : Charge \rightarrow \Omega$
$(\geq -0.424) : Charge \rightarrow \Omega$
$(\geq -0.067) : Charge \rightarrow \Omega$
$(\geq 0.290) : Charge \rightarrow \Omega$
$(\geq 0.647) : Charge \rightarrow \Omega$
$(\geq -0.424) : Charge \rightarrow \Omega$
$(\geq -0.067) : Charge \rightarrow \Omega$
$(\geq 0.290) : Charge \rightarrow \Omega$
$(\geq 0.647) : Charge \rightarrow \Omega$
$(\geq 1.004) : Charge \rightarrow \Omega$
$(= 1) : Bond \rightarrow \Omega$
$(= 2) : Bond \rightarrow \Omega$
$(= 3) : Bond \rightarrow \Omega$
$(= 4) : Bond \rightarrow \Omega$
$(= 5) : Bond \rightarrow \Omega$
$(= 7) : Bond \rightarrow \Omega$

$$(> 0) : Nat \to \Omega$$
$$(> 1) : Nat \to \Omega$$
$$(> 2) : Nat \to \Omega$$
$$(> 3) : Nat \to \Omega$$
$$(> 4) : Nat \to \Omega$$
$$projI_1 : Molecule \to I_1$$
$$projI_a : Molecule \to I_a$$
$$proj\epsilon_{LUMO} : Molecule \to \epsilon_{LUMO}$$
$$projStructure : Molecule \to Structure$$
$$projElement : Atom \to Element$$
$$projAtomType : Atom \to AtomType$$
$$projCharge : Atom \to Charge$$
$$vertices : Structure \to \{Vertex\ Atom\ Bond\}$$
$$edges : Structure \to \{Edge\ Atom\ Bond\}$$
$$vertex : Vertex\ Atom\ Bond \to Atom$$
$$edge : Edge\ Atom\ Bond \to Bond$$
$$connects : Edge\ Atom\ Bond \to (Vertex\ Atom\ Bond \to Nat)$$
$$domCard : (Vertex\ Atom\ Bond \to \Omega) \to \{Vertex\ Atom\ Bond\} \to Nat$$
$$domCard : (Edge\ Atom\ Bond \to \Omega) \to \{Edge\ Atom\ Bond\} \to Nat$$
$$msetExists_2 : (Vertex\ Atom\ Bond \to \Omega) \to (Vertex\ Atom\ Bond \to \Omega) \to (Vertex\ Atom\ Bond \to Nat) \to \Omega$$
$$\wedge_2 : (Charge \to \Omega) \to (Charge \to \Omega) \to Charge \to \Omega$$
$$\wedge_2 : (Atom \to \Omega) \to (Atom \to \Omega) \to Atom \to \Omega$$
$$\wedge_3 : (Atom \to \Omega) \to (Atom \to \Omega) \to (Atom \to \Omega) \to Atom \to \Omega$$
$$\wedge_2 : (Molecule \to \Omega) \to (Molecule \to \Omega) \to Molecule \to \Omega$$
$$\wedge_3 : (Molecule \to \Omega) \to (Molecule \to \Omega) \to (Molecule \to \Omega) \to Molecule \to \Omega$$
$$\wedge_4 : (Molecule \to \Omega) \to (Molecule \to \Omega) \to (Molecule \to \Omega) \to (Molecule \to \Omega) \to Molecule \to \Omega.$$

For the hypothesis language, we experimented with a few ideas and eventually settled on the following predicate rewrite system, which examines, in addition to the three chemical descriptors, the number of occurrences of certain kinds of atoms in the molecule. One can look at more complicated hypothesis languages, but this is the simplest one that achieves good result on the problem.

$$top \rightarrowtail \wedge_4\ (projI_1 \circ top)\ (projI_a \circ top)\ (proj\epsilon_{LUMO} \circ top)\ (projStructure \circ top)$$
$$top \rightarrowtail vertices \circ (domCard\ (vertex \circ top)) \circ (> 0)$$

$$
\begin{aligned}
&top \rightarrowtail projElement \circ top\\
&top \rightarrowtail projAtomType \circ top\\
&top \rightarrowtail projCharge \circ top\\
&top \rightarrowtail \wedge_2\ (projAtomType \circ top)\ (projCharge \circ top)\\
&top \rightarrowtail (= 1)\\
&top \rightarrowtail (= 0)\\
&top \rightarrowtail (\leq -0.718)\\
&(\leq -0.718) \rightarrowtail (\leq -2.418)\\
&\qquad\vdots\\
&(\leq -3.368) \rightarrowtail (\leq -3.718)\\
&top \rightarrowtail (\geq -3.418)\\
&(\geq -3.418) \rightarrowtail (\geq -3.218)\\
&\qquad\vdots\\
&(\geq -2.368) \rightarrowtail (\geq -0.668)\\
&top \rightarrowtail (= Br)\\
&\qquad\vdots\\
&top \rightarrowtail (= S)\\
&top \rightarrowtail (= 1)\\
&top \rightarrowtail (= 3)\\
&\qquad\vdots\\
&top \rightarrowtail (= 232)\\
&top \rightarrowtail (\geq -0.781)\\
&(\geq -0.781) \rightarrowtail (\geq -0.424)\\
&\qquad\vdots\\
&(\geq 0.647) \rightarrowtail (\geq 1.004)\\
&top \rightarrowtail (= 1)\\
&\qquad\vdots\\
&top \rightarrowtail (= 7)\\
&(> 0) \rightarrowtail (> 1)\\
&(> 1) \rightarrowtail (> 2)\\
&(> 2) \rightarrowtail (> 3)\\
&(> 3) \rightarrowtail (> 4).
\end{aligned}
$$

Using the predicate rewrite system given, the learner searched for a definition for *mutagenic* with the *prune* parameter set to 100%. On a 10-fold cross-validation,

ALKEMY achieved an average accuracy of 89.39%. The whole exercise took only a few minutes on a 3 GHz Intel machine. The following simple definition for $mutagenic$ was found. (Note that to improve readability, we have taken the liberty to remove predicates of the form $proj_A \circ top$, where A is some attribute, from the definition actually induced by ALKEMY.)

$$
\begin{array}{l}
mutagenic\ m = \\
\quad if\ \ \wedge_2\ \ (projI_1 \circ (= \bot))\ (proj\epsilon_{LUMO} \circ (\geq -2.368))\ m \\
\quad then\ if\ \ projStructure \circ vertices \circ \\
\qquad\qquad (domCard\ \ (vertex \circ projAtomType \circ (= 38))) \circ (> 1)\ m \\
\qquad then\ if\ \ projStructure \circ vertices \circ (domCard\ \ (vertex \circ (\wedge_2 \\
\qquad\qquad projAtomType \circ (= 22)\ \ projCharge \circ (\geq -0 \circ 067)))) \circ (> 0)\ m \\
\qquad\quad then\ 0 \\
\qquad\quad else\ 1 \\
\qquad else\ 0 \\
\quad else\ 1.
\end{array}
$$

"A molecule is mutagenic iff either

(i) I_1 is true, or

(ii) $\epsilon_{LUMO} < -2.368$, or

(iii) I_1 is false, $\epsilon_{LUMO} \geq -2.368$, it has at least two atoms of type 38, and it does not have an atom of type 22 and charge ≥ -0.067."

Table 7.3 lists the best accuracies achieved by different ILP systems on the Mutagenesis dataset, obtained without constraining the hypothesis language in any way. The accuracy reported for STILL was estimated by running the system on multiple random 90%-10% partitions of the data into training and test sets. All the other accuracies were estimated using 10-fold cross validations.

System	% correct
Progol	88.0
FOIL	86.7
STILL	93.6
TILDE	86.0
ALKEMY	89.4

Table 7.3: Accuracies obtained by different learners on the Mutagenesis dataset

As can be seen, in terms of predictive accuracy, ALKEMY compares well against other ILP systems on this particular domain. The mutagenic theory induced is also attractively simple.

7.2.2.2 Musk

We next look at the Musk problem. This problem has earlier been introduced in Sect. 6.2. We use the same representation scheme described there, repeated here for convenience.

$$-6, -5, \ldots, 5, 6 : \mathit{Distance}$$
$$\mathit{Conformation} = \mathit{Distance} \times \cdots \times \mathit{Distance}$$
$$\mathit{Molecule} = \{\mathit{Conformation}\}.$$

Molecules with a musk odour are labelled true, and those without are labelled false. The function musk we want to learn thus has signature $\mathit{musk} : \mathit{Molecule} \to \Omega$. The hypothesis language contains the following transformations.

$$(= -6) : \mathit{Distance} \to \Omega$$
$$\vdots$$
$$(= 6) : \mathit{Distance} \to \Omega$$
$$(\neq -6) : \mathit{Distance} \to \Omega$$
$$\vdots$$
$$(\neq 6) : \mathit{Distance} \to \Omega$$
$$\mathit{proj}_1 : \mathit{Conformation} \to \mathit{Distance}$$
$$\vdots$$
$$\mathit{proj}_{166} : \mathit{Conformation} \to \mathit{Distance}$$
$$\mathit{setExists}_1 : (\mathit{Conformation} \to \Omega) \to \mathit{Molecule} \to \Omega$$
$$\wedge_2 : (\mathit{Conformation} \to \Omega) \to (\mathit{Conformation} \to \Omega) \to \mathit{Conformation} \to \Omega$$
$$\vdots$$
$$\wedge_9 : (\mathit{Conformation} \to \Omega) \to \cdots \to (\mathit{Conformation} \to \Omega) \to \mathit{Conformation} \to \Omega.$$

We worked with predicate rewrite systems having the following general form.

$$\mathit{top} \rightarrowtail \mathit{setExist}_1\ (\wedge_k\ \mathit{top} \ldots \mathit{top})$$
$$\mathit{top} \rightarrowtail \mathit{proj}_1 \circ (= -6)$$
$$\mathit{top} \rightarrowtail \mathit{proj}_1 \circ (= -5)$$
$$\vdots$$
$$\mathit{top} \rightarrowtail \mathit{proj}_{166} \circ (\neq 5)$$
$$\mathit{top} \rightarrowtail \mathit{proj}_{166} \circ (\neq 6)$$

The main parameter of interest here is k, the number of conjuncts in the predicate in the argument to $\mathit{setExists}_1$. We experimented with different values of k in the range 2

Algorithm	% correct
EM-DD	96.0
Iterated-discrim APR	89.2
GFS elim-kde APR	80.4
C4.5	58.8
Backpropagation net	67.7
TILDE	79.4
RIPPER-MI	77.0
Relic	87.3
ALKEMY	83.5

Table 7.4: Accuracies obtained by different learners on the Musk2 dataset

to 9 and eventually settled on $k = 3$.

Using the predicate rewrite system stated, experiments were carried out on the Musk2 dataset. This dataset contains 102 examples, of which 39 are musk and 63 are not. The 102 molecules have a total of 6598 conformations. A suitably parallelized version of ALKEMY (see [31]) was set up with the $prune$ parameter set to 100% and the $cutout$ parameter set to 25000. To learn multiple-instance concepts, the decision-stump algorithm was used.

Over three 10-fold cross validations, ALKEMY achieved an average accuracy of 83.5%. The following definition for the function $musk$ was induced.

$$
\begin{aligned}
&musk\ m = \\
&\quad if\ setExists_1\ (\wedge_3\ (proj_{29} \circ (\neq -4))\ (proj_{132} \circ (= -2))\ (proj_{119} \circ (= 1)))\ m \\
&\quad then\ 1 \\
&\quad else\ 0.
\end{aligned}
$$

Table 7.4 lists the accuracies obtained by other systems on the Musk2 dataset. Overall, the performance of ALKEMY is competitive. Naturally, it outperforms propositional learners like C4.5 and neural networks, which clearly cannot cope well with multiple-instance data. While unable to match specially designed multiple-instance learning algorithms like EM-DD and Iterated-discrim APR, ALKEMY compares well with other general purpose ILP systems like TILDE, RIPPER-MI and Relic. This experiment again confirms that ALKEMY is competitive with respect to existing ILP systems.

To compare is not to improve.
John French

Chapter 8

Conclusion

We conclude with a discussion of the research contributions and some future work.

8.1 Thesis Contributions

The contribution of this thesis is, above all else, in providing some answers to the two questions stated in Section 1.3.

On the nature of learning with expressive languages

The whole of Chapter 3 was devoted to a treatment of this subject.

On the question of approximation, Section 3.3 brings out the interesting interplay between boolean connectives introduced explicitly through the predicate rewrite system and those introduced implicitly through the choice of the tree-learning algorithm, and from that, clarifies the relationships between natural Alkemic function classes.

On the question of estimation, Section 3.4 discusses a few error bounds suitable for use with ALKEMY and presents techniques for calculating the VC dimensions of predicate classes defined on different data types. The analysis highlights the danger of overfitting brought about by the use of rich expressive languages for representing individuals and hypothesis spaces, and partly explains some of the phenomena observed in practice. The VC dimension results may have wider applications beyond ALKEMY.

On the question of computation, Section 3.5 spells out the inherent complexity of the different optimization problems associated with learning and describes various techniques – the tricks of the trade – for easing the computational burden associated with the use of massive predicate search spaces.

Building upon the above, Section 3.6 presents results on the efficient learnability of Alkemic function classes in the PAC and agnostic PAC models. Learnability issues in practical applications are also discussed.

An understanding of these theoretical questions provides guidance in the crafting of hypothesis spaces and selection of algorithms in practical applications of the system.

On relevance and applicability

This issue was dealt with in Chapters 4, 5, and 6.

We need to develop different tools before we can demonstrate the relevance of the general approach to learning propounded in this thesis. The extensions of ALKEMY to regression and incremental learning are two contributions of this kind. They widen the applicability of the learning system.

In terms of relevance, our work on

1. Musk led to interesting lessons on practical aspects of learning with a rich and tunable hypothesis language.
2. Blocks World resulted in a better understanding of the nature of relational reinforcement learning.
3. the Predictive Toxicology Challenge led to the discovery of real (albeit mostly known) biochemical knowledge about cancer-causing structures in molecules.
4. the TV recommender agent resulted in the design and construction of a deployable and potentially commercializable system that makes strong use of the symbolic facilities provided by the learner.

These applications demonstrate the potential advantages that can be offered by a symbolic learning system. As case studies, they provide information about *when* symbolic learning can profitably be applied, and *how* that can be done.

8.2 Future Work

We finish with some remarks on future research.

The connection between decision lists and linear models in classification is one area that should be investigated in full because it has the potential to shed light on the relationship between symbolic and non-symbolic learning. In particular, the connection may help explain why list-based symbolic learning systems tend not to perform as well as algorithms like (kernelized) support vector machines [179].

Boosting algorithms have been shown to work well with decision trees [163]. It would be interesting to try and understand how such learning techniques can be used to improve the performance of ALKEMY. Generalization bounds for boosted Alkemic decision trees can be obtained by putting together results from [138], [178] and §3.4.3. Convergence properties of algorithms like AdaBoost is also better understood now; see [171]. We can gain a lot quite quickly by building on these results.

Boosting is also interesting from another perspective. It is a class of algorithms that uses as a parameter the base function class. The weak hypothesis assumption has not been studied extensively. There is work on establishing the existence of weak

learners in geometric classifiers (see §3.1 of [139]), but we are not aware of any similar work in symbolic learning. This is a topic worth studying.

The on-line learning algorithms can do with more development, possibly with input from work on incremental theory revision.

Research into tight error bounds that can actually be used for model selection should be conducted. Most existing bounds are informative but quantitatively useless. They tell us what are important parameters that control generalization but can't actually be used to obtain non-trivial bounds. It is important to move beyond that.

Preliminary results in [45] suggest that learning with rich expressive languages, especially one involving the use of sets, can be profitably analysed in Blum's infinite attribute space model [26]. This line of investigation is worth pursuing.

Finally, from an intelligent agent's perspective, one of the chief advantages of symbolic learning is the potential it offers to integrate learning and reasoning on a common platform. Some interesting ideas on how this can be done have been developed; see [131] and [47]. Bringing these ideas to fruition is our most important future work.

Bibliography

1. Christophe Ambroise and Geoffrey J. McLachlan. Selection bias in gene extraction on the basis of microarray gene-expression data. *Proceedings of the National Academy of Sciences*, 99(10):6562–6566, 2002.

2. Martin Anthony. Decision lists and threshold decision lists. Technical Report LSE-CDAM-2002-11, Department of Mathematics and Centre for Discrete and Applicable Mathematics, London School of Economics, 2002.

3. Martin Anthony and Peter L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

4. Annalisa Appice, Michelangelo Ceci, and Donato Malerba. Mining model trees: A multi-relational approach. In *Proceedings of the International Conference on Inductive Logic Programming*, number 2835 in LNAI, pages 4–21. Springer-Verlag, 2003.

5. Joseph C. Arcos, Mary F. Argus, George Wolf, Yin-Tak Woo, and David Y. Lai. *Chemical Induction of Cancer, Volumes I, IIA, IIB, IIIA*. Academic Press, 1968-82.

6. Marta Arias and Roni Khardon. Learning closed horn expressions. *Information and Computation*, 178:214–240, 2002.

7. Marta Arias and Roni Khardon. Complexity parameters of first order classes. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, pages 22–37, 2003.

8. Hiroki Arimura. Learning acyclic first-order horn sentences from entailment. In *Proceedings of the International Conference on Algorithmic Learning Theory*. Springer-Verlag, 1997.

9. Peter Auer. On learning from multi-instance examples: Empirical evaluation of a theoretical approach. In *Proceedings of the 14th International Conference on Machine Learning*, pages 21–29. Morgan Kaufmann, 1997.

10. Peter Auer, Robert C. Holte, and Wolfgang Maass. Theory and applications of agnostic PAC-learning with small decision trees. In *Proceedings of the 12th International Conference on Machine Learning*. Morgan Kaufmann, 1995.

11. Andrew R. Barron. Approximation and estimation bounds for artificial neural networks. *Machine Learning*, 14(1):115–133, 1994.

12. Peter L. Bartlett. The sample complexity of pattern classification with neural networks: The size of the weights is more important than the size of the network. *IEEE Transactions on Information Theory*, 44(2):525–536, 1998.

13. Peter L. Bartlett, Stéphane Boucheron, and Gábor Lugosi. Model selection and error estimation. *Machine Learning*, 48:85–113, 2002.

14. Peter L. Bartlett and Shahar Mendelson. Rademacher and Gaussian complexities: risk bounds and structural results. *Journal of Machine Learning Research*, 3:463–482, 2002.

15. Romualdo Benigni and Alessandro Giuliani. Putting the predictive toxicology challenge into perspective: reflections on the results. *Bioinformatics*, 19:1194–1200, 2003.

16. Jon Bentley. Aha! Algorithms. In *Programming Pearls*, pages 11–20. Addison-Wesley, second edition, 2000.

17. Neil C. Berkman and Tuomas W. Sandholm. What should be minimized in a decision tree: A re-examination. Technical Report 95-20, Computer Science Deparment, University of Massachusetts at Amherst, 1995.

18. Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-dynamic Programming*. Athena Scientific, 1996.

19. Alina Beygelzimer, Varsha Dani, Thomas Hayes, John Langford, and Bianca Zadrozny. Reductions between classification tasks. *Electronic Colloquium on Computational Complexity*, TR04-077, 2004.

20. Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

21. V.G. Blinova, D.A. Dobrynin, V.K. Finn, S.O. Kuznetsov, and E.S. Pankratova. Toxicology analysis by means of the JSM-method. *Bioinformatics*, 19:1194–1200, 2003.

22. Hendrik Blockeel. *Tilde: Top-down Induction of Logical Decision Trees: User's Manual*, 1997.

23. Hendrik Blockeel. *Top-Down Induction of First Order Logical Decision Trees*. PhD thesis, Departement Computerwetenschappen, Katholieke Universiteit Leuven, 1998.

24. Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

25. Hendrik Blockeel, David Page, and Ashwin Srinivasan. Multi-instance tree learning. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 57–64. ACM Press, 2005.

26. Avrim Blum. Learning boolean functions in an infinite attribute space. *Machine Learning*, 9(4):373–386, 1992.

27. Avrim Blum. Rank-r decision-trees are a subclass of r-decision-lists. *Information Processing Letters*, 42 (4):183–185, 1992.

28. Avrim Blum. On-line algorithms in machine learning. In Fiat and Woeginger, editors, *Online Algorithms: The State of the Art*, number 1442 in LNCS, chapter 14. 1998.

29. Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, 1987.

30. Antony F. Bowers, Christophe Giraud-Carrier, and John W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Proceedings of the 17th International Conference on Machine Learning*, pages 81–88. Morgan Kaufmann, 2000.

31. Antony F. Bowers, Christophe Giraud-Carrier, and John W. Lloyd. A knowledge representation framework for inductive learning. Available at `http://rsise.anu.edu.au/~jwl/`, 2001.

32. Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and Regression Trees*. Chapman & Hall, New York, 1984.

33. Robert Bridle and Eric McCreath. Improving the learning rate by inducing a transition model. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 1330–1331, 2004.

34. Nader H. Bshouty and Lynn Burroughs. On the proper learning of axis-parallel concepts. *Journal of Machine Learning Research*, 4:157–176, 2003.

35. Wray Buntine and Tim Niblett. A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 8:75–85, 1992.

36. Wray L. Buntine. *A Theory of Learning Classification Rules*. PhD thesis, School of Computing Science, University of Technology, Sydney, 1992.

37. Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice-Hall, 1999.

38. Mary Elaine Califf and Raymond J. Mooney. Advantages of decision lists and implicit negatives in inductive logic programming. *New Generation Computing*, 16 (3):263–281, 1998.

39. R. Mike Cameron-Jones and John Ross Quinlan. First order learning, zeroth order data. In *Proceedings of the 6th Australian Joint Conference on Artificial Intelligence*. World Scientific, 1993.

40. David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 726–731, 1991.

41. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

42. William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2):303–366, 1994.

43. William W. Cohen. PAC-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research*, 2:501–539, 1995.

44. William W. Cohen. PAC-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, 2:541–573, 1995.

45. William W. Cohen. Learning trees and rules with set-valued features. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 709–716, Menlo Park, CA, 1996. AAAI Press.

46. William W. Cohen and C. David Page Jr. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Computing*, 13:369–409, 1995.

47. Joshua Cole, John W. Lloyd, and Kee Siong Ng. Belief acquisition for agents. In preparation, 2004.

48. Joshua J. Cole, Matthew Gray, John W. Lloyd, and Kee Siong Ng. Personalisation for user agents. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 603–610, 2005.

49. Joshua J. Cole, John W. Lloyd, and Kee Siong Ng. Symbolic learning for adaptive agents. In *Annual Partner Conference, Smart Internet Technology Cooperative Research Centre*, pages 139–148, 2003. Available at `http://rsise.anu.edu.au/~jwl/`.

50. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

51. Elisabeth Crawford, Judy Kay, and Eric McCreath. IEMS - The Intelligent Email Sorter. In *Proceedings of the 19th International Conference on Machine Learning*, pages 83–90, 2002.

52. Stuart L. Crawford. Extensions to the CART algorithm. *International Journal of Man-Machine Studies*, 31:197–217, 1989.

53. Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, Cambridge, 2000.

54. Gautam Das and Michael T. Goodrich. On the complexity of optimization problems for 3-dimensional convex polyhedra and decision trees. *Computational Geometry*, 8:123–137, 1997.

55. Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, 1994.

56. Luc De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, 1992.

57. Luc De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.

58. Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2–3):99–146, 1997.

59. Luc De Raedt and Sašo Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

60. Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2nd edition, 2000.

61. Thomas Dietterich, Michael Kearns, and Yishay Mansour. Applying the weak learning framework to understand and improve C4.5. In *Proceedings of the 13th International Conference on Machine Learning*. Morgan Kaufmann, 1996.

62. Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

63. Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 1997.

64. Kurt Driessens. *Relational Reinforcement Learning*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 2004.

65. Kurt Driessens and Sašo Džeroski. Integrating guidance into relational reinforcement learning. *Machine Learning*, 57(3):271–304, 2004.

66. Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *Proceedings of the European Conference on Machine Learning*, number 2167 in LNAI, pages 97–109. 2001.

67. Harris Drucker and Corinna Cortes. Boosting decision trees. In *Advances in Neural Information Processing Systems 8*, pages 479–485, 1996.

68. Sašo Džeroski, Luc De Raedt, and Hendrik Blockeel. Relational reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning*, pages 136–143. Morgan Kaufmann, 1998.

69. Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.

70. Sašo Džeroski, Stephen Muggleton, and Stuart Russell. PAC-learnability of determinate logic programs. In *Proceedings of the Workshop on Computational Learning Theory*, 1992.

71. Andrzej Ehrenfeucht and David Haussler. Learning decision trees from random examples. *Information and Computation*, 82:231–246, 1989.

72. Andrzej Ehrenfeucht, David Haussler, Michael Kearns, and Leslie Valiant. A general lower bound on the number of examples needed for learning. *Information and Computation*, 82:247–261, 1989.

73. Usama M. Fayyad and Keki B. Irani. What should be minimized in a decision tree? In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 749–754, 1990.

74. Cèsar Ferri-Ramírez, José Hernández-Orallo, and M.J. Ramírez-Quintana. Learning functional logic classification concepts from databases. In *Proceedings of the 9th International Workshop on Functional and Logic Programming*, pages 296–308. UPV University Press, 2000.

75. Peter A. Flach. The geometry of ROC space: Understanding machine learning metrics through ROC isometrics. In *Proceedings of the 20th International Conference on Machine Learning*, pages 194–201. Morgan Kaufmann, 2003.

76. Peter A. Flach, Christophe Giraud-Carrier, and John W. Lloyd. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *LNAI*, pages 185–194. Springer-Verlag, 1998.

77. Yoav Freund and Llew Mason. The alternating decision tree learning algorithm,. In *Proceedings of the 16th International Conference on Machine Learning*, pages 124–133. Morgan Kaufmann, 1999.

78. Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

79. Takeshi Fukuda, Yasuhiko Morimoto, and Shinichi Morishita. Constructing efficient decision trees by using optimized numeric association rules. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 146–155, 1996.

80. Johannes Fürnkranz and Peter Flach. Roc 'n' rule learning – towards a better understanding of covering algorithms. *Machine Learning*, 58(1):39–77, 2005.

81. Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W.H. Freeman, 1979.

82. Thomas Gärtner, Peter A. Flach, Adam Kowalczyk, and Alex J. Smola. Multi-instance kernels. In C. Sammut and A. Hoffman, editors, *Proceedings of the 19th International Conference*, pages 179–186. Morgan Kaufmann, 2002.

83. Thomas Gärtner, John W. Lloyd, and Peter A. Flach. Kernels and distances for structured data. *Machine Learning*, 57:205–232, 2004.

84. Peter Geibel and Fritz Wysotzki. Learning relational concepts with decision trees. In *Proceedings of the 13th International Conference on Machine Learning*, pages 166–174, 1996.

85. Peter Geibel and Fritz Wysotzki. A logical framework for graph theoretical decision tree learning. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 173–180, 1997.

86. Mostefa Golea, Peter L. Bartlett, Wee Sun Lee, and Llew Mason. Generalization in decision trees and DNF: Does size matter? In *Advances in Neural Information Processing Systems 10*, pages 259–265, 1998.

87. Joshua Goodman. An incremental decision list learner. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 17–24, 2002.

88. Charles Gretton and Sylvie Thiébaux. Exploiting first-order regression in inductive policy selection. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 2004.

89. David Guijarro, Victor Lavin, and Vijay Raghavan. Monotone term decision lists. *Theoretical Computer Science*, 259:549–575, 2001.

90. Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniuqes*. Morgan Kaufmann, 2001.

91. Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2001.

92. David Haussler. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation*, 100(1):78–150, 1992.

93. Christoph Helma, Ross King, Stefan Kramer, and Ashwin Srinivasan (editors). *Proceedings of the PKDD01 Workshop on The Predictive Toxicology Challenge*. 2001. Available at `http://falcon.informatik.uni-freiburg.de/~ml/ptc/`.

94. Christoph Helma and Stefan Kramer. A survey of the Predictive Toxicology Challenge 2000-2001. *Bioinformatics*, 19:1179–1182, 2003.

95. José Hernández-Orallo and M.J. Ramírez-Quintana. Inductive functional logic programming. In S. Džeroski and P. Flach, editors, *Inductive Logic Programming*, number 1634 in LNAI, pages 116–127. Springer-Verlag, 1999.

96. Tamás Horváth and Turán György. Learning logic programs with structured background knowledge. *Artificial Intelligence*, 128(1-2):31–97, 2001.

97. Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

98. Aram Karalić and Ivan Bratko. First order regression. *Machine Learning*, 26(2/3):147–176, 1997.

99. Michael Kearns. Boosting theory towards practice: Recent developments in decision tree induction and the weak learning framework. In *Proceedings of the National Conference on Artificial Intelligence*, 1996.

100. Michael Kearns and Yishay Mansour. On the boosting ability of top-down decision tree learning algorithms. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 459–468. ACM Press, 1996.

101. Michael Kearns and Robert Schapire. Efficient distribution-free learning of probabilistic concepts. *Journal of Computer and System Sciences*, 41(1):67–95, 1994.

102. Michael J. Kearns, Yishay Mansour, Andrew Y. Ng, and Dana Ron. An experimental and theoretical comparison of model selection methods. *Machine Learning*, 27:7–50, 1997.

103. Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

104. Roni Khardon. Learning function free horn expressions. *Machine Learning*, 37:241–275, 1999.

105. Jörg-Uwe Kietz and Sašo Džeroski. Inductive logic programming and learnability. *SIGART Bulletin*, 5(1):22–32, 1994.

106. Jörg-Uwe Kietz and Stefan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, 1992.

107. Ross King, Stephen Muggleton, Ashwin Srinivasan, and Michael Sternberg. Structure-activity relationship derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity in inductive logic programming. *Proceedings of the National Academy of Sciences*, 93:438–442, 1996.

108. Ross D. King, Ashwin Srinivasan, and Luc Dehaspe. Warmr: a data mining tool for chemical data. *Journal of Computer-Aided Molecular Design*, 15:173–181, 2001.

109. Donald E. Knuth. *Literate Programming*. CSLI, 1992.

110. Donald E. Knuth. Are toy problems useful. In *Selected Papers on Computer Science*, chapter 10. CSLI, 1996.

111. Donald E. Knuth. Theory and practice, IV. In *Selected Papers on Computer Science*, chapter 9. CSLI, 1996.

112. Donald E. Knuth. The dangers of computer science theory. In *Selected Papers on Analysis of Algorithms*, chapter 2, pages 19–26. CSLI, 2000.

113. Adam Kowalczyk. Learning curves for anti-learning data. 2005.

114. Stefan Kramer. Structural regression trees. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 812–819. AAAI Press, 1996.

115. Stefan Kramer. *Relational Learning vs propositionalization: investigations in inductive logic programming and propositional machine learning*. PhD thesis, Vienna University of Technology, Vienna, Austria, 1999.

116. Stefan Kramer, Nada Lavrač, and Peter Flach. Propositionalization approaches to relational data mining. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 11. Springer, 2001.

117. Stefan Kramer, Bernhard Pfahringer, and Christoph Helma. Stochastic propositionalization of non-determinate background knowledge. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, pages 80–94, 1998.

118. Stefan Kramer and Gerhard Widmer. Inducing classification and regression trees in first order logic. In Sašo Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 6. Springer, 2001.

119. John Langford. *Quantitatively Tight Sample Complexity Bounds*. PhD thesis, Carnegie Mellon University, 2001.

120. John Langford. Tutorial on practical prediction theory for classification. *Journal of Machine Learning Research*, 6:273–306, 2005.

121. John Langford and Bianca Zadrozny. Estimating class membership probabilities using classifier learners. In *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics*, 2005.

122. Nada Lavrač and Sašo Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, New York, 1994.

123. Nada Lavrač and Peter Flach. An extended transformation approach to inductive logic programming. *ACM Transactions on Computational Logic*, 2 (4):458–494, 2001.

124. Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

125. Nick Littlestone and Manfred Warmuth. Relating data compression and learnability. Technical report, University of California, Santa Cruz, 1986.

126. John W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

127. John W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition, 1987.

128. John W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, Bristol University, 1995.

129. John W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.

130. John W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Cognitive Technologies. Springer, 2003.

131. John W. Lloyd. Modal higher-order logic for agents. 2004.

132. Donato Malerba, Floriana Esposito, Michelangelo Ceci, and Annalisa Appice. Top-down induction of model trees with regression and splitting nodes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):612–625, 2004.

133. Michel Manago. Knowledge-intensive induction. In *Proceedings of the 6th International Workshop on Machine Learning*, pages 151–155, 1989.

134. Yishay Mansour and David McAllester. Generalization bounds for decision trees. In *Proceedings of the 13th Annual Conference on Computational Learning Theory*, pages 69–80. Morgan Kaufmann, San Francisco, 2000.

135. Mario Marchand and John Shawe-Taylor. The set covering machine. *Journal of Machine Learning Research*, 3:723–746, 2002.

136. Dragos D. Margineantu and Thomas G. Dietterich. Improved class probability estimates from decision tree models. In D.D. Denison, C.C. Holmes, M.H. Hansen, B. Mallick, and B. Yu, editors, *Nonlinear Estimation and Classification*, number 171 in Lecture Notes in Statistics. Springer-Verlag, 2002.

137. Oded Maron and Tomás Lozano-Pérez. A framework for multiple-instance learning. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

138. Llew Mason. *Margins and Combined Classifiers*. PhD thesis, Research School of Information Sciences and Engineering, The Australian National University, 1999.

139. Ron Meir and Gunnar Rätsch. An introduction to boosting and leveraging. In Shahar Mendelson and Alex Smola, editors, *Advanced Lectures in Machine Learning LNCS2600*. Springer, 2003.

140. Donald Michie, Stephen Muggleton, David Page, and Ashwin Srinivasan. To the international computing community: A new east-west challenge. Technical report, Oxford University Computing Laboratory, 1994.

141. John Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3:319–342, 1989.

142. Raymond J. Mooney and Mary Elaine 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.

143. Shinichi Morishita and Jun Sese. Traversing itemset lattices with statistical metric pruning. In *Symposium on Principles of Database Systems*, pages 226–236, 2000.

144. Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

145. Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

146. Stephen Muggleton and David Page. Beyond first-order learning: Inductive learning with higher-order logic. Technical Report PRG-TR-13-94, Oxford University Computing Laboratory, 1994.

147. Stephen Muggleton and David Page. A learnability model for universal representations. Technical Report PRG-TR3-94, Oxford University Computing Laboratory, 1994.

148. Stephen Muggleton and David Page. A learnability model for universal representations and its application to top-down induction of decision trees. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence 15*, pages 248–267. Oxford University Press, 1998.

149. Stephen H. Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the 5th International Conference on Machine Learning*, pages 339–352, San Mateo, CA, 1988. Morgan Kaufmann.

150. Patrick M. Murphy and Michael J. Pazzani. Exploring the decision forest: an empirical investigation of Occam's razor in decision tree induction. *Journal of Artificial Intelligence Research*, 1:257–275, 1994.

151. Thomas Natschläger and Michael Schmitt. Exact VC-dimension of boolean monomials. *Information Processing Letters*, 59:19–20, 1996.

152. Ziv Nevo and Ran El-Yaniv. On online learning of decision lists. *Journal of Machine Learning Research*, 3:271–301, 2002.

153. Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: Symbols and search. In *ACM Turing Award Lectures – The First Twenty Years*. Addison-Wesley, 1987.

154. Kee Siong Ng. *The Alkemy Source Book*. Computer Sciences Laboratory, ANU, 2003. 330 pages. Available at `http://rsise.anu.edu.au/~kee/`.

155. Kee Siong Ng, John W. Lloyd, and Andrew W. Slater. Predictive toxicology using a decision-tree learner. In *Proceedings of the PKDD01 Workshop on The Predictive Toxicology Challenge*, 2001.

156. Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Number 1228 in LNAI. Springer, 1997.

157. Takashi Okada. Characteristic substructures and properties in chemical carcinogens studied by the cascade model. *Bioinformatics*, 19:1201–1207, 2003.

158. David Page and Ashwin Srinivasan. ILP: A short look back and a longer look forward. *Journal of Machine Learning Research*, 4:415–430, 2003.

159. Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

160. Toan Phung, Michael Winikoff, and Lin Padgham. Learning within the BDI framework: An empirical analysis. In *Proceedings of the International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, 2005.

161. John Ross Quinlan. Learning with continuous classes. In *Proceedings of the 2nd Australian Conference on Artificial Intelligence*, pages 343–348. World Scientific, 1992.

162. John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

163. John Ross Quinlan. Bagging, boosting and C4.5. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 725–730. AAAI Press, 1996.

164. Jan Ramon and Luc De Raedt. Multi instance neural networks. In *Proceedings of the ICML-2000 Workshop on Attribute-Value and Relational Learning*, 2000.

165. Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

166. Ann M. Richard. Structure-based methods for predicting mutagenicity and carcinogenicity: are we there yet? *Mutation Research*, 400:493–507, 1998.

167. Bradley L. Richards and Raymond J. Mooney. First order theory revision. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 447–451. Morgan Kaufmann, 1991.

168. Ryan Rifkin and Aldebaro Klautau. In defense of one-vs-all classification. *Journal of Machine Learning Research*, 5:101–141, 2004.

169. Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

170. Ulrich Rückert and Stefan Kramer. Towards tight bounds for rule learning. In *Proceedings of the 21st International Conference on Machine Learning*. Morgan Kaufmann, 2004.

171. Cynthia Rudin, Ingrid Daubechies, and Robert E. Schapire. The dynamics of AdaBoost: Cyclic behavior and convergence of margins. *Journal of Machine Learning Research*, 5:1557–1595, 2004.

172. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

173. Claude A. Sammut. *Learning concepts by performing experiments*. PhD thesis, Department of Computer Science, University of New South Wales, 1981.

174. Claude A. Sammut. The origins of inductive logic programming. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, 1993.

175. Claude A. Sammut and Ranan 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, chapter 7, pages 187–191. Morgan Kaufmann, 1986.

176. D.M. Sanderson and C.G. Earnshaw. Computer prediction of possible toxic action from chemical structure: the DEREK system. *Human and Experimental Toxicology*, 10:261–273, 1991.

177. Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.

178. Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26 (5):1651–1686, 1998.

179. Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels*. MIT Press, 2002.

180. Ehud Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.

181. John Shawe-Taylor, Peter L. Bartlett, Robert Williamson, and Martin Anthony. Structural risk minimization over data-dependent hierarchies. *IEEE Transactions on Information Theory*, 44(5):1926–1940, 1998.

182. Hans-Ulrich Simon. General bounds on the number of examples needed for learning probabilistic concepts. *Journal of Computer and System Sciences*, 52:239–254, 1996.

183. John Slaney and Silvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125:119–153, 2001.

184. Marina Sokolova, Mario Marchand, Nathalie Japkowicz, and John Shawe-Taylor. The decision list machine. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 921–928. MIT Press, 2003.

185. Eduardo Sontag. VC dimension of neural networks. In Chris M. Bishop, editor, *Neural Networks and Machine Learning*, pages 69–95. Springer-Verlag, 1998.

186. Ashwin Srinivasan, Stephen Muggleton, Ross King, and Michael Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.

187. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

188. Kerry Taylor. *Autonomous Learning by Incremental Induction and Revision*. PhD thesis, Department of Computer Science, The Australian National University, 1996.

189. Paul E. Utgoff. ID5: An incremental ID3. In *Proceedings of the 5th International Conference on Machine Learning*. Morgan Kaufmann, 1988.

190. Paul E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–186, 1989.

191. Paul E. Utgoff. An improved algorithm for incremental induction of decision trees. Technical Report 94-07, Department of Computer Science, University of Massachusetts, 1994.

192. Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997.

193. Leslie G. Valiant. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27(11):1134–1142, 1984.

194. Wim Van Laer and Luc De Raedt. How to upgrade propositional learners to first order logic: A case study. In Saso Džeroski and Nada Lavrač, editors, *Relational Data Mining*, chapter 10. Springer, 2001.

195. Vladimir N. Vapnik and Alexey Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

196. Larry Watanabe and Larry A. Rendell. Learning structural decision trees from examples. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 770–776. Morgan Kaufmann, 1991.

197. Chris Watkins. Q-learning. *Machine Learning*, 8:229–256, 1992.

198. Geoffrey I. Webb. Further experimental evidence against the utility of Occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417, 1996.

199. Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23:69–101, 1996.

200. Barry Wilkinson and Michael Allen. *Parallel Programming : Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, 1999.

201. Yin-Tak Woo, David Y. Lai, Mary F. Argus, and Joseph C. Arcos. Development of structure-activity relationship rules for predicting carcinogenic potential of chemicals. *Toxicology Letters*, 79:219–228, 1995.

202. Xiaobing Wu. Knowledge representation and inductive learning with XML. In *Proceedings of the International Conference on Web Intelligence*, pages 491–494, 2004.

203. Qi Zhang and Sally A. Goldman. EM-DD: An improved multiple-instance learning technique. Technical Report WUCS-01-17, Department of Computer Science, Washington University, 2002.

Index