KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A – 3001 Leuven (Heverlee)

TOP-DOWN INDUCTION OF FIRST ORDER LOGICAL DECISION TREES

Jury :
Prof. Dr. ir. E. Aernoudt, voorzitter
Prof. Dr. ir. M. Bruynooghe, promotor
Prof. Dr. L. De Raedt, promotor
Prof. Dr. B. Demoen
Dr. S. Džeroski,
Institut "Jožef Stefan", Ljubljana, Slovenië
Prof. Dr. D. Fisher,
Vanderbilt University, Nashville, Tennessee, USA
Prof. Dr. I. Van Mechelen

Proefschrift voorgedragen tot het behalen van het doctoraat in de toegepaste wetenschappen

door

Hendrik BLOCKEEL

U.D.C. 681.3*I26

December 1998

D/1998/7515/57

ISBN 90-5682-156-3

Preface

Do. Or do not. There is no try.
— Yoda, Jedi master

This text describes the main results of the research I performed with the Machine Learning group of the Department of Computer Science at the Katholieke Universiteit Leuven. On a global level, this research is about the application possibilities of inductive logic programming in the context of databases. More specifically, the research focuses on two topics. The first topic is what we call inductive database design: finding a good design for a (deductive) database from a set of extensionally defined relations. The second topic is the application of inductive logic programming for data mining. As both topics differ a bit too much to be handled elegantly in one single text, this text only encompasses the second topic. The first one has been described in the literature.

I guess it is needless to mention that this work would not have been realized without the help of a great many people. It does not seem possible to give even a probably approximately complete overview of all the people who have significantly influenced it. Nevertheless I would like to express my gratitude to some people in particular.

First of all, I want to thank the Flemish Institute for the Promotion of Scientific and Technological Research in the Industry (Vlaams Instituut voor de Bevordering van het Wetenschappelijk-Technologisch Onderzoek in de Industrie, IWT) for funding this research. Their financial support is not the only thing I am grateful for, however; it has also been very pleasing to notice that they actively show interest in the research they fund, by following up projects and regularly sending reports on the activities of the institute.

During the four years I spent at the Leuven machine learning group I have had the pleasure to work together with many different people, all of whom contributed in their own way to this work and to the stimulating environment that made it possible. I first want to mention Hilde Adé and Gunther Sablon, who were working in the Leuven machine learning group at the moment I joined it. Being seasoned researchers, they did not only share a lot of their knowledge with the newer members of the group, but most of all, they created a very

pleasant atmosphere at work — an atmosphere that is still there three years after they have left the group.

Luc Dehaspe and Wim Van Laer joined the group around the same time I did, and they are the PhD students with whom I have co-operated most closely. The countless spontaneous discussions among us have had an important influence on my work (and I hope also on theirs!). The most physical evidence of their contributions to this work, however, can probably be found inside the code of the TILDE system. A significant part of this code was borrowed directly from Wim's ICL and Luc's CLAUDIEN system. By making available very readable and reusable code, they have not only significantly sped up the development of TILDE, but most of all, motivated me to implement it in the first place.

I would not do justice to these people, however, if I only praised their professional qualities. They also proved to be highly enjoyable company both at work and in between.

The arrival of Nico Jacobs, Kurt Driessens and Jan Ramon further added to the good atmosphere in our machine learning group — and also to the cosiness of our office, for that matter. They were also the first users of TILDE, alpha-testers one might say, and their feedback has been very helpful. Nico and Jan have also contributed to this work by performing some of the experiments mentioned here.

It will probably not come as a surprise that the people I owe most to are my promotors: Maurice Bruynooghe and Luc De Raedt. Maurice is undoubtedly the person who has had the most condensed influence on this work. While he has followed it from a greater distance than Luc, and contacts with him were less frequent, on those occasions where he did have some advice it was usually short but extremely useful. Some people say a picture is worth a thousand words. I guess Maurice must talk in pictures, then.

But the person who has had by far the most influence on this work is Luc De Raedt. I cannot express how much I owe to him in only a few paragraphs — actually, I'm not sure if even Maurice could. It was Luc who kindled my interest in machine learning in the first place, and who motivated me to apply for a grant at the IWT. It was also he who came up with many of the ideas that have been fleshed out in this text.

But I can imagine that these tasks, while performed in an excellent manner, are still part of the standard job specification of a good promotor. Luc's coaching has gone far beyond that, however. I feel he has really been a teacher to me: a teacher who taught me how to do research and how to report on it; who continuously evaluated my work and offered suggestions for improvements; who pointed to existing work related to mine and brought me into contact with many people in the machine learning community; and who promoted and defended my work at every opportunity.

I am also very grateful for his support during the writing of this text, for his many comments and his insisting on improving the text when I would have settled for less. When, at some moment during the preparation of this text, I had found the time to watch the Star Wars Trilogy, and heard Yoda speak the words quoted at the top of this text, it occurred to me that no words could better describe Luc's attitude.

Of the many people I have met outside the machine learning group of Leuven, I would like to thank in particular a few people who have had a special influence on this work. First, I want to mention Sašo Džeroski, who has at several times co-operated with our group, and who has been one of the most enthousiastic users of the TILDE system.

At the *Fifteenth International Conference on Machine Learning* I have had the pleasure to meet Douglas Fisher, who was so kind to set some time aside for a discussion and to point us to some relevant work. It has been particularly pleasing to see that both Douglas and Sašo accepted the invitation to serve as a Jury member for this dissertation.

I also want to thank the members of the reading committee for their very useful comments on an earlier draft of this text. Besides their improving the final form of this text, each of them has had a significant influence on the research itself. I already mentioned Luc De Raedt and Maurice Bruynooghe. Iven Van Mechelen has influenced this work mainly through his course on *Inductive classification methods*. This course provided a very different view on classification than the typical inductive logic programming views, and this has broadened the scope of this text a lot. With respect to the implementational aspects I want to thank Bart Demoen, who has often offered help and explanations concerning the Prolog engine we use, and who has actively participated in improving the efficiency of TILDE.

Special thanks go to Wim Van Laer, who has been so kind to proof-read the Dutch summary of this text.

I would like to end these acknowledgements by mentioning Frank Matthijs and Bart Vanhaute, who had nothing to do with this text or the research described in it. In a sense they have been all the more valuable to me because of that.

List of Symbols

The following list indicates the meaning of symbols that are commonly used throughout the text. Some of these symbols (e.g., single letters) may in specific cases be used in a different context; in those cases their meaning is always explicitly mentioned.

$\models$	entailment
$\wedge$	conjunction operator
$\vee$	disjunction operator
$\neg$	negation operator
$\leftarrow$	implication operator
$\tilde{S}$	information corresponding to a space S
2^S	power set of S
B	background theory
c	a clause
C	a cluster
$\mathcal{C}$	a clustering
$\mathbf{C}(E)$	the clustering space of a set of instances E
$\mathbf{C}_{\mathcal{L}}(E)$	intensional clustering space of E w.r.t. a language $\mathcal{L}$
C_n^x	binomial coefficient : $C_n^x = n!/(x! \cdot (n-x)!)$
$conj$	a conjunction
CU	class utility
D	description space; the subspace of I disjoint with P
d	a distance
d_E	Euclidean distance
$d_=$	equality distance
dl	function mapping decision trees on decision lists
e	an instance from a given set of instances E
E	a set of instances
$\mathcal{E}$	a partition of a set of instances
$\mathbf{E}$	expected value operator

f	cluster assignment function
$F(s)$	a logical formula that refers to the variable s
I	instance space
L	subspace of $I - P$ available only to the learner
$\mathcal{L}$	a language
λ	lookahead operator
$Lits(c)$	set of literals occuring in a clause c
$\mathcal{M}(P)$	minimal Herbrand model of logic program P
MSE	mean squared error of prediction
p	prototype function
P	prediction space
$pred_{\mathcal{C}}$	predictor function defined by clustering $\mathcal{C}$
PU	partition utility
π	target function
Q	quality criterion
$\leftarrow Q$	query
$\mathbb{R}$	the set of real numbers
$\mathbb{R}^+$	the set of positive real numbers
RE	relative error
ρ	refinement operator
SS_T	total sum of squared distances
SS_B	sum of squared distances between sets
SS_W	sum of squared distances within sets
tr	function mapping decision lists on decision trees
τ	a test in a node of a decision tree
$\mathcal{T}$	a set of tests
x	an arbitrary instance
$\mathbf{x}, \mathbf{y}$	instances represented as vectors
$x_1, \ldots, x_n$	first $\ldots n$-th component of a vector $\mathbf{x}$
Y	target variable
y	a value of the target variable

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work is situated at the intersection of several scientific domains. It is primarily about machine learning, which in itself is a sub-domain of artificial intelligence. Much of the research in machine learning is also relevant to the quite new field of data mining. And finally, the research builds on logic programming, whence it borrows a knowledge representation formalism.

In this introductory chapter, we situate our research in the context of these different fields.

1.1 Machine Learning and Artificial Intelligence

The term *artificial intelligence* is hard to define, which is reflected by the fact that many different definitions exist. Russell and Norvig (1995) (p. 5) give a nice overview of several definitions, classifying them along two dimensions.

For this introduction, however, it is probably easiest to adopt the following nearly trivial definition:

Definition 1.1 (Artificial intelligence) *Artificial intelligence, as a scientific field, is the study of how to make machines exhibit the kind of intelligence that human beings exhibit.*

While it is difficult to give a concise definition of human intelligence, it is relatively easy to identify certain characteristics of it, such as the ability to reason, to be creative, etc. One very important characteristic is the ability to *learn*. Any agent (be it a human, an animal or a machine) that could be called intelligent, should at least be able to learn from its past experience. An agent that blindly makes the same mistakes over and over again would never be called intelligent.

Langley (1996) proposes the following definition of learning:

Definition 1.2 (Learning) (Langley, 1996) *Learning is the improvement of performance in some environment through the acquisition of knowledge resulting from experience in that environment.*

Machine learning can then be trivially defined as follows:

Definition 1.3 (Machine learning) *Machine learning, as a scientific field, is the study of how to make machines learn.*

According to Langley's definition of learning, the learning process consists of two subtasks: acquiring knowledge, and putting it to use. In this text we will be concerned mainly with the acquisition of knowledge. More precisely, we consider the inference of a general theory (the knowledge) from a set of observations (the experience). This reasoning from specific to general is called *inductive* reasoning.

Example 1.1 The following are some examples of inductive reasoning and learning:

- An amateur bird watcher observes 10 ravens and, noticing that these 10 are all black, induces that all ravens are black.
- A child tries to build towers from blocks with different shapes (cubes, cones, spheres, ...). After several trials the child induces that a cube on top of a cone is never stable, and a cone on top of a cube is always stable.

If the child uses the discovered knowledge to build higher towers, we can say, according to Langley's definition, that it has learned something. The amateur bird watcher could use the knowledge that all ravens are black when attempting to classify new birds. ◇

A system that is able to induce knowledge but cannot use this knowledge is not a learning system, according to the above definition. In the literature, however, learning is often used as a synonym for inductive reasoning, and in the remainder of this text we will also use it in this broader sense. Only in this introduction, we stick to the narrow definition of learning. When a system induces knowledge, not to improve its own performance, but simply to provide other systems (or humans) with that knowledge, we call it a *knowledge discovery* system.

1.2 Data Mining and Knowledge Discovery

Knowledge discovery (Frawley *et al.*, 1991; Piatetsky-Shapiro and Frawley, 1991; Fayyad *et al.*, 1996) is a field that has recently become very popular in

both the scientific and the industrial community. Companies are interested in the field because of its large application potential for market studies, process optimization, and other ways of increasing their gains.

Example 1.2 Some examples of business applications of knowledge discovery:

- A bank might discover that a certain service mainly appeals to a specific market segment, and hence focus a promotional campaign on that segment.
- A supermarket might discover that two products are often bought together, and put these products far away from one another so that many customers have to walk past many other products, in the hope that this will increase sales.

◇

The knowledge discovery task is often described as follows:

Definition 1.4 (Knowledge discovery) (Frawley *et al.*, 1991) *Knowledge discovery is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data.*

The process of knowledge discovery actually consists of several subtasks (see also Figure 1.1):

- *Pre-processing of the data.* The data in their original form may contain missing values, noise, or may simply not be in a format fit for applying a data mining algorithm. Transforming the data into a suitable format can be an elaborate and non-trivial task.
- *Data mining.* One or more algorithms are used to extract general laws, patterns or regularities from the data.
- *Post-processing of the results.* The results obtained by the data mining algorithm may not be in an easily interpretable format. It can therefore be desirable to transform them into another, more intelligible format.

The terms "data mining" and "knowledge discovery" have been used somewhat inconsistently in the early literature. More recently a convention was generally adopted that "data mining" refers to the central inductive reasoning task, and "knowledge discovery" refers to the whole process depicted in Figure 1.1. We follow this terminology in this text.

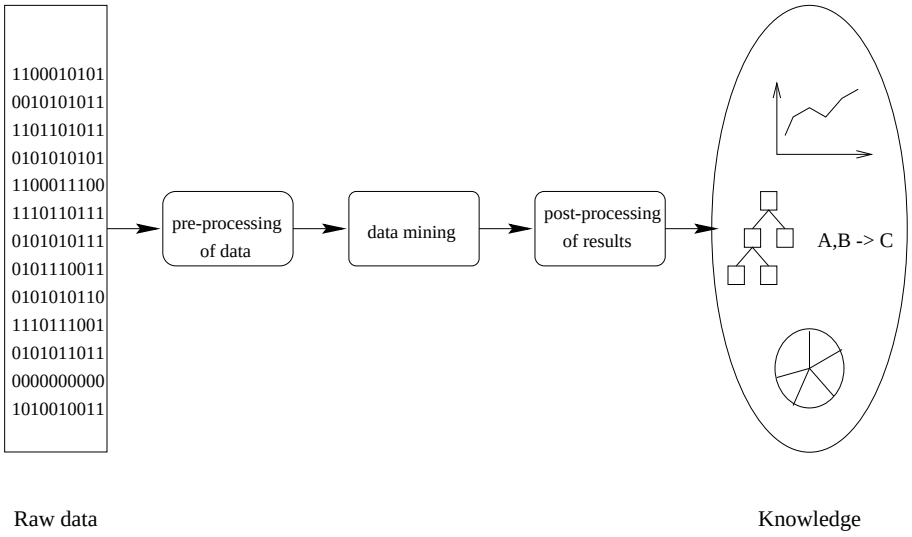


Figure 1.1: The knowledge discovery process.

1.3 Logic Programming and Inductive Logic Programming

Logic programming is a programming paradigm in which programs consist of first order predicate logic formulae. The main representative of logic programming languages is Prolog. In this text it is assumed that the reader is familiar with logic programming and Prolog. Readers for whom this assumption does not hold can consult (Bratko, 1990; Sterling and Shapiro, 1986; Clocksin and Mellish, 1981) (about Prolog) or (Kowalski, 1979; Lloyd, 1987) (about logic programming), which are excellent introductions to these topics.

Logic programming engines (such as Prolog systems) can *deduce* facts from logic formulae; i.e. they can compute those facts that are certainly true, assuming the formulae they start from are correct. A classic example of deductive reasoning is the following: *all humans are mortal, and Socrates is human, hence Socrates is mortal.* In Prolog the premises of the reasoning would be written as follows:

```
mortal(X) :- human(X).    {for all X: if X is human then X is mortal}
human(socrates).          {socrates is human}
```

(the `:-` symbol represents the implication operator $\leftarrow$). One could ask the Prolog system whether Socrates is mortal, in which case the system would answer `yes`, or ask it for which `X` it can prove that `X` is mortal, in which case it

would answer `X = socrates`.

While the classical logic programming engines implement deductive reasoning, *inductive logic programming* (ILP) engines implement inductive reasoning. One could e.g. provide an inductive logic programming system with the following data:

```
raven(bird1).
raven(bird2).

black(bird1).
black(bird2).
```

and ask it what properties ravens have. The system could then come up with the rule `black(X) :- raven(X)`: all ravens are black.

In contrast to deductive reasoning, inductive reasoning does not guarantee that the answer is correct (seeing 2 black ravens does not guarantee that all ravens in the world are black). Therefore the result of inductive inference is usually called a *hypothesis*. Such a hypothesis needs some external motivation, such as statistical evidence. In the above example the evidence is rather weak; observing large numbers of ravens that are all black would make it stronger.

Induction is harder than deduction, and currently inductive logic programming is very much a research topic. Many different techniques and approaches exist, but at present there is no single language or framework for inductive logic programming that has the status Prolog has for deductive logic programming.

1.4 Connections Between These Fields

Both machine learning and data mining depend heavily on inductive reasoning. As data mining has only recently started to receive much attention, the field is less mature than machine learning. It is therefore not surprising that it borrows many techniques from machine learning (and also from statistics, for that matter).

Since inductive logic programming provides a framework for inductive reasoning, it is an obvious candidate as a paradigm for both machine learning and data mining. However, until now, it has not been used extensively for machine learning, and even less for data mining. The main reason for this seems to be its computational complexity. While ILP is more powerful than most other techniques, such as attribute value learning, this power comes at an efficiency cost. Especially in the context of data mining, where very large data sets are often considered, efficiency is crucial.

Learning from interpretations (De Raedt and Džeroski, 1994) is a relatively novel setting for ILP that makes it possible to alleviate this efficiency problem. The learning from interpretations setting could be situated somewhere between

classical ILP and attribute value learning, with respect to both efficiency and expressive power. The learning from interpretations setting opens up new possibilities for using inductive logic programming in the fields of machine learning and data mining.

1.5 Motivation and Contributions

Attribute value learning is much more mature than inductive logic programming, and many sophisticated techniques for a variety of tasks exist for that formalism. The main motivation for this work is the desirability of upgrading some of these techniques to the first order framework of inductive logic programming.

A first contribution of this text is that we unify several induction tasks (classification, regression, and certain kinds of clustering) into one framework which we call predictive clustering. The result is one general inductive technique that can be specialized towards more specific tasks.

A second contribution is the study of first order logical decision trees. These are an upgrade of the classical propositional decision trees, as used for attribute value learning, to first order predicate logic. The properties of these trees are studied, and it is shown that within the learning from interpretations framework they are more expressive than the flat logic programs that most ILP systems induce. This study also sheds new light on several other representation formalisms and puts them in perspective.

As a third contribution, we upgrade the induction of decision trees in the general framework of predictive clustering towards inductive logic programming by means of first order logical decision trees. We present an implementation of the technique and evaluate it empirically. The system is the first inductive logic programming system to combine classification, regression, and several kinds of clustering.

1.6 Structure of this Text

In Chapter 2 we discuss the clustering task in detail, and we identify a special kind of clustering that we call predictive clustering. We show that certain other inductive tasks that are usually not considered to be clustering tasks (induction of classifiers, regression) are in fact special cases of predictive clustering.

In Chapter 3 we demonstrate how induction of decision trees, a technique that is often used for induction of classifiers or regression, can be generalized to predictive clustering. We show that many classical approaches to decision tree induction are instantiations of our general framework.

In Chapter 4 different representation formalisms for learning are compared. One of these, the inductive logic programming setting called *learning from interpretations* will be used throughout the remainder of the text.

In Chapter 5 we introduce and study first order logical decision trees. These trees form a stepping stone for upgrading the general decision tree induction technique from Chapter 3 to inductive logic programming.

Chapter 6 presents the TILDE system, which is an implementation of the general decision tree induction technique from Chapter 3 that induces first order logical decision trees as defined in Chapter 5. Details of the implementation are discussed and the system is evaluated empirically.

In Chapter 7 we investigate how the proposed techniques scale up towards large data sets, as the ability to handle large data sets is essential for data mining. We discuss a re-implementation of TILDE that aims specifically at working with large data sets and evaluate its scaling properties empirically.

Chapter 8 concludes by discussing the main contributions of this work and pointing to future work.

Chapter 2

A Clustering Approach to Prediction

2.1 Introduction

Within machine learning and knowledge discovery a large variety of predictive induction techniques exist. While it is clear that many of these are related to one another, few authors stress this relationship or explicitly exploit it. There are, for instance, many texts on classification and regression, but few texts treat both at the same time (the CART book (Breiman *et al.*, 1984) being an important exception). In this text we try to stress the similarities between the different tasks by explicitly taking the viewpoint that they are just specific kinds of clustering.

In this chapter we introduce clustering, and discuss how it can form the basis of a general approach to predictive induction that encompasses many existing approaches to classification and regression but also offers further possibilities. This predictive clustering approach is the one that will be taken in the remainder of this text.

We first introduce the concepts of distances and prototypes (Section 2.2). In Section 2.3 we discuss clustering. We show how a special kind of clustering that we call predictive clustering generalizes over many current approaches to classification and regression, both supervised and unsupervised. We also locate our approach by giving a more global (though certainly not complete) view on clustering methods, discussing them along several dimensions. Finally, we discuss the evaluation of clusterings.

In Section 2.4 we take a closer look at how clusterings can be used for making predictions, situating existing approaches in this view. In Section 2.5 we discuss several ways in which the clusters can be formed. We again locate the classical

approaches, and show that our framework offers several opportunities that are not exploited by most of them or generalize over *ad hoc* methods. Section 2.6 discusses some related work and Section 2.7 concludes.

2.2 Distances and Prototypes

In order to find clusters of similar objects, one first needs a similarity measure. Similarity is often expressed by means of a *distance* measure: the more similar two objects are, the smaller the distance between them is. We define distances between instances and sets of instances, and then define the prototype of a set as the instance that is most representative for the set.

2.2.1 Distances Between Individual Examples

Definition 2.1 (Distance) *A function $d : I \times I \to \mathbb{R}^+$ is a distance on I if and only if it is positive definite, symmetric and fulfills the triangle inequality:*

$$\forall x, y \in I : d(x, y) \geq 0 \text{ and } d(x, y) = 0 \Leftrightarrow x = y \tag{2.1}$$

$$\forall x, y \in I : d(x, y) = d(y, x) \tag{2.2}$$

$$\forall x, y, z \in I : d(x, y) \leq d(x, z) + d(z, y) \tag{2.3}$$

In some cases these constraints are too restrictive and one may relax them; the resulting measure is then sometimes called a dissimilarity measure. In this text we will only use distances.

There are many different ways to define a distance. Even in the simplest case where example descriptions are vectors of numerical values, a choice can be made between, e.g.,

- the *Euclidean distance*:

$$d_E(\mathbf{x}, \mathbf{y}) = d_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (\mathbf{x}_i - \mathbf{y}_i)^2} \tag{2.4}$$

- the *Manhattan distance*:

$$d_1(\mathbf{x}, \mathbf{y}) = \sum_i |\mathbf{x}_i - \mathbf{y}_i| \tag{2.5}$$

- the *Chebyshev distance*:

$$d_\infty(\mathbf{x}, \mathbf{y}) = \max_i |\mathbf{x}_i - \mathbf{y}_i| \tag{2.6}$$

which are all special cases of the *Minkowski family of distances*:

$$d_L = (\sum_i |\mathbf{x}_i - \mathbf{y}_i|^L)^{1/L} \tag{2.7}$$

It may be necessary to rescale the different dimensions in order to make them comparable, e.g., to rescale so that the values along each dimension have a standard deviation of 1. The *Mahalanobis distance*

$$d_M = (\mathbf{x}_i - \mathbf{y}_i)C^{-1}(\mathbf{x}_i - \mathbf{y}_i)' \tag{2.8}$$

with C the covariance matrix of the vector components ($C_{ij} = \sigma_{ij}$) performs such rescaling, and also takes covariance between variables into account.

Another complication is that it may be desirable to apply transformations to the variables. E.g., is 1 kg of salt more similar to 1 g than 20 kg is to 21 kg? Probably not, but the difference 1 kg – 1 g = 999 g is smaller than 21 kg – 20 kg = 1 kg. A logarithmic transformation may be in place here.

For nominal variables a useful distance is the *equality distance* $d_=$:

$$d_=(x, y) = 0 \Leftrightarrow x = y \tag{2.9}$$

$$= 1 \text{ otherwise} \tag{2.10}$$

which for vectors of nominal variables generalizes to the *Hamming distance*:

$$d_H(\mathbf{x}, \mathbf{y}) = \sum_i d_=(\mathbf{x}_i, \mathbf{y}_i) \tag{2.11}$$

Until now we have only considered vectors of either numerical or nominal variables; the situation becomes more complex when a vector can consist of a mix of nominal and numerical variables. Different notions of distance have to be used for the different components, and the problem of incomparability becomes even larger; see e.g. (Wilson and Martinez, 1997) for a discussion and proposed solutions. The problem escalates even more when one uses first-order descriptions of examples. A veritable wealth of distances and dissimilarities has been proposed in this context, some of them very complex and not well understood; see e.g. (Emde and Wettschereck, 1996; Bisson, 1992b; Hutchinson, 1997; Nienhuys-Cheng, 1997; Ramon and Bruynooghe, 1998; Ramon *et al.*, 1998; Sebag, 1998).

In the following we assume that a distance measure d that computes the distance $d(e_1, e_2)$ between examples e_1 and e_2 is already given. How one should decide upon which distance to use is out of the scope of this text, and we refer to the literature mentioned above.

2.2.2 Distances Between Sets of Examples

Besides the need to measure the distance between individual examples, it is often also necessary to measure the distance between sets of examples. Again, different approaches have been suggested in the literature. The dissimilarity between two sets of instances S_1 and S_2 can, e.g., be defined as

- the smallest distance between any $x \in S_1$ and $y \in S_2$:

$$d'_{min,d}(S_1, S_2) = \inf_{x \in S_1, y \in S_2} d(x, y) \tag{2.12}$$

- the largest distance :

$$d'_{max,d}(S_1, S_2) = \sup_{x \in S_1, y \in S_2} d(x, y) \tag{2.13}$$

- the average distance :

$$d'_{avg,d}(S_1, S_2) = \sum_{x \in S_1, y \in S_2} d(x, y)/(|S_1| \cdot |S_2|) \tag{2.14}$$

- the *Hausdorff distance*:

$$d'_{Hausdorff,d}(S_1, S_2) = \max\{\sup_{x \in S_1} \inf_{y \in S_2} d(x, y), \sup_{x \in S_2} \inf_{y \in S_1} d(x, y)\} \tag{2.15}$$

- the *center distance*:

$$d'_{center,d,p}(S_1, S_2) = d(p(S_1), p(S_2)) \tag{2.16}$$

  where the function p mapping a set onto its center is to be specified (see below)

Note that even when the dissimilarity between single examples is a distance, the dissimilarity between sets does not necessarily inherit this property. The Hausdorff distance and the center distance inherit the properties of a distance, except the property that $d'(S_1, S_2) = 0 \Rightarrow S_1 = S_2$.

In this text we will adopt the center-based approach. The notion of the center of a set of examples is formalized in the concept of a prototype.[1]

[1] The term "prototype" was chosen here only because it conveys a suitable intuitive meaning. Our use of the term is not necessarily related to its use in other contexts.

2.2.3 Prototypes

Definition 2.2 (Prototype function, prototype) *A prototype function is a function $p : 2^I \rightarrow I$ that maps a set of instances E onto a single instance $p(E)$. $p(E)$ is called the prototype of E.*

Ideally, a prototype of a set E should be maximally representative for the instances in E, i.e., as close as possible to any $e \in E$.

Definition 2.3 (Ideal prototype function) *Given a distance d over an instance space I, a prototype function p is ideal if and only if*

$$\forall E \subseteq I : \forall x \in I : \sum_{e_i \in E} d(x, e_i)^2 \geq \sum_{e_i \in E} d(p(E), e_i)^2 \qquad (2.17)$$

While other criteria could be chosen instead of the least squares criterion, choosing the latter in Definition 2.3 ensures that in a Euclidean space (with distance d_E) the ideal prototype $p(E)$ is the mean of all the vectors in E, which is intuitively a good choice for a prototype. In a space of nominal values with distance $d_=(x, y)$, the ideal prototype corresponds to the mode (the most frequently occurring value), which is also desirable. In a first order domain, a prototype function could, e.g., be the (possibly reduced) least general generalization of the examples in the cluster, using Plotkin's (1970) notion of θ-subsumption or the variants corresponding to structural matching (Bisson, 1992b; De Raedt *et al.*, 1997).

Example 2.1 Given $I = \mathbb{R} \times C$ with $C = \{\texttt{blue}, \texttt{red}\}$, and $d(\mathbf{x}, \mathbf{y}) = (x_1 - y_1)^2 + d_=(x_2, y_2)$, a maximally representative prototype of the set of instances {`(1, blue)`, `(2, red)`, `(3, blue)`} is `(2, blue)`. ◇

Unless stated otherwise, in the remainder of this text we assume that a prototype function p is given, and that p is ideal. The dissimilarity between two sets of examples E_1 and E_2 is then defined as the distance $d(p(E_1), p(E_2))$ between the prototypes of the sets.

2.3 Clustering

2.3.1 Problem Definition

Before defining the clustering task, we define the concepts that are involved.

Definition 2.4 (Extensional flat clustering) *An extensional flat clustering $\mathcal{C}$ of a set of instances E is a partition of E.*

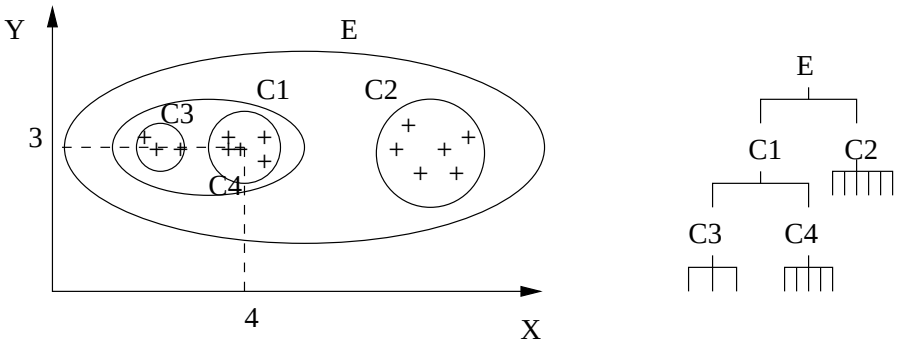


Figure 2.1: A hierarchical clustering.

Definition 2.5 (Extensional hierarchical clustering) *An extensional hierarchical clustering $\mathcal{C}$ of a set of instances E is a set of subsets of E such that*

$$E \in \mathcal{C} \tag{2.18}$$

$$\forall e \in E : \{e\} \in \mathcal{C} \tag{2.19}$$

$$\forall C_1, C_2 \in \mathcal{C} : C_1 \subseteq C_2 \text{ or } C_2 \subset C_1 \text{ or } C_1 \cap C_2 = \emptyset \tag{2.20}$$

Example 2.2 Figure 2.1 shows a set of data E, and an extensional hierarchical clustering of the set. The clustering is represented in two ways: by drawing the clusters in the instance space, and as a tree. The extensional hierarchical clustering represented in the figure is $\mathcal{C} = \{E, C_1, C_2, C_3, C_4\} \cup \bigcup_{e \in E} \{e\}$. Some flat clusterings are $\mathcal{C}_1 = \{C_1, C_2\}$ and $\mathcal{C}_2 = \{C_3, C_4, C_2\}$. ◇

Definition 2.6 (Cluster) *The elements of an extensional clustering are called clusters.*

An extensional clustering of a set E is uniquely determined by describing the clusters (which are just sets of instances) in it. We distinguish two kinds of descriptions:

Definition 2.7 (Extensional description) *An extensional description of a set S is of the form $S = \{s_1, s_2, \ldots, s_n\}$, i.e. an enumeration of its elements.*

Definition 2.8 (Intensional description) *An intensional description of a set S in a language $\mathcal{L}$ is of the form $S = \{s|F(s)\}$ where $F(s)$ is a sentence (formula) in $\mathcal{L}$.*

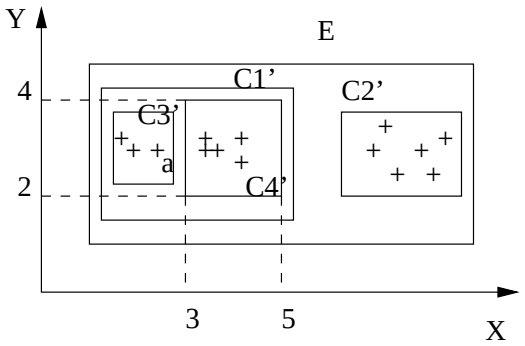


Figure 2.2: An intensional hierarchical clustering, different from the one in Figure 2.1.

While there is a one-to-one correspondence between an extensional clustering of a set E and the extensional description of its clusters, the correspondence with intensional cluster descriptions is one-to-many. I.e., different intensional cluster descriptions may correspond to the same extensional clustering.

Example 2.3 In Figure 2.1 the cluster C_4 could be represented intensionally as $C_4 = \{(x, y) \in E | (x-4)^2 + (y-3)^2 < 1\}$, which is suggested by drawing its boundary as a circle. Figure 2.2 represents a hierarchical clustering $\mathcal{C}'$ that is in a sense different; the intensional description of cluster C'_4 could, e.g., be $C'_4 = \{(x, y) \in E | 3 < x < 5 \wedge 2 < y < 4\}$. Still, the extensional clusterings defined by $\mathcal{C}$ and $\mathcal{C}'$ are the same.

$\diamond$

In order to be able to distinguish clusterings that are intensionally different, we define intensional (as opposed to extensional) clusterings as follows:

Definition 2.9 (Intensional flat clustering) *Given an instance space I and a set of instances $E \subseteq I$, an intensional flat clustering $\mathcal{C}$ of E is a function $\mathcal{C} : I \to 2^I$ such that $\bigcup_{e \in E} \{\mathcal{C}(e)\}$ is an extensional flat clustering of E.*

Definition 2.10 (Intensional hierarchical clustering) *Given an instance space I and a set of instances $E \subseteq I$, an intensional hierarchical clustering $\mathcal{C}$ of E is a function $\mathcal{C} : I \to 2^{2^I}$ such that*

$$\forall e \in E : \{e\} \in \mathcal{C}(e) \tag{2.21}$$

$$\forall e \in E : E \in \mathcal{C}(e) \tag{2.22}$$

$$\forall e \in E : \forall C_1, C_2 \in \mathcal{C}(e) : C_1 \subseteq C_2 \vee C_2 \subseteq C_1 \tag{2.23}$$

$$\bigcup_{e \in E} \mathcal{C}(e) \text{ is an extensional hierarchical clustering.} \tag{2.24}$$

An intensional flat clustering of a set of instances E maps each instance of E onto a single cluster. An intensional hierarchical clustering of E maps each instance $e \in E$ onto a set of clusters that are contained in one another and in the tree representation form a path from the top to $\{e\}$.

Example 2.4 The instance labeled a in Figure 2.2 is mapped by $\mathcal{C}'$ onto the set of clusters $\{E, C'_1, C'_3, \{\mathtt{a}\}\}$, i.e., the set of all the clusters in $\mathcal{C}'$ that a belongs to. An intensional flat clustering could, e.g., map a onto C'_1. ◇

Assuming each cluster C_i in an extensional hierarchical clustering is described intensionally using a formula F_i, this set of intensional cluster descriptions uniquely defines a function

$$\mathcal{C} : I \rightarrow 2^{2^I} : C_i \in \mathcal{C}(x) \Leftrightarrow F_i(x) \tag{2.25}$$

which is an intensional hierarchical clustering according to Definition 2.10. Conversely, if two intensional hierarchical clusterings $\mathcal{C}$ and $\mathcal{C}'$ are equal, their intensional descriptions F_i and F'_i for any cluster C_i must be equivalent in the sense that $\forall x \in I : F_i(x) \Leftrightarrow F'_i(x)$. A similar reasoning can be made for intensional flat clusterings. We conclude that intensional clusterings are equal if and only if all their intensional cluster descriptions are equivalent; in other words, Definitions 2.10 and 2.9 adequately capture intensionality as defined in Definition 2.8.

Definition 2.11 (Clustering) *$\mathcal{C}$ is a clustering of E if and only if it is an intensional or extensional, hierarchical or flat, clustering of E.*

Definition 2.12 (Clustering space) *The set of all clusterings of E is called the clustering space of E and is written $\mathbf{C}(E)$.*

We can now formulate the clustering problem as follows:[2]

Definition 2.13 (Task definition for clustering) *We define the clustering task as follows:*

Given:

- *a set of instances E*
- *a distance d on E*
- *and a quality criterion Q defined over $\mathbf{C}(E)$*

[2]More general formulations exist. One possible extension is that the distance is replaced by a dissimilarity. A further extension is to start from a $n \times n$ *dissimilarity matrix*, i.e. no descriptions of the objects themselves are given, only the dissimilarities between them are; see e.g. (Sneath and Sokal, 1973). Still other approaches allow clusters to overlap.

Find:

- *a clustering* $\mathcal{C}$ *such that* $Q(\mathcal{C})$ *is optimal, i.e.* $\forall \mathcal{C}' \in \mathbf{C}(E) : Q(\mathcal{C}') \leq Q(\mathcal{C})$

The quality criterion Q is not specified in detail in Definition 2.13; typically, however, Q favors clusterings in which the distance between two clusters is large (unless one is a subcluster of the other) and the distance between elements of the same cluster is small.

We illustrate the clustering task with a toy example.

Example 2.5 Assume that a set of objects is given; each object is either a strawberry, a tomato or an apple. Figure 2.3 locates the objects in a two-dimensional space ($\texttt{Color} \times \texttt{Weight}$). A good flat clustering should put all strawberries in one cluster, tomatoes in another one and apples in a third cluster. A good hierarchical clustering should also identify these clusters, but could moreover identify, e.g., different types of apples. In this example we only consider a flat clustering.

In Figure 2.3 both extensional and intensional descriptions of the different clusters are shown. The language $\mathcal{L}$ for the intensional descriptions is assumed to be propositional logic, with propositions of the form $\mathit{Attribute}(x) \oplus \mathit{value}$ with $\mathit{Attribute} = \texttt{Color}$ or $\texttt{Weight}$, $\oplus \in \{<, >, \leq, \geq, =\}$ and $\mathit{value} \in \{\texttt{red}, \texttt{green}, \texttt{blue}, \ldots\} \cup I\!R$ (this corresponds to the so-called attribute value formalism). The cluster of apples, for instance, is extensionally described as

$$\mathit{Apples} = \{c, d, f, l, n\}$$

and intensionally as

$$\mathit{Apples} = \{x | \texttt{Color}(x) = \texttt{green} \wedge \texttt{Weight}(x) > 130g\}$$

Note that an equally valid intensional description of the cluster of apples would be, e.g., $\mathit{Apples} = \{x | \texttt{Color}(x) = \texttt{green} \wedge \texttt{Weight}(x) > 100g\}$. ◇

Predictive clustering

Many machine learning and data mining tasks involve the induction of a procedure to classify new instances or to predict unknown values for them. Prediction of a nominal variable is equivalent to *classification*; prediction of a continuous value is usually referred to as *regression*. For convenience, we will use the terms classification and regression both to refer to the induction of the predictor and its use (although originally, the term regression denotes the construction of the predictor rather than its use).

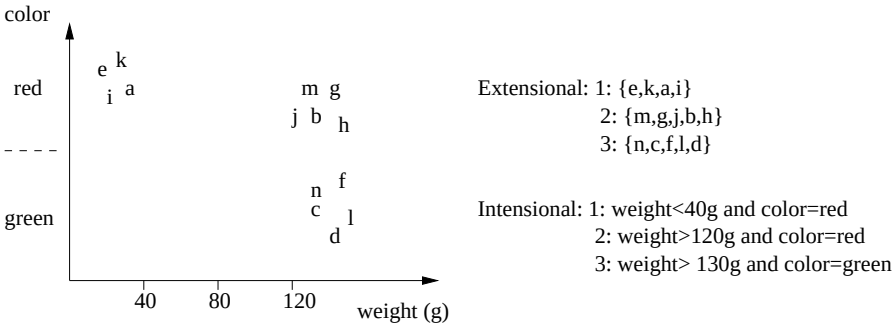


Figure 2.3: An example of clustering, in the context of fruit and vegetables.

Both classification and (certain types of) regression[3] can be seen as special cases of clustering. Indeed, many predictors are structured (implicitly) in such a way that an unseen instance is first assigned to a certain cluster, and the prototype of the cluster is then used to make a prediction.

To show this, we first define *predictive clustering* as a special case of clustering that is fit for prediction purposes. It is assumed that a function f exists that assigns unseen instances to existing clusters, and that the prototype of the assigned cluster is used as a prediction for the unseen instance; the quality criterion Q is maximal if predictions are as close as possible to the actual values, according to the distance measure d.

In the following we assume that a probability distribution over I exists. $\mathbf{E}$ denotes the expected value operator.

Definition 2.14 (Cluster assignment function) *A cluster assignment function is a function* $f : I \times \mathbf{C}(I) \rightarrow 2^I$ *such that*

$$\forall x \in I, \forall E \subseteq I : \forall \mathcal{C} \in \mathbf{C}(E) : f(x, \mathcal{C}) \in \mathcal{C} \tag{2.26}$$

I.e., given any clustering $\mathcal{C}$ *of a set of instances* E, f *assigns each possible instance in* I *to a cluster in* $\mathcal{C}$.

Definition 2.15 (Predictive clustering) *We define the task of predictive clustering as follows:*

Given:

- *an instance space* I

[3] Sometimes called *piece-wise* regression.

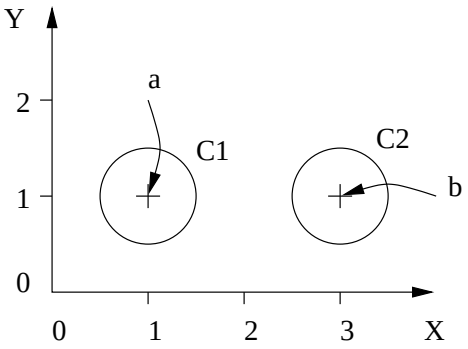


Figure 2.4: Predictive clustering.

- *a distance d over I*
- *a set of instances $E \subseteq I$*
- *a prototype function p*
- *a cluster assignment function f*

Find:

- *a clustering $\mathcal{C}$ over E that maximizes*

$$Q(\mathcal{C}) = -\mathbf{E}(d(x, p(f(x, \mathcal{C})))^2) \tag{2.27}$$

(where x ranges over I)

Note that, except for the minus, the right hand side of Equation 2.27 can be seen as a kind of variance. Maximizing Q corresponds to minimizing the expected intra-cluster variance in the clustering.

Example 2.6 Figure 2.4 illustrates our predictive clustering setting. We assume that $I = \mathbb{R}^2$ and $d(\mathbf{x}, \mathbf{y}) = d_E(\mathbf{x}, \mathbf{y})$. There are two clusters C_1 and C_2 with prototypes $p(C_1) = (1, 1)$ and $p(C_2) = (3, 1)$. Assume furthermore that we have a cluster assignment function that maps instances onto the cluster of which the prototype is closest:

$$f(x, \mathcal{C}) = C \Leftrightarrow \forall C' \in \mathcal{C} : d(x, p(C')) \geq d(x, p(C)).$$

Then $\mathbf{a} = (1, 2)$ would be mapped onto $(1, 1)$, and $\mathbf{b} = (4, 1)$ would be mapped onto $(3, 1)$. ◇

Maximizing Q intuitively means that E is clustered so that the prototype of a resulting cluster C_i is as representative as possible for the part of the instance space assigned to C_i. In other words, each example in I will be mapped to a prototype that is quite similar to it. The prototype can be seen as a prediction, or as a basis for prediction.

Classification and regression

In the context of classification and regression, the prediction usually is not a full instance, but a single attribute which we call the target attribute.

Definition 2.16 (Target function) *Given an instance space I and a prediction space P, a target function $\pi : I \to P$ is a function mapping instances onto their target value.*

For instance, in a classification setting, we call $\pi(x)$ the class of x.

Definition 2.17 (Predictor, prediction) *Given a cluster assignment function f, a prototype function p and a target function π, each clustering $\mathcal{C}$ defines a function $pred_{\mathcal{C}} : I \to P$:*

$$pred_{\mathcal{C}}(x) = \pi(p(f(x, \mathcal{C}))) \tag{2.28}$$

We call $pred_{\mathcal{C}}$ a predictor, and $pred_{\mathcal{C}}(x)$ its prediction for x.

We can now define classification and regression as special cases of predictive clustering:

Definition 2.18 (Classification) *Classification is a special case of predictive clustering where the range of π is nominal and the distance d between two instances is $d(x, y) = d_{=}(\pi(x), \pi(y))$.*

Most classification systems try to maximize predictive accuracy, which in this context is defined as follows:

Definition 2.19 (Predictive accuracy) *The predictive accuracy of a classifier pred is the probability that it makes a correct prediction, i.e.*

$$a = \mathbf{E}(1 - d_{=}(pred(x), \pi(x))) \tag{2.29}$$

It is easy to show that maximizing a indeed corresponds to maximizing Q for a distance $d(x, y) = d_{=}(\pi(x), \pi(y))$:

$$\begin{aligned} Q(\mathcal{C}) &= -\mathbf{E}(d(x, p(f(x, \mathcal{C})))^2) \\ &= -\mathbf{E}(d_{=}(\pi(x), pred_{\mathcal{C}}(x))^2) \\ &= -\mathbf{E}(d_{=}(\pi(x), pred_{\mathcal{C}}(x))) \qquad \text{(since } d \text{ is 0 or 1)} \\ &= a - 1 \end{aligned}$$

Definition 2.20 (Regression) *Regression is a special case of predictive clustering where the range of π is continuous and the distance d between two instances is $d(x,y) = d_E(\pi(x), \pi(y))$.*

Most regression systems minimize the mean squared prediction error

$$MSE = \mathbf{E}(d_E(\pi(x), pred_C(x))^2) \tag{2.30}$$

which is easily seen to be equal to $-Q(\mathcal{C})$.

The usefulness of this formalization may not be obvious right now, but in the remainder of this text the view that classification and regression are special types of predictive clustering will be adopted several times. On these occasions the above formalization will be further illustrated.

Limitations of this approach

We identify two limitations to our definition of classification and regression.

- The above definitions presume a specific quality criterion for predictions: accuracy in the case of classification, the least squares criterion in the case of regression. They thus do not encompass approaches that use other criteria. This could be changed by making the quality criterion in Definition 2.15 more general.
- It is assumed that within a cluster one specific value will always be predicted (the prototype); this excludes, e.g., linear piece-wise regression, , where in each cluster a linear prediction model should be stored (e.g., $Y = aX + b$, where a and b differ according the cluster). Our definition can be extended by making the prototype $p(C)$ a function that maps individual instances onto predictions, rewriting Equation 2.27 as

  $$Q(\mathcal{C}) = -\mathbf{E}(d(x, p(C)(x))^2) \tag{2.31}$$

  and Equation 2.28 as

  $$pred_C(x) = \pi(p(f(x,\mathcal{C}))(x)) \tag{2.32}$$

Example 2.7 Given a cluster of vectors $C = \{(1,3),(2,5),(3,7)\}$ the prototype of the cluster could be a function that maps an instance (x,y) (y is the target variable) onto $(x, 2x+1)$. The prediction for an instance $\mathbf{x} = (5,10)$ (assuming it has been assigned to C)[4] is then $\pi(p(C)(\mathbf{x})) = \pi((5,11)) = 11$. The distance $d(\mathbf{x}, p(C)(\mathbf{x})) = |10 - 11| = 1$. ◇

[4] We assume here that we know that $y = 10$ but the predictor does not; otherwise it need not really compute a prediction.

In this text we will not make use of these extensions, therefore we stick to the simple definitions as stated. These definitions already encompass a large subset of the many approaches to classification and regression.

A last remark: in practice, $d(x, p(f(x, \mathcal{C})))$ is unknown for unseen instances. Q is then usually estimated via

$$\hat{Q}(\mathcal{C}) = -\frac{\sum_{e \in E} d(e, p(f(e, \mathcal{C})))^2}{|E|} - compl(\mathcal{C}) \tag{2.33}$$

where E is the set of examples from which $\mathcal{C}$ was generated, and $compl(\mathcal{C})$ is a penalty that is higher for more complex cluster descriptions (without such a penalty, the optimal clustering would be the trivial clustering where each cluster consists of one element).

2.3.2 Dimensions of Clustering

Clustering systems can be distinguished along many dimensions. It is not our intention to give a complete overview of all these dimensions, nor to locate all the existing clustering systems along these dimensions. Rather, we discuss a few dimensions to give the reader just a flavor of the variety of clustering systems that exists, and to introduce some terminology that will be used later on in this text.

In this section we first briefly discuss *flat* versus *hierarchical* clustering. We next discuss *intensional* versus *extensional* clustering systems, then introduce the notion of *conceptual* clustering. We also look at clustering approaches from the point of view of the *flexibility* they have with respect to defining clusters. This dimension is not orthogonal to the others but strongly correlated with the intensional/extensional distinction. Finally we compare *predictive* and *descriptive* clustering.

Flat versus hierarchical clustering

As might be suspected from our problem definition (Definition 2.13), clustering systems can be divided into systems that find flat clusterings and systems that find hierarchical clusterings.

Hierarchical clustering algorithms can be (but are not necessarily) derived from flat clustering algorithms in a straightforward way: one simply repeats the flat clustering algorithm over and over again. The *divisive* approach works top-down: the data set is divided into large clusters, each of which is then divided into smaller clusters, and so on up to the level of single instances. Alternatively, the *agglomerative* approach works bottom-up: small clusters of examples are formed, then the clusters are joined into larger clusters, and so on until one single cluster is obtained.

The LEADER algorithm (Hartigan, 1975) is an example of a flat clustering algorithm. An example of an agglomerative clusterer is KBG (Bisson, 1992a). Divisive systems are RUMMAGE (mentioned in (Fisher and Langley, 1985)), DISCON (Langley and Sage, 1984), CLUSTER/2 (Michalski and Stepp, 1983), ... The incremental system COBWEB (Fisher, 1987) has a flavor of both, as it has operators both for combining and splitting clusters.

Intensional versus extensional clustering

We call a clustering system an intensional clustering system if it returns intensional descriptions of the clusters it finds; otherwise we call the system an extensional clustering system.

The LEADER algorithm is an example of an extensional clustering system. Intensional clusterers are KBG, RUMMAGE, DISCON, CLUSTER/2 and COBWEB.[5]

The descriptions returned by an intensional clusterer can be understood in two ways. They can be seen as *characterizing* the clusters, or as only *discriminating* between them. While characteristic descriptions should contain a maximum of information about a cluster, discriminant descriptions should contain a minimum of information (they should focus on what distinguishes a cluster from the other clusters).

Example 2.8 The description of apples in Figure 2.3 is "an apple is green and weighs more than 130g". If we want to answer the question "What are apples like?", a better description of apples that is still consistent with the picture is "an apple is green and weighs between 130g and 180g". It is better because it gives more information about apples. We call this a characteristic description.

However, if want to answer the question "How can I know if something is an apple?", the answer should allow to discriminate apples from strawberries and tomatoes with a minimum of information. In this case it suffices to look at the color: "if the object is green, it is an apple". This sentence is a discriminant description. ◇

Intensional clustering systems may return either discriminant descriptions, characteristic descriptions, or both. An approach that is sometimes followed is that an intensional clustering system first finds discriminant descriptions to define the clusters, then for each cluster calls a procedure for characterizing it.

In our example the language that is used for an intensional description of clusters is based on the same properties as the distance measure. (A cluster can be described by means of the weight and color of its elements, and we want to find clusters in which elements are similar with respect to weight and color.) In

[5] COBWEB returns probabilistic descriptions, which are intensional in a broader sense than defined by Definition 2.8.

general, this need not be the case. The intensional description language could make use of a strict subset of the properties used for the distance measure, could use a superset of it, or could even use a set that is totally disjoint from it. Alternatively, the language for discriminant descriptions may differ from the language for characteristic descriptions.

Example 2.9 In the Fruit&Vegetables example, one could add a property `Taste` to the description of each object and include it in the language for characteristic descriptions as well as in the distance measure, but not in the language for discriminant descriptions. Objects are then clustered according to their taste, but assigning an unseen instance to a cluster is always possible without knowing anything about its taste (by means of the discriminant descriptions). ◇

In general, one would typically use easily observable properties for a discriminant description of the clusters, but use less easily observable properties for the clustering process itself (in order to get a higher-quality clustering) and for the characteristic descriptions.

Conceptual clustering versus numerical taxonomy

The term *conceptual clustering* was introduced by Michalski and Stepp (1983). It refers to a general type of clustering where similarity between examples is computed relative to a background of concepts and other examples, and the clusters themselves should be defined in terms of (possibly other) concepts. Michalski and Stepp (1983) contrast this setting with numerical taxonomy (e.g., (Sneath and Sokal, 1973)), where the focus is on forming the clusters rather than describing them. This contrast emphasizes the difference between what we call intensional and extensional clustering in this text. Since we have already discussed that, we now focus on the computation of distances relative to a background of concepts and examples.

In this setting the similarity criterion is non-trivial; it can be complex and domain-dependent. It can even be the case that the similarity of two objects is not fully determined by the objects themselves, but also by the other objects in the set. Figure 2.5 illustrates this. The data are represented as dots in a two-dimensional space. Both in situation 1 and 2, there is a subset of data that forms a cross. The most obvious way of clustering the data depends on the context. In a context where straight lines are natural concepts (situation 2), the cross would be clustered into two straight lines; but in a context where hooks are natural concepts, it is more natural to cluster the cross into two hooks (situation 1).

While the context for the clustering would usually be given in advance, it can also be suggested by the data themselves, as is the case in these drawings.

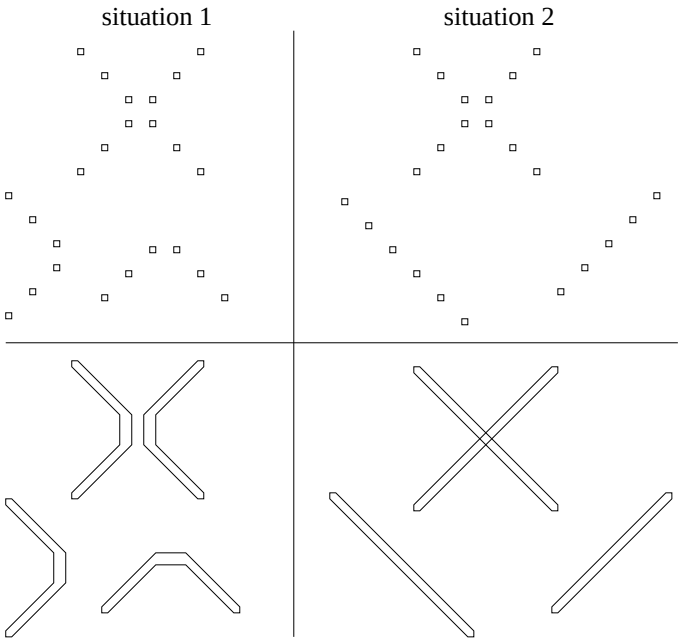


Figure 2.5: Conceptual clustering. The way in which the cross would typically be divided into two clusters depends on the context. The context itself is, in these drawings, suggested by the other points in the data set.

The abundance of hooks in situation 1 could make the clusterer prefer hooks as the typical cluster form, while in situation 2 it would prefer straight lines.

Note that the current view on clustering makes it necessary to adopt a non-trivial definition of similarity (at least if we stick with the assumption that a good cluster groups examples that are similar). In the context of Figure 2.5 two points are similar if one can superimpose a certain shape over them (a hook, or a straight line) such that the shape covers many other points. Thus, how similar two instances are does not depend solely on these instances, but also on the positions of the other instances.

We can thus distinguish three different contexts for conceptual clustering, in ascending order of difficulty:

1. The similarity between two instances is determined by the descriptions of these two instances themselves, and by a context consisting of a fixed set of concepts that are considered important.

2. The similarity between two instances is determined by all the instances in the data set (not only the two between which the similarity is computed), and by a fixed context.

3. The similarity is determined by all the instances in the data set, and by a context that is itself constructed from the data.

Figure 2.5 is an illustration of the third setting. A classic example that illustrates the same issue is shown in Figure 2.6. This is an experiment where human subjects are asked to classify the middle symbol on a card. Depending on whether people see the "ABC" card or the "12 13 14" card, they classify the middle symbol as a "B" or a "13", even though it is exactly the same symbol. One could say that the middle symbol is more similar to the letter "B" if it is surrounded by letters, and is more similar to the number 13 if it is surrounded by numbers.

A prototypical conceptual clusterer is Michalski and Stepp's CLUSTER/2 algorithm. The development of intensional clustering is strongly related to conceptual clustering, hence most intensional clusterers are also conceptual clusterers. Most clusterers work in context 1 or 2, however.

Flexibility with respect to cluster formation

We can distinguish the different approaches to clustering along a dimension that represents the flexibility that a system has in splitting a cluster into subclusters. Extensional clustering systems have no restrictions with respect to how a cluster can be split. Intensional clustering systems have restrictions according to the language of cluster descriptions they allow: weak restrictions if a large set of complex descriptions is allowed, strong restrictions if only a limited set of

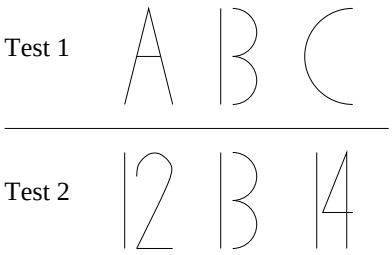


Figure 2.6: Do people think the symbol in the middle is more similar to the letter 'B', or to the number 13? It depends on the symbols surrounding it.

simple descriptions is allowed. In machine learning, this is usually referred to as the *language bias* of the system.

The stronger the restrictions on the language, the harder it is to define high quality clusters. Figure 2.7 shows how different kinds of clustering systems would proceed on an example data set. The data set contains three clearly distinguishable clusters, indicated by **1**, **2** and **3** on the drawing. An intensional clustering system can only use the attributes A, B, and C to describe the clusters. Each attribute can be 0 or 1. Straight lines indicate the boundaries in the example space between points with a value of 0 or 1 for a certain variable.

An extensional clustering system can define any set of examples to be a cluster, hence it should find the clusters without any problem. An intensional clustering system can only use the straight lines that are drawn to define clusters. By using combinations of variables (i.e. descriptions such as $A = 0 \wedge B = 0$), such a system can still identify the clusters correctly; but if it can only use single variables to describe subclusters it cannot find the correct clusters. In Figure 2.7 the third clustering method can only find two clusters in the data set (looking only at the highest level of the tree): one with $A = 0$ and one with $A = 1$.

Figure 2.7, while illustrating the problem, also suggests a solution: by allowing an intensional clustering system that can only use simple descriptions to build a hierarchy of clusters instead of a flat clustering, the problem is partially solved. On the second level of the tree, the clusters **1** and **2** are identified correctly. The cluster **3** however has not been found as a single cluster, but is divided into two subclusters.

We can actually look at the rightmost cluster tree in Figure 2.7 in two ways:

- We can view it as a cluster hierarchy. In this case a cluster is associated with each node. Some nodes define clusters of low quality; this is the case for the nodes on level 1 (directly below the root) and for the nodes

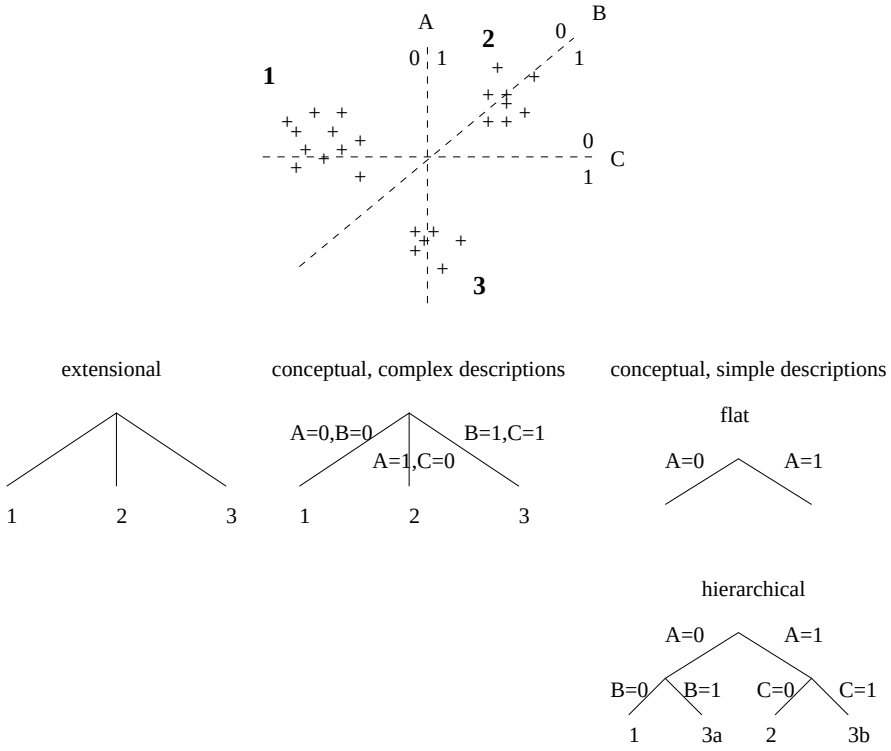


Figure 2.7: Different clustering methods.

labelled **3**. One could say that the hierarchy identifies "weak" intermediate clusters in its search for good clusters. These weak clusters may be less apparent from the data but are not necessarily meaningless.

- We can also choose to ignore the weak clusters in the tree, and extract a flat clustering from the tree by selecting a set of nodes S so that no node in S is an ancestor of an other node in S, and all nodes together cover all the examples. In Figure 2.7 we would then select the flat clustering {**1**, **2**, **3a**, **3b**}. Note that this clustering is still not optimal. Further post-processing could be performed to improve it, e.g., by rearranging clusters as is done in (Fisher, 1996).

  Several methods for extracting a flat clustering from a hierarchy have been proposed; see e.g. (Kirsten and Wrobel, 1998).

Intensional clustering systems that use complex descriptions of subclusters can yield clusters of higher quality, but this comes at the cost of a higher computational complexity. Indeed, the search space involved in finding complex descriptions is larger than the search space involved in finding single descriptions.

Another disadvantage of using complex descriptions is exactly the fact that they are complex. The idea behind intensional clustering is that clusters are good if they have simple intensional descriptions. There may be other criteria for the quality of clusters than just the simplicity of their description, but when the latter has a high weight there may not be a good reason for allowing complex tests.

Summarizing, we can say that using only simple descriptions is computationally more efficient than using complex descriptions, but may yield lower quality clusterings. This difference is more pronounced for flat clustering than for hierarchical clustering.

Figure 2.8 places several existing clustering systems on an axis indicating how flexible they are with respect to defining clusters: COBWEB (Fisher, 1987), CLUSTER/2 (Michalski and Stepp, 1983), LEADER (Hartigan, 1975), RUMMAGE (Fisher and Langley, 1985) and DISCON (Langley and Sage, 1984). Note that COBWEB, although returning intensional (probabilistic) descriptions for clusters, has the flexibility of an extensional system in forming the clusters: the clustering can be any partition of the example set.

Predictive versus descriptive clustering

One can see a clustering as purely descriptive, i.e. describing structure in the data; or one can see it as a means for making predictions. Whether a clustering is considered good or not depends on this. We illustrate this with Figure 2.9. Two flat clusterings are shown; the intensional cluster descriptions are equally

flexible		**less flexible**
extensional		intensional
	complex descriptions	simple descriptions
LEADER	CLUSTER/2	RUMMAGE
COBWEB		DISCON

Figure 2.8: Clustering systems plotted along the *flexibility* dimension.

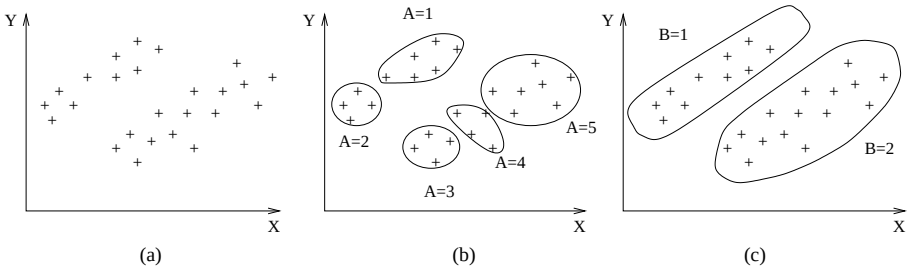


Figure 2.9: The difference between predictive and descriptive clustering. (a) original set of points; (b) a bad descriptive but good predictive clustering; (c) a good descriptive but bad predictive clustering.

simple in both cases. Assuming we want to use the clusters to predict values for X and Y, clustering (b) is reasonably good: the examples in each cluster differ little with respect to the variables X and Y. This means that X and Y can be predicted accurately if the cluster is known. We say that the *predictability* of the variables is high.

In clustering (c) the predictability of the variables is lower, because the clusters are spread out more. On the other hand, these clusters identify the existing structure in the data in a much better way: if one looks at plot (a), the clusters in (c) are apparent, while those in (b) are not. From a descriptive point of view, clustering (c) is to be preferred.

Note that in this case, a hierarchical clustering system could identify both the large and the smaller clusters. In this sense, hierarchical clusterers are good at combining both predictive and descriptive quality. Still, different clustering systems may be biased towards different quality criteria, and one should choose a clusterer accordingly.

2.3.3 Evaluation of Clusterings

The quality of a clustering can be measured in different ways. The main criterion for choosing a certain quality measurement should be the goal of the clustering: is it seen as purely descriptive (identifying clusters in a set of data) or is it to be used for prediction?

Descriptive Quality

Measuring the descriptive quality of a clustering is hard, and there are no agreed-upon criteria. One of the more popular criteria is the *partition utility*, defined in (Fisher, 1996) as

$$PU(\{C_1, \ldots, C_N\}) = \sum_k CU(C_k)/N \tag{2.34}$$

i.e., the average *category utility* of each cluster; the latter was introduced by Gluck and Corter (1985) and can be defined as

$$CU(C_k) = P(C_k) \sum_i \sum_j (P(A_i = V_{ij}|C_k)^2 - P(A_i = V_{ij})^2) \tag{2.35}$$

It is assumed here that each instance is described by a vector of attributes A_i, and the domain of each attribute A_i is a set of values V_{ij}. Category utility measures both the *predictability* of attribute values (how precise can the attribute of an instance be predicted if the instance's cluster is known), and their *predictiveness* (how well can the cluster be predicted if the attribute value is known). Note that this measurement assumes finite domains for all the attributes; moreover Fisher (1996) discusses some problems with it.

Predictive Quality

If a clustering is going to be used for prediction, the predictability of the attributes that are going to be predicted is the most important criterion. In some cases, such as (supervised or unsupervised) classification, one single nominal attribute is to be predicted. The accuracy with which the class can be predicted is usually the evaluation criterion then, see e.g. (Fisher, 1987). Note that this is a special case of the variance criterion we introduced in Definition 2.15.

In the regression context variance itself is a good relative quality criterion, but it is less fit as an absolute criterion; e.g., if the intra-cluster variance is 10, should we consider this high or low? A more popular criterion is therefore the *relative error*

$$RE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{2.36}$$

This criterion compares (on a test sample) the mean squared error of a predictor with the mean squared error of a default predictor that consistently predicts the global mean of the training sample (the y_i are the values observed in the test sample, the $\hat{y}_i$ are the corresponding predictions, and $\bar{y}$ is the global mean of the training sample).

The previous criteria assumed that one single variable is to be predicted. Another setting is when any value may need to be predicted in a new instance; this is sometimes called *flexible prediction* or *pattern completion*. The quality of a clustering should then be computed as an average over the predictability of each single attribute. Such an evaluation can be found in e.g. (Fisher, 1996). Unfortunately, when both continuous and discrete attributes are used, it is not obvious how such an average should be computed, as the predictability of discrete and continuous attributes is expressed in different ways.

Assuming that the distance between examples reflects the similarity of their attributes, the relative error criterion can be generalized to a quality criterion for flexible prediction:

$$RE = \frac{\sum_{i=1}^{n} d(e_i, \hat{e}_i)^2}{\sum_{i=1}^{n} d(e_i, p(Tr))^2} \tag{2.37}$$

where the e_i are the examples in a test set, $\hat{e}_i$ the corresponding predictions and $p(Tr)$ is the prototype of the training set. In the clustering context, it compares how far off the predictions are from the actual examples, compared to how far off the prototype of the training set is.

Note that if the predictor was constructed by a predictive clusterer by finding a clustering $\mathcal{C}$ of the training set Tr, then

$$RE = \frac{\sum_{i=1}^{n} d(e_i, p(f(e_i, \mathcal{C})))^2}{\sum_{i=1}^{n} d(e_i, p(Tr))^2} \tag{2.38}$$

which shows that by trying to maximize Q, a predictive clusterer tries to minimize the expected value of RE.

2.4 Using Clusterings for Prediction

2.4.1 Extensional Clusterings

Given an extensional clustering and a new instance (one that is not in the clustering), how can one predict unknown information for the instance?

This is usually done using a two-step process. In a first step, the instance is assigned to a cluster. This assignment is typically based on the distance of the instance to the clusters in the extensional clustering. The distance of the instance to a cluster could be chosen as the distance to the prototype of the

cluster, the average distance to all the elements of the cluster, the distance to the closest element of the cluster, etc. This choice determines the cluster assignment function f.

Once the instance has been assigned to a cluster C, the second step can be performed, which consists of making a prediction about the new instance, based on the information in C. The most obvious way to do this is to predict a value that is typical for the examples in C; i.e., predict the value that is observed in the cluster prototype: $\pi(p(C))$.

Both steps are represented in the following scheme:

Instance Based Learning

The above prediction method is related to instance-based learning (Aha *et al.*, 1991; Mitchell, 1997; Langley, 1996). An instance-based learner typically stores all the examples it has seen (or at least some of them), and when it sees a new instance, bases its prediction for it on the previously observed examples that are most similar to it. The k-nearest neighbor method, for instance, consists of gathering the k stored examples that are most similar to the new instance (k is a parameter of the method), and predicting the class value that occurs most often among these k examples (or the mean of the observed values, for a continuous target variable).

An interesting difference with the k-nearest neighbor method is that the latter does not form any clusters in advance; rather, at the time of prediction a cluster of k examples is improvised. The second step of the prediction is then similar to the one we propose. Note that our extensional clustering based prediction method coincides with 1-nearest neighbor if all the clusters consist of a single example that is also the prototype of the cluster.

2.4.2 Intensional Clusterings

Intensional clusterings can be used for predicting unseen properties of examples in a similar way as extensional clusterings. In Section 2.3 we distinguished discriminant and characteristic descriptions of clusters. Assuming a clustering method yields both discriminant and characteristic descriptions of the clusters it forms, the results of the clustering method can be used for making predictions in the following way:

- assign a new object to a cluster using the discriminant descriptions
- predict unknown properties of the object using the characteristic description of the cluster

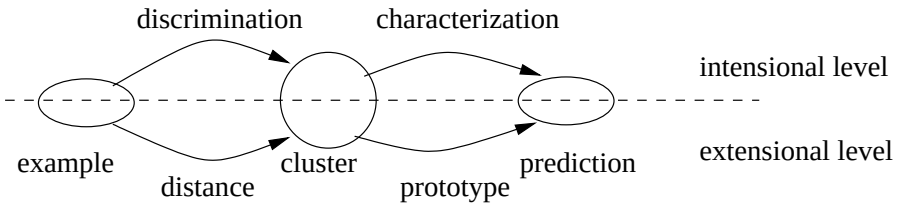
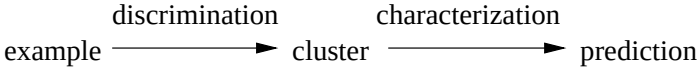


Figure 2.10: Clustering based prediction.

This is a two-step prediction process that is very similar to the one used with extensional clustering:



Note that prediction is to be understood in a very general sense here; one can not only predict specific values but also more general properties such as `Weight>130`, `if Color=green then Taste=sour`, etc.

2.4.3 Mixing Extensional and Intensional Reasoning

We have seen two-step prediction processes both at the extensional level and at the intensional level. Since the clusters themselves are the same on both levels, the two types of reasoning can be mixed, and we obtain the situation depicted in Figure 2.10.

This scheme indicates different reasoning mechanisms for making predictions: from an example description the cluster is predicted, either on the intensional or extensional level; then from the cluster one can infer a characteristic description of the example or predict a missing value.

Example 2.10 We illustrate the different types of reasoning on the Fruit & Vegetables example. Suppose we have the following intensional cluster descriptions:

	discriminant	characteristic
apples	`Weight>80g`	`Weight` $\in [130g - 180g]$
	$\wedge$`Color` $=$ `green`	$\wedge$`Color` $=$ `green` $\wedge$ `Taste` $=$ `sour`
tomatoes	`Weight>80g`	`Weight` $\in [120g - 180g]$
	$\wedge$`Color` $=$ `red`	$\wedge$`Color` $=$ `red` $\wedge$ `Taste` $=$ `sour`
strawberries	`Weight<80g`	`Weight` $\in [10g - 40g]$
		$\wedge$`Color` $=$ `red` $\wedge$ `Taste` $=$ `sweet`

Extensional cluster descriptions are sets of examples. We here consider the prototype to be part of the extensional description. We assume the prototype for apples is `(green, 160g, sour)`.

Some examples of the different reasoning mechanisms are (reasoning on the extensional/intensional level is indicated by $\Rightarrow_E$ and $\Rightarrow_I$):

1. `Color=green` $\Rightarrow_I$ instance is an apple $\Rightarrow_I$ `Taste` $=$ `sour` $\wedge$ `Weight` $\in [130g, 180g] \wedge$ `Color` $=$ `green`. "Is an apple" is an abbreviation for "belongs to the cluster of apples" here.

   This type of reasoning allows us to complete a partial description of an object.

2. `Color=green` $\Rightarrow_I$ the instance is an apple $\Rightarrow_E$ it must be similar to the prototypical apple `(green, 160g, sour)`.

   This type of reasoning can be used for predicting, e.g., the taste of an object by observing other properties of it (its color in this case).

3. An instance is most similar to some object in the "apple" cluster $\Rightarrow_E$ it is an apple $\Rightarrow_E$ it is similar to the prototypical apple `(green, 160g, sour)`.

   With this type of reasoning, we do not look merely at properties of the object itself but compare it with other objects, in order to predict its taste.

4. An instance is most similar to some object in the "apple" cluster $\Rightarrow_E$ it is an apple $\Rightarrow_I$ it conforms to the characteristic description of apples: `Weight` $\in [130g - 180g] \wedge$ `Color` $=$ `green` $\wedge$ `Taste` $=$ `sour`.

   We infer a more detailed description of the object by comparing it with other objects. E.g., the object looks like an apple, apples weigh between 130g and 180g, hence this object also has a weight between those boundaries.

$\diamond$

We remark that:

- a characteristic description of a cluster may contain elements that are not present in an explicit representation of any object. E.g., a prototype, being a representation of a single example, has one single weight. From a prototype one could not predict that an apple must have a weight between 130g and 180g. Hence, characteristic descriptions allow predictions of a more general kind than prototypes.

- while in the above example the information used in intensional descriptions is the same as that used in extensional descriptions, this need not be the case. Some parts of an example description may be fit for computing distances, others (e.g., properties described in a first order language, where distances are harder to define) may be more suitable for intensional descriptions.

- predictive induction systems (among which systems that induce rule sets or decision trees for classification or regression) usually induce hypotheses that involve reasoning of type (2). For instance, the normal way of using a regression tree is to sort an instance down the tree into a leaf (i.e. assigning it to a cluster via intensional reasoning) and predict the value stored in that leaf. A rule based system would first find a rule that applies to the instance (i.e. it assigns it to a cluster; the body of the rule is an intensional description of the cluster) and then make a prediction based on a (extensional) value stored in the head of the rule.[6]

The last remark illustrates that rule based systems and decision tree induction systems can be seen as predictive clusterers.

2.5 Creating Clusterings for Prediction

2.5.1 Clustering in Different Spaces

In the previous section we discussed how clusters can be used; we now focus on how they are created. We distinguish several variants of the clustering process, according to the directions along which the clusters should be coherent.

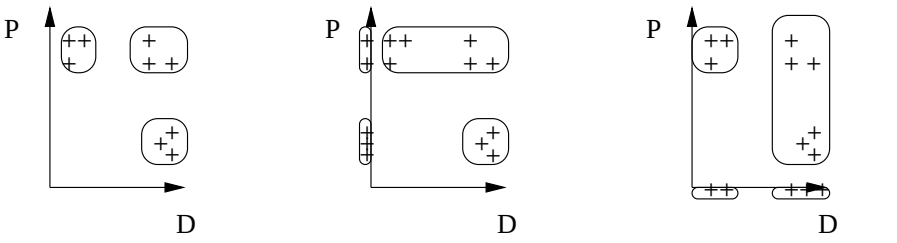
We distinguish those situations where the variables to be predicted are known beforehand (as is the case for, e.g., classification or regression) and those situations where they are not (flexible prediction). In the latter case, one simply creates clusters that are coherent in the instance space I (i.e., have low variance in I). In the former case however, we can divide the instance space I into a subspace D (the part of the instance that will be given to the predictor) and a subspace P (the part that will have to be predicted)[7]:

$$I = D \times P. \tag{2.39}$$

Instead of clustering in I, one can then also cluster in D or P. By "clustering in a subspace of I" we mean that the distance criterion for that subspace is

[6] We should mention that in the case of rule set induction, the clusters may overlap, which makes a more general definition of clustering than ours necessary. The approach essentially stays the same though.

[7] It is assumed here that except for the target variable, all the information available to the learner will also be available to the predictor.



(a) clustering in I (b) clustering in P (c) clustering in D

Figure 2.11: Clustering in the instance space and in its subspaces.

used when forming the clusters. Figure 2.11 gives an illustration. The D and P subspaces are represented as (one-dimensional) axes on this figure, although they may of course be of higher dimensionality. The projections of instances and clusters onto relevant axes is also shown.

We can then distinguish the following settings:

- form clusters in P: this is the classical **supervised learning** setting.

  The idea behind forming clusters in P is that, since predictions will be made for P only, it is sufficient that the instances within one cluster are close to one another in the P subspace; coherence in D is not important. Classification and regression, as we have defined them, cluster in P (the quality criterion is based on the distance in the prediction space P).

- form clusters in D: this is the classical **unsupervised learning** setting.

  When the clusterer does not have access to the values in P, the only option that is left is to cluster in D. Such unsupervised learning presupposes that examples close together in D will also be close together in P. If this assumption is false, good results cannot be obtained. Such a case is shown in Figure 2.11(c): the two rightmost clusters in I simply cannot be distinguished in D.

- form clusters in I.

  To our knowledge this setting is mainly used for descriptive clustering or flexible prediction, i.e. when P is not known beforehand.

If we compare the clusterings in Figure 2.11, clustering (a) is the only "good" clustering in I; the other methods identify overly general clusters. Clustering (b) is a good clustering in P, while (a) identifies overly specific clusters

(two clusters are formed that are really indistinguishable in P); similarly, (a) identifies overly specific clusters in D while (c) identifies the correct clusters.

Note that, from a descriptive point of view, it is as undesirable to find overly specific clusters as to find overly general clusters. However, from a predictive point of view finding overly specific clusters is not necessarily worse than finding the right clusters, while finding overly general clusters is. Hence, clustering in I might be an interesting alternative to clustering in P.

2.5.2 Do Learners Ignore Useful Information?

In the previous section, we have shown how supervised learners actually cluster in P instead of in I, reasoning that the resulting clusters should be coherent in P. In this section we look at this approach from an information-based point of view, i.e. what kinds of information are available to the learner and how well does it exploit the available information?

Given an N-dimensional space S, we define the information $\tilde{S}$ corresponding to S, as the set of dimensions of S. For instance, if $I = \mathit{Weight} \times \mathit{Color}$, then $\tilde{I} = \{\mathit{Weight}, \mathit{Color}\}$. Having the information $\tilde{S}$ available for an instance means that the coordinates of the instance in S are known.

In the previous section we assumed that the only information that the learner has in addition to what the predictor has, is the target variable. We now slightly extend the learning setting, in that the learner might have extra information $\tilde{L}$ available that the predictor cannot use and that need not be predicted either. Thus, instead of Equation 2.39 we have:

$$I = D \times P \times L \tag{2.40}$$

In terms of information, we get:

$$\tilde{I} = \tilde{D} \cup \tilde{P} \cup \tilde{L} \tag{2.41}$$

We define one last piece of information: $\tilde{H}$ is the information that the learner uses in its heuristics. We can then express two constraints that every learner must satisfy:

$$\tilde{H} \subseteq \tilde{I} \tag{2.42}$$

$$\tilde{D} \cap \tilde{P} = \emptyset \tag{2.43}$$

These constraints say that a learner cannot use heuristics based on unavailable information, and that we consider only non-trivial predictions.

For an **unsupervised** system it furthermore holds that

$$\tilde{H} \cap \tilde{P} = \emptyset \tag{2.44}$$

These are the only restrictions that apply for inductive learners. However, practical systems often impose extra constraints. For most classical supervised systems it holds that

$$\tilde{L} = \emptyset \quad (2.45)$$

$$\tilde{H} = \tilde{P} \quad (2.46)$$

where $\tilde{P} = \{Y\}$ with Y a discrete (classification) or continuous (regression) variable. I.e., the predictor is allowed to use all the knowledge available to the learner (except the value that is to be predicted), and the heuristics for building the predictor are based solely on the variable to be predicted.

For some applications it might be feasible to provide the learner with well-studied examples, about which more is known than what will be known of the instances for which predictions will have to be made, i.e. $\tilde{L} \neq \emptyset$.

Example 2.11 This example is based on the Mutagenesis dataset (see Appendix A for a description). Suppose one wants to find a theory that predicts the mutagenicity of molecules from their molecular structure. For the molecules that are provided to the learner, one could add information that is known to be relevant, e.g., the numerical features `lumo` and `logp`. By taking this extra information into account, it may be possible to obtain a better clustering. Still, as long as these extra features are not used in the intensional descriptions of the clusters, they need not be available for making predictions. ◇

Thus, our framework offers the possibility to use certain information (besides the target variable) for building a predictor, even if that information will not be available to the predictor. Most classical approaches to induction do not exploit such knowledge (see, e.g., the description of tree learners in Chapter 3).

Let us now take a look at Equation 2.46: $\tilde{H} = \tilde{P}$. To our knowledge this equation has been violated by very few, if any, predictive induction algorithms. The equation says that the search for a good predictive theory is guided by the variable to be predicted, and by that variable alone. For instance, tree induction algorithms use heuristics such as information gain or gain ratio (Quinlan, 1993a), Gini index (Breiman *et al.*, 1984) or mean squared prediction error (Kramer, 1996; Breiman *et al.*, 1984); rule set induction algorithms count the number of correct and incorrect predictions to guide their search (Clark and Niblett, 1989; De Raedt and Van Laer, 1995; Michalski *et al.*, 1986) or some other measure of predictive quality in the case of regression (Karalič and Bratko, 1997).

The reasoning behind this could be that the search for a high quality predictor can best be guided by computing the predictive quality of intermediate predictors; in other words, the quality of an intermediate predictor is the best indicator of the quality of the predictor it will ultimately lead to.

However, we see two counterarguments against this reasoning:

- It is not always true that the best predictor, according to a quality criterion Q, can best be found by using Q as heuristic. For instance, systems that build classification trees and try to maximize the accuracy of the final tree use information gain (Quinlan, 1993a) or Gini index (Breiman *et al.*, 1984) as heuristics. These are not equivalent to using the accuracy of the intermediate tree, in the sense that the heuristic may prefer one tree over another without its accuracy being higher. Still, these heuristics on average yield a final tree with higher accuracy than when accuracy itself is used as a heuristic. Breiman *et al.* (1984) discuss this phenomenon.

- While the heuristics just mentioned are different from accuracy, they are computed from the same information $\tilde{P}$, so Equation 2.46 still holds. One situation where this may not be an optimal approach, is when noise is present in the data. If the values for $\tilde{P}$ can be inaccurate, an algorithm using these values to guide its search might easily be misguided. Making use of other values in the search heuristic can make it more robust.

  In statistics it is known that when one has several independent estimators $\hat{\theta}_i$ for a certain parameter θ, each estimator being unbiased and having a standard error σ_i, the estimator

$$\hat{\theta} = \frac{\sum_{i=1}^{n} \frac{1}{\sigma_i^2} \hat{\theta}_i}{\sum_{i=1}^{n} \frac{1}{\sigma_i^2}} \tag{2.47}$$

  (a weighted average of the original estimators) is optimal in the sense that of all unbiased estimators that are a linear combination of the $\hat{\theta}_i$, $\hat{\theta}$ has the smallest standard error. While it is difficult to apply this result in a more general context where variables are not necessarily numerical and structural information can be taken into account, the general message remains. In the presence of noise, the observed class of an instance may not be the best estimator of its real class; instead, a better estimator can probably be formed by including other information.

In the context of unsupervised learning, it is common to let the heuristic use all the available information except $\tilde{P}$. The fact that such an approach works, suggests that even for supervised learning, information outside $\tilde{P}$ may be relevant. So, it turns out that learning with $\tilde{H} = \tilde{P}$ is only a special case of a more general supervised learning setting where $\tilde{P} \subseteq \tilde{H} \subseteq \tilde{I}$.

If we view the use of only $\tilde{P}$ in the heuristics as one extreme, and using everything except $\tilde{P}$ as the other extreme, it becomes clear that between these two extremes a broad range of other possibilities remains unexplored. Some of these are discussed in the next section.

2.5.3 Applications of Predictive Clustering

In addition to the classical approaches to classification, regression, unsupervised learning and flexible prediction, which are all special cases of predictive clustering, there are several other interesting applications that do not fall into one of the categories mentioned, but extend them in some way or combine properties of different categories.

Classification from Scarce Class Information

Assume one has to induce a classifier from a set of data where only a small percentage of the examples is labelled with a class. One way to handle this setting is to perform unsupervised learning, and once the clusters are formed (and not before) use the available class values to compute a prototype for each cluster. If the clusters are coherent with respect to classes, this method should yield relatively high classification accuracy with a minimum of class information available. This is quite similar in spirit to Emde's method for learning from few classified examples, implemented in the COLA system (Emde, 1994).[8]

In the context of the information-based approach we have just discussed, this technique can be explained by stating that $\tilde{H} \cap \tilde{P} = \emptyset$. More generally, one could devise an algorithm where $\tilde{H} \supset \tilde{P}$, i.e. the learner uses not only class information but also other information in its heuristics. A similar reasoning can be followed for regression.

Noise Handling

Our clustering based prediction technique is not only robust with respect to missing class information, but also with respect to noise in general. Discriminant descriptions are not very robust in this respect: if, e.g., in our Fruit&Vegetables example a red apple were encountered, following the discriminant descriptions of clusters it would certainly be misclassified, even if it has all the other characteristics of an apple. When discriminant descriptions can be supplemented with, e.g., characteristic descriptions or distance based criteria, a much more robust predictive system is obtained.

Combining Advantages of Different Approaches

There are many ways in which extensional and intensional reasoning could be combined. For instance, one could assign an object to a cluster according to

[8]The concept learner COLA first makes use of the first-order clusterer KBG-2 (Bisson, 1992a) to cluster the data, then defines the concept as a disjunction of some of these clusters, where preference is given to clusters that contain many positive and few negative examples (and, by the user's choice, many or few unclassified examples, biasing it towards more general or more specific concepts).

its discriminant description, check whether it fits the (more elaborate) characteristic description of the cluster, and if not, use an extensional method (e.g., k-nearest neighbor) to predict missing values. Alternatively, one could compute the distance of the example to the prototype, and if it is too far off, again resort to an extensional prediction method.

The possibility to combine extensional and intensional reasoning points out opportunities to join existing prediction techniques (such as instance based learning and rule set induction) into one hybrid system that combines the advantages of both separate techniques. The effect of this would not be limited to increased robustness with respect to noise, but could also lead to more efficient prediction methods or higher predictive accuracy even in noise-free domains. The latter claim is confirmed by some related work, which we discuss in the following section.

2.6 Related work

Part of this chapter is based on (Blockeel *et al.*, 1998b), but extends it in that the latter paper only discusses predictive clustering in the decision tree context, while the approach followed in this chapter makes abstraction of the representation formalism.

Our work is of course related to the many approaches to clustering that exist; within machine learning COBWEB (Fisher, 1987) and CLUSTER/2 (Michalski and Stepp, 1983) are probably the most influential ones. Although clustering is usually seen as a descriptive technique (it identifies structure in the data), the possibility of using it for predictions has been recognised since long. In fact, many clustering systems have been evaluated by measuring their potential to make accurate predictions, given the difficulty of assessing the descriptive quality of clusterings. Nevertheless, to our knowledge an explicit clustering based approach to classification and regression has never been discussed in detail.

Hybrid Approaches

Concerning hybrid approaches to predictive induction, our work bears an interesting relationship with Domingos' work on combining instance based learning with rule induction (Domingos, 1996; Domingos, 1998). In Domingos' approach, a rule induction system is used to induce rules from a data set; these rules are considered a generalized form of instances. Predictions are then made for new instances, based on their distance from the induced rules. The generalized instances have exactly the same functionality as the clusters (or more specifically, the prototypes) in our approach.

Another approach towards combining the use of distances for prediction with explicit induction is presented by Webb (1996). Webb, in contrast to

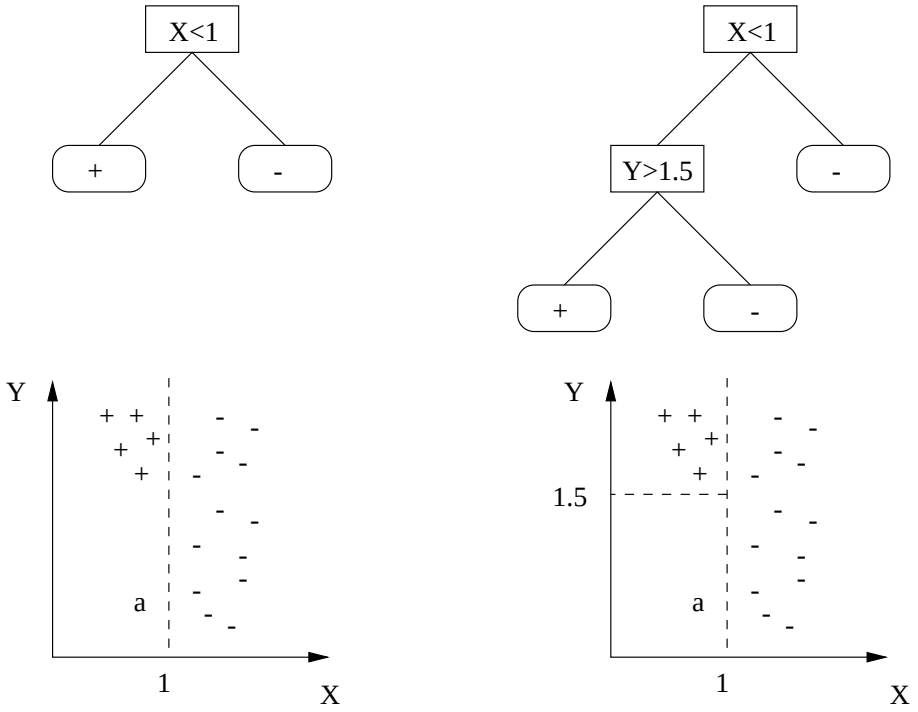


Figure 2.12: An example of an instance that has a high probability of being misclassified by a simple tree.

Domingos, focuses on decision trees instead of rule sets. He demonstrates that the predictive accuracy of a decision tree can significantly increase by making the tree more complex (instead of pruning it, as most decision tree induction systems do). The argument for splitting leaves into separate parts, even if those leaves are pure with respect to the class value, is based on distances. An example is given in Figure 2.12. The unseen example labelled (a) is in a leaf of positive instances, while all its closest neighbors are negative. Webb's approach handles such cases by constraining the area where an example will be predicted to be positive as much as possible, and to this aim keeps splitting nodes that with the standard decision tree approach would become leaves.

A third approach to combining distance based prediction and induction is followed by SRT (Kramer, 1996); this regression tree builder makes predictions using the classical sorting approach (i.e. it sorts an example down the tree into a leaf), but it also performs a check by computing the distance between the example and the leaf it belongs to. When the distance is suspiciously high, the example is reclassified by assigning it to the cluster that is closest to the

example, considering the tree as a hierarchy of clusters. The effect is very similar to that of Webb's approach. Kramer refers to this technique as *outlier detection*.

Domingos (1996), Webb (1996) and Kramer (1996) have all been able to show that mixing a distance based approach with explicit induction often leads to an increase in predictive accuracy. These results provide support for our thesis that a general approach can combine the advantages of more specific approaches.

2.7 Conclusions

In this chapter we have first introduced distances, prototypes and clustering. We have identified a special case of clustering that we called predictive clustering; it encompasses classification and regression but can also be used for flexible prediction and unsupervised learning. We have distinguished a purely extensional form of it, which was related to instance based learning; a purely intensional form; and a mixed form. The latter turns out to be the most interesting one, since it offers a broad range of possibilities; moreover it contains several classical approaches to predictive induction (rule set induction, decision tree induction) as special cases. These classical approaches turn out to occupy only the extremes of a spectrum provided by our framework. We have then argued that the area in between these extremes may provide interesting opportunities with respect to, e.g., noise handling, as well as for combining existing approaches. Some recent work with such hybrid approaches has been discussed in this context, providing support for the usefulness of a unifying approach.

Chapter 3

Top-down Induction of Decision Trees

3.1 Introduction

Decision trees are among the most popular tools for machine learning and data mining. They are most often used in the context of classification, and hence are often defined with this specific purpose in mind. Sometimes they are used for regression, in which case a slightly different definition is given. In both cases the tree is seen as simply a function mapping instances onto predictions. A few authors have pointed out, however, that a decision tree also represents a cluster hierarchy, and hence decision tree induction can be used in the clustering context.

In this chapter we study decision trees in the context of predictive clustering. We first show in Section 3.2 how decision trees represent the two steps in the prediction process we discussed in the previous chapter. Then, in Section 3.3, we discuss top-down induction of decision trees as a generic procedure that can be instantiated with parameter procedures. We show how some typical instantiations result in the classical TDIDT approaches to prediction.

This approach is reminiscent of the one in the previous chapter, where we showed that classification and regression can be seen as special cases of clustering. The difference is that the discussion in Chapter 2 made abstraction of the precise induction method; in this chapter we narrow the focus to one method (induction of decision trees) and study it in more detail.

In Section 3.4 we briefly compare decision trees with another popular representation formalism: rule sets. Section 3.5 discusses some related work and in Section 3.6 we conclude.

procedure SORT(T: tree, e : example) **returns** leaf:
    $N := T$
    while $N \neq$ **leaf**(C) **do**
        let $N =$ **inode**$(\tau, \{(r, t_r) | r \in range(\tau)\})$
        $N := t_{\tau(e)}$
    return N

procedure TARGET(N : leaf) **returns** prediction:
    let $N =$ **leaf**(C)
    return $p'(C)$

procedure PREDICT(T: tree, e: example) **returns** prediction:
    return TARGET(SORT(T, e))

Figure 3.1: Making a prediction for an example using a decision tree.

3.2 Decision Trees

Definition 3.1 (Decision tree) *A decision tree for a domain I is defined recursively as follows:*

- **leaf**(C) *is a decision tree for I (C can be anything)*
- **inode**(τ, S) *is a decision tree for I if and only if τ is a function from I to some set R (we call τ a test) and S is a set of couples such that $\forall r \in R : (r, t_r) \in S$ and t_r is a decision tree for I*

I.e., each node in a decision tree contains a test and associates a subtree with each possible outcome of the test.

A decision tree maps an instance e onto a leaf in the following manner: starting with the root node r, one subjects the instance e to the test τ_r and selects the child node associated with $\tau_r(e)$. This procedure is repeated for the new node until a leaf node is reached. We say that the instance has then been sorted into that leaf. The SORT algorithm in Figure 3.1 implements this procedure. In this algorithm a leaf is represented as **leaf**(C) with C an extensional representation of the cluster. Note that if we consider leaves as clusters, SORT is in fact a cluster assignment function f.

Through this sorting procedure, each decision tree defines a function $t : I \to \mathcal{C}$, where I is the instance space and $\mathcal{C}$ is the set of all possible leaves. We can consider t a specialization of the cluster assignment function f for $\mathcal{C}$, i.e. $t(e) = f(e, \mathcal{C})$.

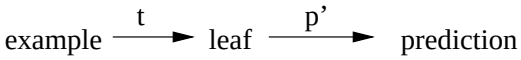


Figure 3.2: Two steps in the mapping from instances to predictions.

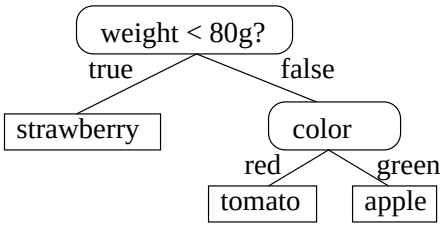


Figure 3.3: A fruit classifier in decision tree format.

For convenience we define

$$p' : \mathcal{C} \to P : p'(C) = \pi(p(C)) \tag{3.1}$$

Figure 3.2 then illustrates how t and p' implement the two-step prediction process we discussed in the previous chapter. An algorithm is given in Figure 3.1; it has the same structure. In practical implementations, however, the TARGET function is often made implicit, i.e. $p'(C)$ is stored in a leaf instead of C. Alternatively, $p(C)$ could be stored, where $p(C)$ could be either a prototype as we have defined it, or a generalized prototype (so that, e.g., trees with linear models in the leaves are also allowed).

Note that t makes use of the tests in the tree, which form an intensional description of the clusters (see further), while p' makes use of the prototype, which is part of the extensional description. The process is thus an instantiation of the "*example* $\Rightarrow_I$ *cluster* $\Rightarrow_E$ *prediction*" type of reasoning.

We can distinguish several kinds of decision trees. A *classification tree* is a decision tree where P is nominal; such a tree is shown in Figure 3.3. A *regression tree* is a decision tree where P is numerical. One could call a decision tree a *flexible prediction tree* if $p' = p$.

A decision tree is very similar to a cluster hierarchy: all the nodes on one level define a partition of the examples, and the children of a node define a partition of the examples covered by that node. Each cluster has a discriminant description of the form

$$\{e | \bigwedge_i \tau_i(e) = r_i\}$$

with i ranging over all the ancestor nodes of the cluster and r_i the result of $\tau_i(e)$ for all e in the cluster. Thus, the induction of a decision tree can be seen as a descriptive hierarchical clustering method, returning discriminant descriptions of the clusters. Such a tree could be called a *clustering tree*.

Example 3.1 For the tree in Figure 3.3, the following table shows the clusters in the tree and the corresponding discriminant descriptions:

cluster	description
strawberries	`Weight`$<$`80g` $=$ `true`
tomatoes+apples	`Weight`$<$`80g` $=$ `false`
tomatoes	`Weight`$<$`80g` $=$ `false` $\wedge$ `Color` $=$ `red`
apples	`Weight`$<$`80g` $=$ `false` $\wedge$ `Color` $=$ `green`

In practice, an expression such as `Weight`$<$`80g` $=$ `false` is of course abbreviated to `Weight`$\geq$`80g`. $\diamond$

We want to stress here that the quality criterion for descriptive clusterings is in a sense more severe than for predictive clusterings. For a predictive clustering it suffices that the intra-cluster variance of the leaf clusters is small. If a tree is considered as a descriptive cluster hierarchy, the quality of the clusters defined by internal nodes is also important. In the previous chapter we discussed flexibility of clustering, and noted that by allowing complex cluster descriptions one may obtain clusterings of higher quality. In the context of decision trees this translates to allowing complex tests τ_i in the internal nodes of a tree.

3.3 Induction of Decision Trees

Most systems that induce classification or regression trees construct the tree from the root to the leaves. This method is usually referred to as *Top-Down Induction of Decision Trees* (TDIDT) (Quinlan, 1986), or as the *divide-and-conquer* approach. Both incremental and non-incremental approaches exist. In this text we only consider non-incremental approaches.

A generic non-incremental TDIDT-algorithm is shown in Figure 3.4. We use the convention that procedures that are in fact parameters of the algorithm are written in normal capitals (e.g. OPTIMAL_SPLIT), while real procedure names are written in small capitals (e.g. TDIDT). Our TDIDT algorithm[1] consists of two phases, a growing phase and a pruning phase. The tree growing algorithm works as follows: given a set of examples E, it constructs some set of tests $\mathcal{T}$ (GENERATE_TESTS). This set usually depends on the form of the

[1] TDIDT is used as an abbreviation of Top-Down Induction of Decision Trees, while TDIDT refers to the specific algorithm in Figure 3.4.

example descriptions, but may also depend on specific values occurring in the data. For instance, if example descriptions contain an attribute `Color` that takes only the values `red`, `blue` in the observed data, then the tests `Color=red` and `Color=blue` are in $\mathcal{T}$.

Each test $\tau \in \mathcal{T}$, being a function from the instance space to a finite result space, induces a partition $\mathcal{E}$ on E that can be defined as follows:

$$\mathcal{E} = \{E_j | E_j = \{e \in E | \tau(e) = r_j\}\}. \tag{3.2}$$

The algorithm calls the function OPTIMAL_SPLIT to find that test $\tau \in \mathcal{T}$ that partitions E in some optimal way.

The algorithm next calls a function STOP_CRIT to check whether the optimal partition $\mathcal{E}$ that was found is sufficiently good to justify the creation of a subtree. If it is not, a leaf is constructed containing some relevant information about E (this relevant information is computed by the function INFO). If $\mathcal{E}$ is sufficiently good, then GROW_TREE is called recursively on all the $E_j \in \mathcal{E}$, and the returned trees t_j become subtrees of the current node.

In many systems the GROW_TREE procedure grows an overly large tree that may overfit the training data. Therefore, after the tree-growing phase these systems have a post-pruning phase in which branches are pruned from the tree, in the hope to obtain a better tree. The pruning algorithm PRUNE performs this.

The functions OPTIMAL_SPLIT, STOP_CRIT, INFO and PRUNE are parameters of TDIDT that will be instantiated according to the specific task that is at hand (e.g., classification requires different functions than regression)[2]. Note that the choice of OPTIMAL_SPLIT, STOP_CRIT and PRUNE determines the function t that maps examples to leaves, whereas the INFO function determines whether C, $p(C)$, $p'(C)$ or possibly something else is stored.

Most (classification or regression) tree induction systems that exist today are instantiations of this generic algorithm: we mention C4.5 (Quinlan, 1993a), CART (Breiman *et al.*, 1984), STRUCT (Watanabe and Rendell, 1991), SRT (Kramer, 1996), ... The main exceptions are incremental decision tree learners (Utgoff, 1989; Chapman and Kaelbling, 1991). These systems typically build the tree top-down, but contain operators for changing the tree when new evidence suggests to do so (by changing tests, rearranging nodes, extending leaves or collapsing internal nodes into leaves).

We now discuss possible instantiations of the generic algorithm in Figure 3.4 in the context of classification, regression and clustering. Table 3.1 gives an overview of these instantiations, stressing the similarities between the different algorithms.

[2]The function GENERATE_TESTS is also a parameter, but is irrelevant to the discussion in this chapter.

function GROW_TREE(E: set of examples) **returns** decision tree:
    $\mathcal{T}$:= GENERATE_TESTS(E)
    τ := OPTIMAL_SPLIT($\mathcal{T}$, E)
    $\mathcal{E}$:= partition induced on E by τ
    if STOP_CRIT(E, $\mathcal{E}$)
    then return leaf(INFO(E))
    else
        for all E_j **in** $\mathcal{E}$:
            t_j := GROW_TREE(E_j)
        return inode(τ, $\{(j, t_j)\}$)

function TDIDT(E: set of examples) **returns** decision tree:
    T' := GROW_TREE(E)
    T := PRUNE(T')
    return T

Figure 3.4: The TDIDT algorithm.

	OPTIMAL_SPLIT	STOP_CRIT	INFO	PRUNE
classification	gain (ratio) Gini index	χ^2-test min. coverage MDL	mode	C4.5 valid. set
regression	intra-cluster variance of target variable	F-test, t-test min. coverage MDL	mean	valid. set
clustering	intra-cluster variance	F-test, t-test min. coverage MDL	prototype identity	valid. set

Table 3.1: Overview of the different tasks that can be performed with TDIDT by instantiating its procedure parameters.

3.3.1 Splitting Heuristics

Given a set of tests $\mathcal{T}$, the function OPTIMAL_SPLIT computes for each $\tau \in \mathcal{T}$ the partition induced by τ on the set of examples E. It evaluates these partitions and chooses the test τ that is optimal with respect to the task that is to be performed.

Classification

For classification, many quality criteria have been proposed. We mention a few of them; for each one it is the case that a split is considered optimal if it maximizes the criterion.

- *Information gain* (Quinlan, 1993a): the *class entropy* of a set of examples E is defined as

$$s(E) = \sum_{i=1}^{k} p(c_i, E) \log p(c_i, E) \tag{3.3}$$

  where k is the number of classes, the c_i are the classes and $p(c_i, E)$ is the proportion of the examples in E that belong to class c_i. The information gained by performing a test τ is

$$G = s(E) - \sum_{E_i \in \mathcal{E}} \frac{|E_i|}{|E|} s(E_i) \tag{3.4}$$

  where $\mathcal{E}$ is the partition on E induced by τ.

- *Information gain ratio* (Quinlan, 1993a): the information gain obtained with a test is compared to the maximal gain that can be offered by any test τ' for which the cardinality of the induced partition and of the elements of this partition are the same as for τ.

$$MG = \sum_{E_i \in \mathcal{E}} p_i \log p_i \tag{3.5}$$

  with $p_i = |E_i|/|E|$. The gainratio is the ratio of the gain and this maximal gain:

$$GR = G/MG \tag{3.6}$$

- the *Gini heuristic* (Breiman *et al.*, 1984): this is similar to information gain, but instead of class entropy, the Gini index for impurity is used:

$$g(E) = \sum_{i=1}^{k} p(c_i, E)(1 - p(c_i, E)) = 1 - \sum_{i=1}^{k} p(c_i, E)^2 \tag{3.7}$$

The quality of a split is computed as

$$Q = g(E) - \sum_{E_i \in \mathcal{E}} \frac{|E_i|}{|E|} g(E_i) \tag{3.8}$$

These criteria seem to be the most popular ones for induction of classification trees. Empirical comparisons between different criteria can be found in (Mingers, 1989; Buntine and Niblett, 1992); (Breiman *et al.*, 1984) and (Breiman, 1996) contain some more theoretical discussions. An interesting observation is that none of these heuristics are directly related to the intra-cluster variance in the partition induced by the test (as also remarked in Section 2.5.2).

Regression

Regression systems typically use as quality criterion the intra-subset (or within-subset) variation of the target variable (see e.g. CART (Breiman *et al.*, 1984), SRT (Kramer, 1996)):

$$SS_W = \sum_{E_i \in \mathcal{E}} \sum_j (y_{ij} - \bar{y}_i)^2 \tag{3.9}$$

where y_{ij} denotes the j-th observation of the target variable in the set E_i, $\bar{y}_i = \sum_j y_{ij}/|E_i|$ and $\bar{y} = \sum_{i,j} y_{ij}/|E|$. This quality criterion should be minimized, not maximized.

From the statistical technique known as analysis of variance (ANOVA), it is known that if a set of values for a variable is partitioned into subsets, the total variation of the variable (measured as the sum of squares of differences between the values and the mean) can be decomposed into a within-subset and a between-subset variation, as follows:

$$SS_T = SS_B + SS_W \tag{3.10}$$

with

$$SS_T = \sum_{i,j} (y_{ij} - \bar{y})^2 \tag{3.11}$$

$$SS_B = \sum_i n_i (\bar{y}_i - \bar{y})^2 \tag{3.12}$$

where n_i is the number of elements belonging to subset i, SS_T stands for total variation, SS_B is between-subset variation and SS_W is within-subset variation. Equation 3.10 shows that minimizing SS_W is equivalent to maximizing SS_B. In the case where only two subgroups are formed, maximizing SS_B is equivalent to maximizing $|\bar{y}_1 - \bar{y}_2|$.

Clustering

Since the heuristic for regression minimizes variance, a generalization towards predictive clustering is straightforward:

$$SS_T = \sum_{i,j} d(e_{ij}, p(E))^2 \tag{3.13}$$

$$SS_B = \sum_{i} n_i d(p(E_i), p(E))^2 \tag{3.14}$$

$$SS_W = \sum_{i} \sum_{j} d(e_{ij}, p(E_i))^2 \tag{3.15}$$

If $d = d_E$, it still holds that $SS_T = SS_B + SS_W$, but this does not hold for every distance. A clustering system then has to choose between maximizing SS_B and minimizing SS_W. If only two clusters E_1 and E_2 are formed, it could also maximize $d(p(E_1), p(E_2))$.

3.3.2 Stopping Criteria

Many different stopping criteria have been proposed in the literature. Some very **simple criteria** are:

- stop splitting when a cluster is sufficiently coherent (i.e. its variance is below an acceptable threshold)
- stop splitting when the number of examples covered by a node is below some threshold

A more complicated criterion is the **MDL principle** (Rissanen, 1978). MDL stands for *Minimal Description Length*. The reasoning behind such a criterion is that the correct target values of a set of examples can be encoded in the form of a hypothesis, together with a list of corrections: an exhaustive enumeration of all the values that are predicted incorrectly by the hypothesis. When comparing two hypotheses with different predictive quality and different complexity, one should prefer the one with minimal description length; in other words: only make a hypothesis more complex if the gain in predictive quality is sufficiently large to justify it.

Hence, MDL is some sort of exchange rate that is applied when trading simplicity for accuracy. While the method has theoretical foundations, it is still relatively *ad hoc* in the sense that the best theory is not necessarily the most compact one. Moreover, applying it outside the classification context is quite complicated. Kramer (1996) discusses how it can be applied for regression. This version of MDL could in principle be generalized to the clustering context.

Another family of stopping criteria is based on **significance tests**. In the classification context a χ^2-test is often used to check whether the class

distributions in the subtrees differ significantly (the same test is used by some rule-based systems, e.g. CN2 (Clark and Niblett, 1989) and ICL (De Raedt and Van Laer, 1995)). A χ^2-test was also incorporated in Quinlan's ID3 algorithm (Quinlan, 1986), the predecessor of C4.5. It was not incorporated in C4.5 because, as Quinlan notes (Quinlan, 1993a), the test is relatively untrustworthy and better results are usually obtained by not using any significance test but pruning the tree afterwards. A similar argument is given by Breiman *et al.* (1984).

Significance tests do have the advantage that one can stop growing a tree relatively early, instead of growing a large tree and pruning away most of its branches; thus, an important gain in efficiency is achieved. For this reason, they are still incorporated in many systems.

Since regression and clustering use variance as a heuristic for choosing the best split, a reasonable heuristic for the stopping criterion seems to be the F-test. If a set of examples is split into two subsets, the variance should decrease significantly, i.e.,

$$F = \frac{SS_T/(n-1)}{SS_W/(n-k)} \tag{3.16}$$

should be significantly large (where SS_T and SS_W are defined by Equations 3.13 and 3.15, k is the number of subsets in the partition and n is the number of examples in the whole set). When $k = 2$ the F-test is equivalent to a t-test on the prototypes, which could be used instead.

3.3.3 Information in Leaves

The INFO function computes the information that is to be stored in a leaf. Typically, one stores only that information that will be necessary for prediction, i.e. the value that is to be predicted:

- for classification trees: store the mode of the class values observed in the leaf
- for regression trees: store the mean of the values of the target variable that are observed in the leaf
- for clustering trees: store an extensional cluster representation C, or the prototype $p(C)$

Storing a prototype or extensional representation makes flexible prediction possible, but of course it can also be done for classification or regression trees.

3.3.4 Post-pruning

Because it is hard to find good stopping criteria, many tree building systems grow an oversize tree and afterwards prune away those branches that do not seem useful. This procedure is computationally more expensive, but is known to yield better trees (see e.g. (Breiman *et al.*, 1984; Quinlan, 1993a)).

Several methods for post-pruning are in use. C4.5 uses an estimate of the tree accuracy on unseen data; this estimate is based on the errors in the training set as well as the complexity of the tree.[3] The method lacks a firm statistical ground, but seems to work reasonably well in practice. It only works for classification trees, however (although it could possibly be generalized towards regression or clustering).

Another way to estimate the accuracy on unseen data, is to simply remove a random sample from the training set, and use it as unseen data to evaluate the tree afterwards. The set that is removed in this way is called a *validation set*, hence we call this pruning method *validation set based pruning*. The idea is that through consecutive pruning, a series of subtrees of the original tree is generated; the subtree that has the highest quality on the validation set is chosen as the final tree. This method is described by Breiman *et al.* (1984), who also propose a more sophisticated algorithm that uses the same underlying idea. In their approach, pruning is controlled by a parameter α that represents the marginal cost of having an extra leaf in the tree. The optimal value for α is estimated using a cross-validation: several auxiliary trees are grown, each with a different validation set, and α is chosen so that the average quality of each auxiliary tree on its own validation set is maximized. This α is then used to prune a tree that has been grown on the whole data set.

Advantages of this method are that it can be used for any kind of trees, as long as there is some notion of the quality of a tree and a way to compute it (see Section 2.3.3), and that it is a statistically sound technique to estimate the quality of a tree on unseen data. A disadvantage of the simple validation set based pruning method is that the training sample becomes smaller, which may affect the quality of the tree; this disadvantage is avoided with Breiman *et al.* (1984)'s cross-validation-based algorithm.

3.3.5 Summary

Looking back at Table 3.1, we can conclude that the currently existing approaches to the induction of classification or regression trees are all very similar from the point of view of building predictive clustering trees:

[3] Using the data in the leaf, a confidence interval for the predictive accuracy of the leaf is constructed; the accuracy of the leaf is estimated as the lower bound of this interval, following the reasoning that the training set accuracy in the leaf is probably too optimistic. Smaller leaves yield larger intervals, hence more pessimistic estimates.

- The OPTIMAL_SPLIT procedure basically minimizes the intra-cluster variance of the partition induced by a test; exceptions are mainly found with classification, where variance based heuristics work less well than specialized heuristics based on information gain (ratio) or Gini index.
- The STOP_CRIT procedure for regression is similar to that for clustering, but for classification the significance test and the implementation of the MDL technique differ.
- Given that modes and means are special cases of prototypes, the INFO procedure is essentially the same in all cases.
- The validation set based post-pruning method can be used in all cases, although at least one classification system (C4.5) contains a specific method that does not immediately generalize to the other settings.

3.4 Trees versus Rules

There are two very popular representation formalisms within symbolic machine learning; decision trees form the first one, a second one is rule sets. We now briefly discuss induction of rule sets and compare it with induction of decision trees. This section provides some motivation for the use of decision trees in this text, but is not mandatory for reading the remainder of this text.

3.4.1 Induction of Rule sets

A hypothesis can be represented by a set of rules, where each rule is in the following format:

`IF` *condition* `AND` *condition* `AND ... THEN` *conclusion*

Most predictive systems learn rules where the conclusion of the rule indicates the class. An example of what a typical rule set would look like for the Fruit&Vegetables example is given in Figure 3.5. Both attribute-value rules and a Prolog program are given. Indeed, Prolog programs are basically rule sets, and since practically all ILP systems learn logic programs or Prolog programs, most of them (though not all, see e.g. (Boström, 1995)) follow the covering approach.

Sometimes the rules in a rule set are supposed to be ordered: a rule is only applicable when none of the preceding rules are applicable. In that case the rule set is called a *decision list*. Decision lists have the advantage that it is more easy to define a concept that has exceptions: the rule describing the exceptions is then written before the more general rule. When no ordering of

A rule set in the attribute-value framework:

```
IF Weight < 80g THEN Class = strawberry
IF Color = red AND Weight >= 80g THEN Class = tomato
IF Color = green THEN Class = apple
```

The same rule set written as a Prolog program:

```
strawberry(X) :- weight(X, W), W<80.
apple(X) :- color(X, green).
tomato(X) :- color(X, red), weight(X, W), W>=80.
```

Figure 3.5: Examples of rule sets in the Fruit&Vegetables example.

rules is present, the general rule often has to be made more complex to exclude the exceptions.

Decision lists can be represented easily in Prolog by putting a cut at the end of each program clause. The semantics of propositional rule sets (whether they form a decision list or just a set of unordered rules) are usually defined by the system that induces them. (One could change the IF-THEN format to an IF-THEN-ELSE format to make the ordering of the rules more explicit.)

Most algorithms for the induction of rule sets are variants of the algorithm shown in Figure 3.6. The task is to learn rules of the form `IF` *conditions* `THEN` *positive*, i.e. the rule set as a whole should make a positive prediction for those instances that are indeed positive.

If all the conditions in a rule are true for a specific example, we say that the rule *covers* the example. Basically, the approach is that one rule at a time is learned. Rules are preferred that cover as many positive examples as possible, and as few negatives as possible (since these two criteria may conflict, weights have to be assigned to both according to their importance; the exact weights may vary in different instantiations of the algorithm). Each new rule should at least cover some previously uncovered instances. This is continued until no more uncovered positive instances exist, or until no additional rules can be found.

The variant shown in Figure 3.6 grows individual rules top-down:[4] it always starts with a general rule that covers all examples and keeps adding conditions until the rule does not cover any negative examples (i.e. not covering negative examples gets an infinitely high weight here). The procedure OPTIMAL_CONDITION selects a condition that removes as many negative examples from the covering of R and keeps as many positive examples in it as possible. This procedure uses a heuristic which may vary along different

[4]Bottom-up approaches to rule growing also exist.

```
procedure INDUCE_RULESET(E+, E-: set of examples) returns set of rules:
    H := ∅
    U := E+
    while U ≠ ∅ do
        R := GROW_RULE(U, E-)
        H := H ∪ {R}
        U := U − {e|e is covered by R}
    return H

procedure GROW_RULE(E+, E-: set of examples) returns rule:
    R := IF true THEN Class=pos
    while R covers some negative examples
        T := GENERATE_TESTS(E+, E-)
        C := OPTIMAL_CONDITION(R, T, E+, E-)
        add C to the condition part of R
    return R
```

Figure 3.6: The covering algorithm, also known as separate-and-conquer.

implementations.

The approach of learning rules one at a time is usually referred to as the *separate-and-conquer* approach, or as the *covering approach*. The latter name comes from the fact that it focuses on uncovered examples. Once a set of positive examples has been covered by a rule, it is removed from the set on which the algorithm focuses.

The covering algorithm as given here only induces unordered rule sets. Induction of decision lists takes a more complicated approach. First of all, rules predicting different classes must be learned in an interleaved fashion, because for a set of rules predicting the same class the order of the rules can never be important. A different algorithm is followed according to whether rules are learned from first to last in the list, or in the opposite direction. In the latter case, the strong bias towards keeping negative examples uncovered should be removed, because when a rule covers negatives this may be taken care of by new rules that are still to be found.

Examples of rule-based systems are CN2 (Clark and Niblett, 1989), the AQ series of programs (Michalski *et al.*, 1986) and almost all ILP systems : PROGOL(Muggleton, 1995), FOIL(Quinlan, 1993b), ICL(De Raedt and Van Laer, 1995), ... The ILP systems FOIDL(Mooney and Califf, 1995) and FFOIL(Quinlan, 1996) learn first order decision lists.

3.4.2 A Comparison Between Trees and Rules

Comparing the tree in Figure 3.3 with the rule sets in Figure 3.5 reveals a strong correspondence: they have a similar structure and use the same tests. Not surprisingly, in the attribute value framework trees can be always be converted to rule sets using a very simple algorithm that is described in e.g. (Quinlan, 1993a)[5]. Basically, for each leaf one just collects the tests on the path from the root to the leaf and constructs a rule from these. For the decision tree in Figure 3.3 this yields the following rule set:

```
IF Weight < 80g THEN Class = strawberry
IF Weight >= 80g AND Color = red THEN Class = tomato
IF Weight >= 80g AND Color = green THEN Class = apple
```

Typically a rule set that is derived in this straightforward manner from a tree contains redundant tests. Quinlan (1993a) shows how these can be eliminated during a post-processing phase. The above rule set is then reduced to the one in Figure 3.5.

Given this strong correspondence between decision trees and rule sets, one might question whether there are any important differences at all between the representation formalisms. These do exist. We divide them into two groups: differences with respect to the representation itself, and differences with respect to the induction process.

Differences with respect to the representation

A first difference is **understandability**. A rule set is generally considered to be easier to understand than a decision tree. The hypothesis is structured in a modular way: each rule represents a piece of knowledge that can be understood in isolation from the others. (Although this is much less the case for decision lists.)

A second difference originates in the concept learning point of view, and concerns the **representation of the concept** that is to be learned. From the concept learning point of view, the concept that one wants to learn can be either the function mapping examples onto their labels, or a specific class that is to be predicted. E.g. for the Poker data set (see Appendix A), one could try to learn the concept of a pair (which is one of the classes) or consider the association of hands with names to be the concept. The first one is probably the most natural here. In the finite element mesh data set (see Appendix A), where the aim is to predict into how many pieces an edge must be divided, it makes more sense to say that the concept to be learned is that of a good mesh (i.e. predicting a good number of each edge) than to say that one wants to

[5] More surprisingly, this algorithm does not work in the first order logic framework, as we will show in Chapter 5.

learn the concept of "an edge that should be divided into four parts", among other concepts.

For the task of learning a definition for one class, rules are more suitable than trees because typical rule based approaches do exactly that: they learn a set of rules for this specific class. In a tree, one would have to collect the leaves containing this specific class, and turn these into rules.

On the other hand, if definitions for multiple classes are to be learned, the rule based learner must be run for each class separately. For each individual class a separate rule set is obtained, and these sets may be inconsistent (a particular instance might be assigned multiple classes) or incomplete (no class might be assigned to a particular instance). These problems can be solved, see e.g. (Van Laer *et al.*, 1997), but with a tree based approach they simply do not occur: one just learns one hypothesis that defines all the classes at once.

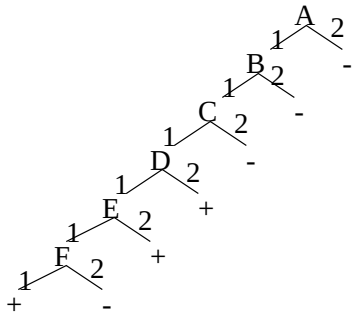
We could summarize this by saying that rules are more fit when a single concept is learned, and this concept corresponds to one single class. When multiple concepts are learned, or the concept corresponds to the mapping of instances onto classes (or numeric values), a tree based approach is more suitable.

A third difference is the way in which **exceptions** can be handled. When a concept is easiest to describe with general rules that have some exceptions (indicated by more specific rules), tree based approaches can usually handle it more easily than rule based approaches (except when decision lists are induced). The covering algorithm easily finds concepts that can be described as the union of sub-concepts, where each sub-concept can be described by a relatively simple rule; but sometimes a concept can easily be described as the difference of two concepts $C_1 = C_2 - C_3$, but much less easily as a union of other sub-concepts C_i (these C_i may have complicated descriptions which makes them harder to find). Figure 3.7 illustrates this with a simple example: a tree is given together with an equivalent rule set. The class *positive* has a complicated definition when we define it as a rule set, but a much simpler definition when written in tree format or as a decision list.

Differences with respect to the induction process

The divide-and-conquer approach is usually more **efficient** than the separate-and-conquer approach. For separate-and-conquer, the size of the set of examples that is used to grow a new rule can be expected to decrease more or less linearly (actually only the number of uncovered positive examples decreases; for each rule all the negative examples are taken into account). With a divide-and-conquer approach, if the splits are balanced, the size of the sets of examples decreases exponentially when one goes down the tree. We refer to (Boström, 1995) for more details.

Decision tree:



Rule set:

```
IF A=1 AND B=1 AND C=1 AND D=2 THEN Class = +
IF A=1 AND B=1 AND C=1 AND E=2 THEN Class = +
IF A=1 AND B=1 AND C=1 AND F=1 THEN Class = +
```

Decision list:

```
IF D=1 AND E=1 AND F=2 THEN Class = -
ELSE IF A=1 AND B=1 AND C=1 THEN Class = +
ELSE Class = -
```

Figure 3.7: A simple tree, together with an equivalent rule set. Although the rule set is much more complex than the tree, it cannot be simplified further. A decision list, however, does allow for a more compact representation.

format			theory	# literals
DNF	+	$\Leftrightarrow$	$(A=1 \wedge B=1 \wedge C=1 \wedge D=2) \vee$ $(A=1 \wedge B=1 \wedge C=1 \wedge E=2) \vee$ $(A=1 \wedge B=1 \wedge C=1 \wedge F=1)$	12
	−	$\Leftrightarrow$	$A=2 \vee B=2 \vee C=2 \vee (D=1 \wedge E=1 \wedge F=2)$	6
CNF	+	$\Leftrightarrow$	$A=1 \wedge B=1 \wedge C=1 \wedge (D=2 \wedge E=2 \wedge F=1)$	6
	−	$\Leftrightarrow$	$(A=2 \vee B=2 \vee C=2 \vee D=1) \wedge$ $(A=2 \vee B=2 \vee C=2 \vee E=2) \wedge$ $(A=2 \vee B=2 \vee C=2 \vee F=2)$	12

Table 3.2: CNF and DNF definitions for the + and − classes from Figure 3.7.

A second difference is that divide-and-conquer is a more **symmetric** approach than separate-and-conquer. The latter strategy focuses on one single class (positive) and only tries to find sufficient conditions for this class. It does not try to find sufficient conditions for the other class or classes. This causes the approach to be very dependent on a choice that the user has to make, namely for which class one wants to find sufficient conditions.

If we look again at Figure 3.7, we see that if the user had decided to learn rules for − instead of + (i.e. calling class − *positive*), a rule set would have been produced that contains only 6 conditions, just like the decision list:

```
IF A=2 THEN Class = -
IF B=2 THEN Class = -
IF C=2 THEN Class = -
IF D=1 AND E=1 AND F=2 THEN Class = -
```

The reason for this dependency is that some concepts are more easily described in disjunctive normal form (DNF), i.e. as a disjunction of conjunctions (each conjunction thus represents a sufficient condition); others are more easily described in conjunctive normal form (CNF), i.e. as a conjunction of disjunctions (each disjunction thus represents a necessary condition); and sometimes a mix of the two is even better. Whenever a concept is easier to describe in DNF, its complement is easier to describe in CNF (see (De Raedt *et al.*, 1995) for details). Table 3.2 illustrates this by comparing CNF and DNF definitions for + and −.

Most rule-based systems construct DNF descriptions. Hence, they are good at learning classes that have a simple DNF description, but less good at learning classes with a simple CNF but complex DNF description.

The fact that some concepts are easier to describe using CNF than using DNF has also been noted by Mooney (1995), who proposes an alternative learning approach that yields CNF descriptions. Also the ICL system (De Raedt

and Van Laer, 1995) originally returned CNF descriptions of concepts, although later versions of it offer the user a choice between CNF and DNF descriptions.

It is noteworthy that the divide-and-conquer approach is not subject to whether a concept is easier to describe in CNF or in DNF. We already saw that decision trees can always be converted to rule sets (i.e. DNF descriptions). In a similar fashion, they can be converted to CNF descriptions. It is possible, then, that a simple tree yields a simple DNF description of a certain class but a complex CNF description of it; that it yields a simple CNF but complex DNF description of the class; or that it does not yield any simple description at all in any of these normal forms, because the simplest description would make use of a mix of both CNF and DNF.

Conclusions

Our comparison suggests that the differences between induction of decision trees and induction of rule sets should not be underestimated; both induction methods are biased differently and may yield very different theories. We will see later that in inductive logic programming almost all systems are rule based. The difference between trees and rules is a motivation for introducing tree based methods in ILP.

3.5 Related work

This chapter is mainly based on (Blockeel *et al.*, 1998b). This work was influenced strongly by Pat Langley's discussion of decision trees in his book *Elements of Machine Learning* (Langley, 1996). It is this discussion that pointed us to the viewpoint that decision trees describe clustering hierarchies, although we have since learned that several other authors have taken this view either explicitly (e.g. Fisher (1985; 1996), who mentions the possibility of using TDIDT-techniques for clustering) or implicitly (e.g. Kramer (1996), who adopts such a view for his outlier detection method).

There are many texts on decision tree induction in the context of classification or regression; probably the most influential ones in the machine learning community are (Quinlan, 1986) and (Breiman *et al.*, 1984).

3.6 Conclusions

In this chapter we have taken a look at top-down induction of decision trees (TDIDT) from the viewpoint of predictive clustering. We have described how instantiations of this general approach yield many existing approaches to TDIDT. Induction of decision trees can thus be seen as a promising method for implementing predictive induction, as defined in the previous chapter, with

all the possibilities that it creates (such as flexible prediction, noise handling, hybrid approaches, ...) We have also compared trees with rules, mainly focusing on the differences between them, and arguing that these differences are large enough to justify the co-existence of both formalisms.

Chapter 4

First Order Logic Representations

4.1 Introduction

In the context of symbolic induction, two paradigms for representing knowledge can be distinguished: attribute value representations and first order logic representations. The latter are used in the field of inductive logic programming. Representation of knowledge in first order logic can itself be done in different ways. We distinguish two main directions in ILP: the representation of the knowledge base (data and background knowledge) as one single logic program, where each example is represented by a single clause (usually a ground fact); and the representation of examples as sets of facts or interpretations. These two different paradigms within ILP are known as *learning from entailment* and *learning from interpretations*. These are not the only possible settings; an overview of different settings and the relationships among them can be found in (De Raedt, 1997).

In this chapter we discuss in turn the three different formalisms for symbolic induction that we have mentioned: attribute value learning, learning from interpretations and learning from entailment. We will use a running example to illustrate and compare these formalisms. The task that is considered in the example is to learn definitions of concepts that are used in the poker card game (i.e. learn what a pair, double pair, full house... is).

Section 4.2 briefly discusses the relationship between concept learning and induction of predictors; this is useful for comparing the task definitions for the different settings. Next, we introduce attribute value learning in Section 4.3 and learning from interpretations in Section 4.4. The two are compared from a database point of view in Section 4.5. We then discuss learning from entailment

(Section 4.6) and relate it to the other settings in Section 4.7. The last two sections discuss related work and conclude.

4.2 Concept Learning and Intensional Clustering

Since in this chapter we will compare different representation formalisms, it is useful to first define intensional clustering using a language $\mathcal{L}$ in more detail.

Definition 4.1 (Intensional clustering space) *The intensional clustering space of a set of instances E with respect to a language $\mathcal{L}$, written $\mathbf{C}_{\mathcal{L}}(E)$, is the set of all clusterings over E for which each cluster has an intensional description in $\mathcal{L}$.*

Definition 4.2 (Intensional clustering task) *We define the task of intensional clustering in a language $\mathcal{L}$ as follows:*

Given:

- *an instance space I*
- *a distance d on I*
- *a set of instances $E \subseteq I$*
- *a language $\mathcal{L}$*
- *a quality criterion Q defined over $\mathbf{C}_{\mathcal{L}}(E)$*

Find*:*

- *a clustering $\mathcal{C} \in \mathbf{C}_{\mathcal{L}}$ such that $Q(\mathcal{C})$ is optimal, i.e.* $\forall \mathcal{C}' \in \mathbf{C}_{\mathcal{L}}(E) : Q(\mathcal{C}') \leq Q(\mathcal{C})$
- *an intensional description in $\mathcal{L}$ for each cluster $C \in \mathcal{C}$*

This clustering task can be instantiated towards predictive clustering, classification and regression, similarly to what was done in Chapter 2. Often, the language $\mathcal{L}$ is then extended slightly so that the prediction function as a whole can be described in $\mathcal{L}$.

Example 4.1 We return to the Fruit&Vegetables example. Assume $\mathcal{L}$ is propositional logic over a set of propositions

$$\mathcal{P} = \{\texttt{Color} = \texttt{red}, \texttt{Color} = \texttt{green}\} \cup \{\texttt{Weight} < x\texttt{g} | x \in I\!R\}.$$

An intensional clustering algorithm could come up with the following description of strawberries, which is in $\mathcal{L}$: `Weight < 80g` $\wedge$ `Color = red`.

For predictive purposes, a slightly different language

$$\mathcal{L}' = \{\bigwedge_i (F_i \rightarrow \texttt{Class} = v_i) | v_i \in \{\texttt{apple}, \texttt{tomato}, \texttt{strawberry}\} \wedge \texttt{F}_\texttt{i} \in \mathcal{L}\}$$

allows to describe the predictor as follows:

$$\texttt{Weight} < \texttt{80g} \wedge \texttt{Color} = \texttt{red} \rightarrow \texttt{Class} = \texttt{strawberry}$$

$\diamond$

Definition 4.3 (Concept learning) *The concept learning task can in general be stated as follows:*

Given*:*

- *an instance space* I
- *a set of positive examples* $E^+ \subseteq I$
- *a set of negative examples* $E^- \subseteq I$
- *and a language* $\mathcal{L}$

Find*: an intensional description in* $\mathcal{L}$ *of* E^+

Under the assumption that $\mathcal{L}$ is closed with respect to disjunction (i.e. if formulae $F_1 \in \mathcal{L}$ and $F_2 \in \mathcal{L}$, then $F_1 \vee F_2 \in \mathcal{L}$), which will be the case for all the languages considered in this text, concept learning can be seen as a special case of classification, where the prediction space is {`positive, negative`}. The concept description is the disjunction of those clusters for which the value `positive` is predicted.

Example 4.2 In the previous example, the concept of strawberries corresponded to one single cluster, hence the intensional description of that cluster is also an intensional description of the concept. A concept may also consist of several clusters, as this example shows: suppose we have a classifier described as

$(\texttt{X} = 1 \wedge \texttt{Y} = 0 \rightarrow \texttt{Class} = \texttt{positive}) \wedge (\texttt{X} = 1 \wedge \texttt{Y} = 1 \rightarrow \texttt{Class} = \texttt{negative}) \wedge$
$(\texttt{X} = 0 \wedge \texttt{Y} = 0 \rightarrow \texttt{Class} = \texttt{negative}) \wedge (\texttt{X} = 0 \wedge \texttt{Y} = 1 \rightarrow \texttt{Class} = \texttt{positive})$

The classifier predicts `positive` for two clusters, hence the concept description is the disjunction of both cluster descriptions:

$$\mathtt{X} = 1 \wedge \mathtt{Y} = 0 \vee \mathtt{X} = 0 \wedge \mathtt{Y} = 1$$

$\diamond$

This shows that intensional clustering, concept learning and learning predictors are all related. In the following we will often assume a predictive setting, but the results are generally applicable for clustering and concept learning as well.

4.3 Attribute Value Learning

In the attribute value formalism, it is assumed that each example can be described by a fixed set of attributes for which values are given. Each example is thus represented by a vector of values. All the data together form a table where each row corresponds to an example and each column to an attribute. The hypothesis language is propositional logic, where propositions usually are of the form "Attribute $\oplus$ value" with $\oplus$ an element of a predefined fixed set of operators, e.g. $\{<, \leq, =\}$.

Definition 4.4 (Attribute value learning) *The task of learning a predictor in the attribute value framework can be defined as follows:*

Given:

- *a target variable Y*
- *a set of labelled examples E; each example is a vector labelled with a value y for the target variable*
- *and a language $\mathcal{L}$ consisting of propositional logic over a given set of propositions $\mathcal{P}$*

Find: *a hypothesis H of the form $\bigwedge_i (F_i \rightarrow \mathtt{Y} = y_i)$ where each $F_i \in \mathcal{L}$ such that for each example e with label y,*

- $H \models \mathtt{Y} = y$
- $\forall y' \neq y : H \not\models \mathtt{Y} = y'$

The attribute value formalism is the formalism that is most frequently used for data mining and symbolic machine learning. Most systems working in this framework do not allow the use of a background theory about the domain, although in principle this would not be a problem.

Rank1	Suit1	R.2	S.2	R.3	S.3	R.4	S.4	R.5	S.5	Class
7	♠	8	♡	K	♣	Q	♡	7	♡	pair
2	♠	5	♣	A	♣	4	♠	Q	◇	nought
3	♡	3	♣	8	♣	8	♠	8	♡	full house
4	♠	2	♠	A	♠	A	♣	9	♠	pair
A	♣	A	◇	2	♠	6	◇	4	♡	pair
5	♡	4	♡	7	♡	K	♣	2	◇	nought

Table 4.1: Representing examples for learning poker concepts. Each tuple represents one hand of five cards and the name that is given to the hand.

Table 4.1 shows how poker hands can be represented in this framework. Each row in the table describes one example (one hand of cards) by listing the rank and suit of every card in the hand, and also stating its class label.

While this may seem a very natural representation of poker hands, it is not fit for learning definitions of the classes. A good definition for the concept of a pair is very hard to write if this format is used: one needs to check that exactly two cards have the same rank. A possible definition of the concept of a pair is the following:[1]

(`Rank1=2` $\wedge$ `Rank2=2` $\wedge$ [Ranks 3,4,5 differ from 2 and from each other])
$\vee$ (`Rank1=3` $\wedge$ `Rank2=3` $\wedge$ [Ranks 3,4,5 differ from 3 and from each other])
$\vee \cdots$
$\vee$ (`Rank1=2` $\wedge$ `Rank3=2` $\wedge$ [Ranks 2,4,5 differ from 2 and from each other])
$\vee \cdots$
$\rightarrow$ `Class = pair`

Note that the cards with equal ranks could be any 2 of the 5 cards, which yields $C_5^2 = 10$ combinations. Moreover, one cannot test the equality of two attributes directly, they can only be compared with constants; therefore it is necessary to test for the 13 different constants whether the ranks of 2 specific cards are equal to them. This means that the hypothesis as written above contains 130 lines. Note that each line in itself is far from complete: each part between brackets is to be expanded, for instance into a disjunction of 132 conjunctions similar to

$$\texttt{Rank3} = 3 \wedge \texttt{Rank4} = 4 \wedge \texttt{Rank5} \neq 2 \wedge \texttt{Rank5} \neq 3 \wedge \texttt{Rank5} \neq 4$$

The full definition would then contain a total of 17160 such conjunctions.

While the above format of the hypothesis is not necessarily the simplest one, any correct hypothesis will have a complexity that is of the same order

[1] The definition is written in the form of a predictor, conform to the Definition 4.4; the actual concept description is the antecedent of the rule.

Er12	Er13	Er14	Er15	Er23	Er24	Er25	Er34	Er35	Er45	Class
no	no	no	yes	no	no	no	no	no	no	pair
no	no	no	no	no	no	no	no	no	no	nought
yes	no	no	no	no	no	no	yes	yes	yes	full house
no	no	no	no	no	no	no	yes	no	no	pair
yes	no	no	no	no	no	no	no	no	no	pair
no	no	no	no	no	no	no	no	no	no	nought

Table 4.2: Constructed attributes for learning poker concepts. The meaning of, e.g., Er25 is that cards 2 and 5 have equal rank.

of magnitude. The problem is that in the attribute value framework the data simply cannot be handled in the format of Table 4.1.

A better way to tackle the problem is to construct new attributes and add them to the table. Observing that equality of different attributes (in this case: equality of the ranks of different cards) may be interesting to check, we can explicitly add information about such equalities to the table. This yields Table 4.2. There are 10 attributes $\texttt{Er}_{ij}$ in this table; $\texttt{Er}_{ij}$=yes if the ranks of the cards i and j are equal.

One could now learn from the original data in Table 4.1 augmented with the constructed attributes in Table 4.2, or use the constructed attributes only. The latter is sufficient for learning the concept of a pair, but not for, e.g., learning a flush or straight.

The definition of a pair can now be written in a much simpler way:

(Er12 = yes $\wedge$ Er13 = no $\wedge \cdots$)
$\vee$ (Er12 = no $\wedge$ Er13 = yes $\wedge \cdots$)
$\vee \cdots$
$\vee$ (Er12 = no $\wedge \cdots \wedge$ Er45 = yes)
$\rightarrow$ Class = pair

This definition contains 10 conjuncts, and each conjunct consists of 10 terms. It basically says that a hand is a pair if exactly one of the ten Er-attributes is a yes. This makes clear that by constructing one more attribute, one that indicates how many yes values there are in a tuple, a very concise definition can be obtained: Pairs_with_equal_rank = 1 $\rightarrow$ Class = pair.

This example illustrates a number of important aspects of attribute value learning:

- Learning from the most natural representation of the examples is not always feasible. It may be necessary to add attributes to the data. These new attributes are supposed to capture properties of the example that are expected to be important for the concept that is to be learned. Note that the new attributes do not add new information about the example:

they can be computed from the other attributes. They only explicitate information.

- The complexity of a hypothesis depends on the attributes that are available. A hypothesis can seem deceptively simple if it makes use of attributes that are very complex to compute.

- It may not be obvious to find good attributes. In the above example we could easily come up with constructed attributes because we already knew what the correct definition of a pair is. In general the task of constructing good attributes is much harder.

Note that constructing good attributes is as much a part of the knowledge discovery process as the search for a good hypothesis itself is.

The dependency of attribute value learners on the representation can be solved in several ways. One is to have an inductive learner construct good attributes itself. This task is known as *feature construction* and has been the subject of many studies, see e.g. (Wnek and Michalski, 1993; Bloedorn and Michalski, 1996; Srinivasan and King, 1996; Kramer *et al.*, 1998). Another approach to alleviate the problem is enlarging the hypothesis space by allowing tests that involve multiple attributes. For instance, one could have a test such as `Rank1=Rank2`. This approach is taken by Bergadano *et al.* (1997) with the RELIC system.

4.4 Learning from Interpretations

In the learning from interpretations setting each example e is represented by a separate Prolog program that encodes its specific properties, as well as its label. Background knowledge about the domain can be given in the form of a Prolog program B. The interpretation that represents the example is the set of all the ground facts that are entailed by $e \wedge B$ (this set is called the *minimal Herbrand model* of $e \wedge B$). The hypothesis to be induced is a first order theory, represented as a Prolog program.

Example 4.3 Suppose we have an example called Tweety, represented by the following Prolog program:

```
bird.
color(yellow).
likes(bird_seed).
gets(bird_seed).
```

and a background theory

```
flies :- bird.
swims :- fish.
happy :- likes(X), gets(X).
```

then the interpretation representing Tweety is {`bird, color(yellow), likes(bird_seed), gets(bird_seed), flies, happy`}. It sums up all the properties of Tweety that can be derived from its description together with the background knowledge. ◇

The essential difference with attribute value learning is that a set is used to represent an example, instead of a (fixed-length) vector. This makes the representation much more flexible.

Definition 4.5 (Learning from interpretations) *Learning from interpretations is usually set in a predictive context, where the task definition is the following:*

Given:

- *a target variable* Y
- *a set of labelled examples* E*; each example is a Prolog program* e *labelled with a value* y *for the target variable*
- *a language* $\mathcal{L} \subseteq$ *Prolog*
- *and a background theory* B

Find: *a hypothesis* $H \in \mathcal{L}$ *such that for all labelled examples* $(e, y) \in E$*,*

- $H \wedge e \wedge B \models label(y)$*, and*
- $\forall y' \neq y : H \wedge e \wedge B \not\models label(y')$

This task reduces to the concept learning task by choosing as labels `positive` and `negative`.

Table 4.3 shows how the hands would be represented in the learning from interpretations setting. Note that the label need not be indicated explicitly by a predicate called `label`; in this example the nullary predicates `pair`, `full_house` etc. represent the labels.

A correct definition of a pair in first order logic is:

$$\begin{aligned} &\exists Rank, Suit_1, Suit_2 : \texttt{card}(Rank, Suit_1) \wedge \texttt{card}(Rank, Suit_2) \wedge Suit_1 \neq Suit_2 \\ &\quad \wedge \nexists Suit_3 : (\texttt{card}(Rank, Suit_3) \wedge Suit_3 \neq Suit_1 \wedge Suit_3 \neq Suit_2) \\ &\quad \wedge \nexists Rank_2, Suit_3, Suit_4 : \texttt{card}(Rank_2, Suit_3), \texttt{card}(Rank_2, Suit_4), Rank_2 \neq \\ &\qquad Rank, Suit_4 \neq Suit_3 \\ &\qquad \rightarrow \texttt{pair} \end{aligned}$$

```
{card(7, spades), card(8, hearts), card(king, clubs),
 card(queen, hearts), card(7, hearts), pair}

{card(2, spades), card(5, clubs), card(ace, clubs),
 card(4, spades), card(queen, diamonds), nought}

{card(3, hearts), card(3, clubs), card(8, clubs),
 card(8, spades), card(8, hearts), fullhouse}

{card(4, spades), card(2, spades), card(ace, spades),
 card(ace, clubs), card(9, spades), pair}

{card(ace, clubs), card(ace, diamonds), card(2, spades),
 card(6, diamonds), card(4, hearts), pair}

{card(5, hearts), card(4, hearts), card(7, hearts),
 card(king, clubs), card(2, diamonds), nought}
```

Table 4.3: Representing the poker data in the learning from interpretations setting.

The first line states that there must be two different cards with equal ranks, the next line signifies that there must not be a third card with the same rank, and the last two lines of the antecedent add the constraint that there must be no other pair of cards with equal ranks.

The definition can also be written as a Prolog program, which is the form in which most ILP systems represent the result. A Prolog program equivalent to the above definition is shown in Figure 4.1.

Note that this definition, although relatively complex, is much simpler than the first two definitions that were given in the attribute value learning framework. Still, the definition does not make use of any constructed features in the data, whereas the second propositional definition did.

Of course, one could still add constructed features to the data, e.g. counting how many combinations of two cards with the same rank there are. This could be done by adding a `pairs_with_equal_ranks` fact to each example[2]; for instance the first example would be extended with `pairs_with_equal_ranks(1)`. The definition then becomes

$$\texttt{pairs\_with\_equal\_ranks}(1) \rightarrow \texttt{pair}.$$

[2] Alternatively, the predicate can be defined in the background by writing a Prolog program that counts the number of pairs with equal ranks in a hand.

```
pair :-
  card(Rank, Suit1), card(Rank, Suit2), Suit1 \= Suit2,
  not third_card(Rank, Suit1, Suit2),
  not second_pair(Rank, Suit1, Suit2).

third_card(Rank, Suit1, Suit2) :-
  card(Rank, Suit3), Suit1 \= Suit2, Suit1 \= Suit3.

second_pair(Rank, Suit1, Suit2) :-
  card(Rank2, Suit3), card(Rank2, Suit4),
  Rank2 \= Rank, Suit4 \= Suit3.
```

Figure 4.1: A correct definition of a pair in Prolog.

This shows that feature construction can also be done (at least in principle) in ILP, and that there as well it leads to more compact definitions. The need for feature construction is smaller in ILP than in attribute value learning, due to the greater expressivity of first order logic, but it still exists. Feature construction can be seen as a special case of *predicate invention*, which is a difficult and extensively studied topic in ILP (see, e.g., (Stahl, 1996) for an overview).

4.5 A Relational Database Viewpoint

ILP systems use logic-based data representations, but in practice, data are often stored in a relational database. In this section, we discuss the difference between attribute value learning and first order learning from a relational database point of view. This offers an interesting and clear view of the relationship between the two settings, namely that first order learning amounts to learning from multiple relations and attribute value learning amounts to learning from one relation.

There is a simple way to convert the data in a relational database to the learning from interpretations format; in the second part of this section we present an algorithm.

4.5.1 Attribute Value Learning Versus Learning from Multiple Relations

Attribute value learning always involves learning from one single relation, and more specifically: learning a hypothesis that relates different attributes of the

same tuple. We could call this single-relation, single-tuple learning. In the first order context, one can learn patterns that relate attribute values of several tuples, possibly belonging to different relations, to one another.

Example 4.4 Assume the following database is given:

FINES

Name	**Job**	**Speed**	**Fine**
Ann	teacher	150 km/h	Y
Bob	politician	160 km/h	N
Chris	engineer	120 km/h	N
Dave	writer	155 km/h	N
Earnest	politician	120 km/h	N

KNOWS

Name1	**Name2**
Ann	Chris
Ann	Dave
Bob	Earnest
Chris	Dave
Dave	Bob

An attribute value learner can learn a rule such as "if you drive faster than 120 km/h and are not a politician, you get a fine", but not the (more correct) rule "if you drive faster than 120 km/h, are not a politician and do not know a politician, then you get a fine". This rule relates attributes of different tuples in the relation FINES that are linked via the relation KNOWS. ◇

It is known from relational database theory, see e.g. (Elmasri and Navathe, 1989), that storing data in one single table can be very inefficient; it is advisable to structure a database into several relations that are in some normal form. Attribute value learning is limited in the sense that in order to take into account all the information in a database, one has to convert all the relations into a single relation. The resulting relation can be huge, typically containing a large amount of redundant information; moreover its computation is costly. This problem is ignored in many approaches to data mining. In his invited talk at the *Fifteenth International Conference on Machine Learning* (1998), Ron Kohavi acknowledged the shortcomings of attribute value learning in this respect and stressed the need for learning techniques that learn from multiple relations. De Raedt (1998) shows that the conversion of multiple relations to a single relation, together with the problems it generates for learning (as we have seen with the Poker example, correct theories may become extremely complex and hard to find), causes attribute value learning to become much more inefficient than first order learning on this kind of problems.

The following is an example of a database where learning from multiple relations is desirable, and where joining the relations into one single relation is clearly not feasible.

Example 4.5 Assume that one has a relational database describing molecules. The molecules themselves are described by listing the atoms and bonds that occur in them, as well as some properties of the molecule as a whole. Mendelev's

MENDELEV

Number	**Symbol**	**Atomic weight**	**Electrons in outer layer**	...
1	H	1.0079	1	
2	He	4.0026	2	
3	Li	6.941	1	
4	Be	9.0121	2	
5	B	10.811	3	
6	C	12.011	4	
...	...	...	...	...

MOLECULES

Formula	**Name**	**Class**
H_2O	water	inorganic
CO_2	carbon dioxide	inorganic
CO	carbon monoxide	inorganic
CH_4	methane	organic
CH_3OH	methanol	organic
...	...	...

CONTAINS

Formula	**Atom_id**
H_2O	h2o-1
H_2O	h2o-2
H_2O	h2o-3
CO_2	co2-1
CO_2	co2-2
...	...

ATOMS

Atom_id	**Element**
h2o-1	H
h2o-2	O
h2o-3	H
co2-1	O
...	...

BONDS

Atom_id1	**Atom_id2**	**Type**
h2o-1	h2o-2	single
h2o-2	h2o-3	single
co2-1	co2-2	double
co2-2	co2-3	double
...	...	...

Figure 4.2: A chemical database.

periodic table of elements is a good example of background knowledge about this domain. Figure 4.2 illustrates what such a chemical database could look like. A possible classification problem here is to classify unseen molecules into organic and inorganic molecules, based on their chemical structure. ◇

Note that the database contains both information about specific molecules (i.e. specific examples) and background knowledge (e.g., Mendelev's periodic table).

This example should make clear that in many cases joining all the relations into one single, huge relation is not an option. The information in Mendelev's table, for instance, would be duplicated many times. Moreover,

unless a multiple-instance learner is used (Dietterich *et al.*, 1997) all the atoms a molecule consists of, together with their properties, have to be stored in one tuple, so that an indefinite number of attributes is needed.

While mining such a database is not feasible using propositional techniques, it is feasible using learning from interpretations. We now proceed to show how a relational database can be converted into a suitable format.

4.5.2 Conversion from Relational Database to Interpretations

Typically, each predicate in the logical representation will correspond to one relation in the relational database. Each fact in an interpretation is a tuple in the database, and an interpretation corresponds to a part of the database (a set of subsets of the original relations). Background knowledge can be stored in the relational database by means of views as well as extensional tables.

Converting a relational database to a set of interpretations can be done easily and in a semi-automated way via the following procedure and with the help of the INTERPRETATIONS algorithm in Figure 4.3.

- Decide which relations in DB are background knowledge.
- Let KB be the original database without the background relations.
- Choose an attribute K in a relation that uniquely identifies the examples; we call this the example key.
- Collect all foreign keys in the database scheme of KB in a set FK. Foreign keys are attributes that are in fact references to other attributes; we write a foreign key from $R_1.A$ to $R_2.B$ as $(R_1.A \rightarrow R_2.B)$. For instance, in Example 4.5 the *Name* attribute in FINES is a foreign key to **Name1** in KNOWS, which we write FINES.**Name** $\rightarrow$ KNOWS.**Name1**.
- Run INTERPRETATIONS(KB, K, FK). This algorithm works as follows. For each example with key k, a database KB_k is constructed in which each relation is a subrelation of a relation in KB. The algorithm first gathers all the tuples in KB that contain k, and puts them in the corresponding relations in KB_k. It then keeps adding tuples outside KB_k that are referenced from inside KB_k to relations in KB_k, until KB_k does not change anymore. After this has been done for each k, the set of all databases KB_k is returned.

  Note that values are assumed to be typed in this algorithm, i.e., two values are considered to be equal only if their types are also equal. For instance, if a relation PERSON has an **ID** attribute and an **Age** attribute that are both represented as numbers, PERSON.**ID** = PERSON.**Age** never

procedure INTERPRETATIONS(KB: database, K: example key,
FK: set of foreign keys)
returns set of databases
  for each value k of K:
    {*add tuples that contain example key* k}
    for each $R_i \in KB$:
      $R_i' := \{r_i \in R_i | \exists A : r_i.A = k\}$
    $KB_k := \bigcup_i \{R_i'\}$
    repeat
      {*add tuples that are referenced from* S *via a foreign key*}
      for each $R_i' \in KB_k$:
        $R_i' := R_i' \cup \{r_i \in R_i | \exists (R_j.A \rightarrow R_i.B) \in FK : \exists r_j \in R_j' : r_j.A = r_i.B\}$
    until KB_k does not change anymore
  return $\bigcup_k KB_k$

Figure 4.3: Conversion of a relational database to interpretations.

succeeds even if the values of both attributes are equal, because a person identification number and an age cannot be the same.

- Convert the interpretations and background relations into logic format.

A tuple $(attr_1, \ldots, attr_n)$ of a relation R can trivially be converted to a fact $R(attr_1, \ldots, attr_n)$. By doing this conversion for all KB_k, each KB_k becomes a set of facts describing an individual example k. The extensional background relations can be converted in the same manner into one set of facts that forms the background knowledge. Background relations defined by views can be converted to equivalent Prolog programs.

The only parts in this conversion process that are hard to automate are the selection of the background knowledge (typically, one selects those relations where each tuple can be relevant for many examples) and the conversion of view definitions to Prolog programs. Also, the user must indicate which attribute should be chosen as an example key, as this depends on the learning task.

Example 4.6 In the chemical database, we choose as example key the molecular formula. The background knowledge consists of the MENDELEV table. In order to build a description of H_2O, the INTERPRETATIONS algorithm in Figure 4.3 first collects the tuples containing H_2O; these are present in MOLECULES and CONTAINS. The tuples are put in the relations MOLECULES' and CONTAINS'. The database KB_{H_2O} is thus initialized as shown in Figure 4.4.

The tuples in KB_{H_2O} contain references to **Atom_id**'s h2o-i, $i = 1, 2, 3$, so the tuples containing those symbols are also collected (tuples from ATOMS and BONDS). The result is shown in the second part of Figure 4.4. The new tuples refer to the elements H and O, which are foreign keys for the MENDELEV relation. Since this relation is in the background, no further tuples are collected; hence, KB_{H_2O} as shown in the second part of Figure 4.4 is complete.

Converting the tuples to facts, we get the following description of H_2O:

```
{molecules('H2O', water, inorganic), contains('H2O', h2o-1),
contains('H2O', h2o-2), contains('H2O', h2o-3), atoms(h2o-1,
'H'), atoms(h2o-2, 'O'), atoms(h2o-3, 'H'), bonds(h2o-1, h2o-2,
single), bonds(h2o-2, h2o-3, single)}
```

◇

Some variations of this algorithm can be considered. For instance, when the example key has no meaning except that it identifies the example, this attribute can be left out from the example description (in our example, all occurrences of `'H2O'` could be removed if we consider this name unimportant).

The key notion in this conversion process is *localization of information*. It is assumed that for each example only a relatively small part of the database is relevant, and that this part can be localized and extracted. Further in this text, we will refer to this assumption as the *locality assumption*.

Definition 4.6 (Locality assumption) *The locality assumption states that all the information that is relevant for a single example is located in a small part of the database.*

If the locality assumption is not fulfilled, the localization process as described above would give rise to a significant duplication of information.

4.6 Learning from Entailment

Learning from entailment is the most frequently used paradigm within inductive logic programming. The framework is described in e.g. (Muggleton and De Raedt, 1994) (under the name of *normal semantics*) and at this moment almost all ILP systems use it, e.g. PROGOL (Muggleton, 1995), FOIL (Quinlan, 1990), SRT (Kramer, 1996), FORS (Karalič and Bratko, 1997), ...

All the data, as well as the background knowledge, are encoded in a Prolog program. The concept that is to be learned is represented by a predicate, and the learner has to learn a definition for the predicate from a set of positive and negative examples. In principle, this can go as far as the synthesis of a full logic program from examples. In practice, the problems that are considered are often constrained or simplified versions of this very general and difficult task.

Initialization:

MOLECULES'

Formula	Name	Class
H_2O	water	inorganic

CONTAINS'

Formula	Atom_id
H_2O	h2o-1
H_2O	h2o-2
H_2O	h2o-3

ATOMS'

atom_id	element

BONDS'

Atom_id1	Atom_id2	Type

After one step:

MOLECULES'

Formula	Name	Class
H_2O	water	inorganic

CONTAINS'

Formula	Atom_id
H_2O	h2o-1
H_2O	h2o-2
H_2O	h2o-3

ATOMS'

atom_id	element
h2o-1	H
h2o-2	O
h2o-3	H

BONDS'

Atom_id1	Atom_id2	Type
h2o-1	h2o-2	single
h2o-2	h2o-3	single

Figure 4.4: Construction of the subdatabase KB_{H_2O}.

For instance, predictive induction such as classification and regression can be seen as a constrained version of program synthesis. Examples are then typically represented by constants, and they are labelled either using a unary predicate (e belongs to class p if $p(e)$ is true), or using a binary predicate that indicates the label of the example ($label(e, y)$ means that the label of example e is y).

Definition 4.7 (Learning from entailment) *In the learning from entailment framework, we define the concept learning task as follows:*

Given:

- *a set of positive examples* E^+ *(each example is a clause)*
- *a set of negative examples* E^-
- *background knowledge* B *(a Prolog program)*
- *and a first order language* $\mathcal{L} \subseteq$ *Prolog*

Find: *a hypothesis* $H \subseteq \mathcal{L}$ *such that*

- $\forall e \in E^+ : H \wedge B \models e$ *and*
- $\forall e \in E^- : H \wedge B \not\models e$

Table 4.4 shows how the data from the Poker example would typically be represented in this ILP setting: each example is represented by a single fact.[3] Note that every hand that is not a pair is explicitly written as a negative example. Some ILP systems would automatically deduce from the fact that e.g. `hand3` is a full house, that it is not a pair; in that case no negative examples need to be given. Note, however, that the learner then makes the assumption that the predicate to be learned is a function. This is general enough for the classification or regression setting but less general than learning a predicate. Examples of ILP systems learning from positive examples only are FOIDL (Mooney and Califf, 1995), FFOIL (Quinlan, 1996) and PROGOL (in a specific setting, see (Muggleton, 1996; Muggleton *et al.*, 1996)).

The following first order logic formula defines the concept of a pair:

$$\exists Rank, Suit_1, Suit_2 : \texttt{card}(Hand, Rank, Suit_1) \wedge \texttt{card}(Hand, Rank, Suit_2) \wedge Suit_1 \neq Suit_2$$
$$\wedge \nexists Suit_3 : \texttt{card}(Hand, Rank, Suit_3) \wedge Suit_3 \neq Suit_1 \wedge Suit_3 \neq Suit_2$$
$$\wedge \nexists Rank_2, Suit_3, Suit_4 :$$
$$\texttt{card}(Hand, Rank_2, Suit_3) \wedge \texttt{card}(Hand, Rank_2, Suit_4) \wedge Suit_3 \neq Suit_4 \wedge Rank_2 \neq Rank$$
$$\rightarrow \texttt{pair}(Hand)$$

[3] In principle, one can also use non-fact clauses to represent the examples, but this is not often done.

```
card(hand1, 7, spades).
card(hand1, 8, hearts).
card(hand1, king, clubs).
card(hand1, queen, hearts).
card(hand1, 7, hearts).
card(hand2, 2, spades).
card(hand2, 5, clubs).
card(hand2, ace, clubs).
card(hand2, 4, spades).
card(hand2, queen, diamonds).
card(hand3, 3, hearts).
card(hand3, 3, clubs).
card(hand3, 8, clubs).
card(hand3, 8, spades).
card(hand3, 8, hearts).
card(hand4, 4, spades).
card(hand4, 2, spades).
card(hand4, ace, spades).
card(hand4, ace, clubs).
card(hand4, 9, spades).
card(hand5, ace, clubs).
card(hand5, ace, diamonds).
card(hand5, 2, spades).
card(hand5, 6, diamonds).
card(hand5, 4, hearts).
card(hand6, 5, hearts).
card(hand6, 4, hearts).
card(hand6, 7, hearts).
card(hand6, king, clubs).
card(hand6, 2, diamonds).

% positive examples
pair(hand1).
pair(hand4).
pair(hand5).

% negative examples
:- pair(hand2).    %nought(hand2).
:- pair(hand3).    %fullhouse(hand3).
:- pair(hand6).    %nought(hand6).
```

Table 4.4: Representing the poker data in the learning from entailment setting. Negative examples are written with a preceding `:-` symbol; the original information is written as a comment.

```
pair(Hand) :-
  card(Hand, Rank, Suit1), card(Hand, Rank, Suit2),
  Suit1 \= Suit2,
  not third_card(Hand, Rank, Suit1, Suit2),
  not second_pair(Hand, Rank, Suit1, Suit2).

third_card(Hand, Rank, Suit1, Suit2) :-
  card(Hand, Rank, Suit3), Suit1 \= Suit2, Suit1 \= Suit3.

second_pair(Hand, Rank, Suit1, Suit2) :-
  card(Hand, Rank2, Suit3), card(Hand, Rank2, Suit4),
  Rank2 \= Rank, Suit4 \= Suit3.
```

Figure 4.5: A correct definition of a pair in Prolog.

A Prolog version of this definition is shown in Figure 4.5. Note that this definition is very similar to the one found when learning from interpretations; the main difference is that when learning from interpretations no explicit reference to an example identifier (the variable *Hand*) is given; all information is automatically assumed to be information about the example that is to be classified, not about any other example.

With respect to the representation of the data (comparing Table 4.4 with Table 4.3), we note that in the learning from interpretations setting information about one example is clearly separated from information about other examples[4], whereas when learning from entailment all this information, together with the background knowledge, forms one single Prolog program. We will return to this in the next section.

The Poker example is an example of a classification task. Once more, we stress that ILP is much more general and can be used to learn (in principle) any definition of a predicate, e.g. from the following data:

```
member(1, [1,2,3]).
member(5, [3,8,5,2]).
:- member(2, [1,4]).
:- member(8, []).
:- member(12, [1,2,3]).
```

one could induce a definition for the `member` predicate:

```
member(X, [X|Y]).
```

[4]It is also separated from the background knowledge, although this is less clear here because in this example there is no background knowledge.

```
member(X, [Y|Z]) :- member(X, Z).
```

This illustrates that in the learning from entailment setting, ILP has an extremely broad application potential. In practice, this potential is severely limited due to the complexity of many of these tasks. Within computational learning theory several negative results have been published (Džeroski *et al.*, 1992; Cohen and Page, 1995; Cohen, 1995), showing that many tasks that can in principle be handled by ILP systems cannot be handled in practice, due to practical limitations such as space and time constraints (more specifically: they are not PAC-learnable with polynomial time complexity; PAC stands for *probably approximately correct*, see Intermezzo 1 for a brief discussion on PAC-learnability). These results have strongly influenced the attitude towards ILP of many researchers in machine learning and data mining.

An interesting property of the learning from interpretations setting is that under fairly general assumptions, first order hypotheses are PAC-learnable in this setting (De Raedt and Džeroski, 1994). This is mainly due to the fact that the setting exploits the locality assumption; we will discuss this in the next section.

4.7 Relationships Between the Different Settings

4.7.1 On the Origin of Learning From Interpretations

Originally, learning from interpretations (also called *nonmonotonic* ILP) was seen as an alternative setting, where a hypothesis is a set of clauses such that each positive example (together with the background) makes each clause in the set true, and none of the negative examples do so. I.e., a hypothesis H is to be found such that

- $\forall e \in E^+ : H$ is true in $\mathcal{M}(e \wedge B)$
- $\forall e \in E^- : H$ is false in $\mathcal{M}(e \wedge B)$

where $\mathcal{M}(e \wedge B)$ is the minimal Herbrand model of $e \wedge B$ (intuitively: the interpretation represented by $e \wedge B$, as illustrated in Example 4.3). This is to be contrasted with the classical setting where

- $\forall e \in E^+ : H \wedge B \models e$
- $\forall e \in E^- : H \wedge B \not\models e$

Note that in the classical setting H (together with B) is supposed to explain e, while in the nonmonotonic setting H describes e; therefore the settings are also sometimes called *explanatory ILP* and *descriptive ILP*. A brief history of how learning from interpretations evolved from a descriptive technique towards a predictive (explanatory) technique is given in Intermezzo 2.

Intermezzo 1: PAC-learnability

Computational learning theory (COLT) is the subfield of machine learning that is concerned with describing how hard a certain learning task is. In general, the complexity of an algorithm is described by giving its time and space requirements for solving a certain problem, in terms of certain parameters that are indicative of the size or difficulty of the problem. Based on such a description algorithms can for instance be classified as tractable (execution time is polynomial in the parameters) or not tractable (execution time is not polynomial but, e.g., exponential).
Of course, complexity results depend on the parameter that is used; e.g., if adding two numbers is linear in the number of bits used to represent the numbers, it is logarithmic in the numbers themselves; and if an algorithm for computing the factorial of a number n has a time complexity $O(n)$, it is exponential in the number of bits used to represent the number. Whether this algorithm is tractable or not depends on the parameters in terms of which the complexity is described. Hence, in order to adequately describe the complexity of learning tasks, these parameters need to be fixed.
Nowadays the PAC-learning framework, introduced by Valiant (1984) is probably the most popular framework for studying the complexity of learning tasks or algorithms. PAC stands for *probably approximately correct*. In this framework, it is assumed that the task is to learn a hypothesis that approximates the correct theory to an agreeable extent. A learning algorithm need not guarantee that it will find such a hypothesis, but should have a certain probability (close to 1) of doing so.

Definition 4.8 (δ-correctness) *A predictor* pred *is δ-correct if and only if $P(pred(x) \neq \pi(x)) < \delta$ (where $\pi(x)$ is the class of x).*

Definition 4.9 (PAC-learnability) *A function π is PAC-learnable if and only if for each $\delta > 0$ and for each $\epsilon > 0$ there exists an algorithm that, given a sufficient number of examples, with probability $1-\epsilon$ induces a δ-correct predictor pred.*

The *sample complexity* is the number of examples needed to learn such a theory; the *time complexity* is the time needed to learn it. A PAC-learning algorithm is considered to be *tractable* if it has time complexity polynomial in $1/\delta$ and $1/\varepsilon$.

Intermezzo 2: History of Learning From Interpretations

The nonmonotonic setting (Helft, 1989) was originally meant for descriptive induction. The first ILP system to use this setting was CLAUDIEN (De Raedt and Bruynooghe, 1993; De Raedt and Dehaspe, 1997), which returned clausal descriptions of regularities in the data. The development from clausal discovery towards prediction can be sketched as follows:

1. CLAUDIEN, the clausal discovery engine, examines a set of data D and returns as hypothesis H all the clauses (within a certain language, and excluding redundant clauses) that are true for the data (De Raedt and Bruynooghe, 1993).
For instance, given a set of data on family relationships, CLAUDIEN would derive clauses such as $\leftarrow$ `parent(X,X)` (nobody is his own parent) or `sibling(X,Y)` $\leftarrow$ `parent(X,Z), parent(Y,Z), X` $\neq$ `Y` (people with the same parent are siblings).

2. A later version of CLAUDIEN can handle multiple sets of data. It returns all clauses that are true within each data set. It is recognised that, by regarding the hypothesis as a definition of a concept, CLAUDIEN can be seen as a concept learner that learns from positive examples only (De Raedt and Džeroski, 1994). It learns the most specific concept that covers all the examples.
For instance, CLAUDIEN could be provided with 10 data sets, each set describing a different typical Belgian family. The system would then return a hypothesis that represents the concept of a typical Belgian family, containing the same clauses as shown above but also clauses such as `Y=Z` $\leftarrow$ `married(X,Y), married(X,Z)` (one can be married to only one person).

3. A further development is the ICL system (De Raedt and Van Laer, 1995). The interpretations fed into this system are labelled; all the interpretations with a certain label are considered to be positive examples, the other interpretations are negative examples. ICL then tries to find clauses that are true in all the positive examples, but are false in at least some of the negative examples. It searches for a minimal set of clauses such that each negative example violates at least one clause. ICL thereby is the first ILP system to learn a concept from positive and negative examples that are represented as interpretations.
For instance, suppose ICL is provided with 10 families labelled "Belgian" and 10 families labelled "Kuwaitian". ICL then tries to find a minimal set of clauses that allows to distinguish Belgian from Kuwaitian families. Such a set could contain the clause `Y=Z` $\leftarrow$ `married(X,Y), married(X,Z)`, if this is true for all Belgian families but violated for some Kuwaitian families.

4. It is noticed that the hypotheses returned by ICL, which (being sets of clauses) are in conjunctive normal form (CNF), usually become easier to interpret if they are converted to disjunctive normal form (DNF). ICL is extended so that it can learn DNF theories instead of CNF theories, if the user desires this. A note on the relationship between DNF and CNF is (De Raedt *et al.*, 1995), which also links to earlier work on learning CNF (Mooney, 1995).
For instance, assuming the clause `Y=Z` $\leftarrow$ `married(X,Y), married(X,Z)` is useful for distinguishing Belgian families from Kuwaitian families, a sufficient condition for a family being Kuwaitian would be `married(X,Y), married(X,Z), Y` $\neq$ `Z`. Such a condition would appear in the DNF definition of Kuwaitian families.

4.7.2 Learning From Interpretations Links Attribute Value Learning to Learning From Entailment

Learning from interpretations can be situated in between the other settings; in a sense it provides a link between them. Learning from interpretations differs from learning from entailment (as it is practically used) in that it exploits the locality assumption by providing a clear separation between different pieces of information, in two ways:

- the information that is contained in examples is separated from the information in the background knowledge;
- information in one example is separated from information in another example.

It is this separation of information that provides the link with attribute value learning, where such a separation is also present.

Figure 4.6 gives a graphical illustration of how the three settings differ. In attribute value learning, the learner has to find a link between the information in one example and the target variable. When learning from interpretations, the learner has to link the information that is contained in one example, together with the background knowledge, to the target. When learning from entailment, the learner has to link an indefinitely large part of the database to the target variable. It is possible that only a part of the information in the database is relevant, but in general this is not guaranteed, and even when it is it may not be obvious which part that is. Hence, the learner has to look up the relevant information in a large database, which may be a costly operation.

Figure 4.6 also illustrates a second point: a difference in assumptions that can be made about completeness of knowledge. If we do not consider the possibility of missing or noisy values, then each example description in attribute value learning is complete. Since the examples are only a sample from a certain population, the knowledge about the population itself is not complete. We say that example descriptions are *closed*, and the description of the population is *open*.

For learning from interpretations, the situation is exactly the same. For learning from entailment, it is very different however. Due to the fact that no clear separation between information about specific examples is made, individual example descriptions are not distinguished from the description of the training sample as a whole, and hence example descriptions must be open.[5]

[5] Actually, the locality can be introduced in learning from entailment by learning from clauses with a non-empty body. The information local to an example is then put in the body of the clauses. This representation is not often used, however, and few systems support it.

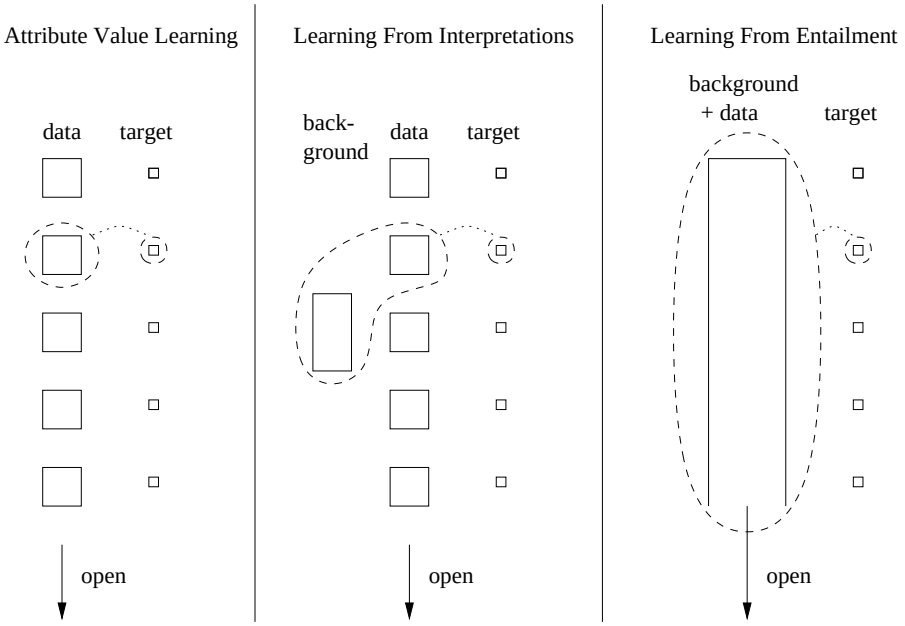


Figure 4.6: Graphical representation of the relationship between the different settings, focusing on separation of information and openness.

4.7.3 Advantages of Learning From Interpretations

The separation of information that is realized by learning from interpretations has positive effects in two respects.

First of all, it affects the efficiency of the learning process. De Raedt and Džeroski (1994) have shown that in the learning from interpretations setting, PAC-learning a clausal theory is tractable (given upper bounds on the complexity of the theory). This positive result stands in contrast with the negative PAC-learning results obtained for the learning from entailment setting, and is strongly related to the property that all example descriptions are complete.

Second, many techniques in attribute value learning implicitly exploit the independence of the examples. Since learning from interpretations also assumes this independence, such techniques can trivially be upgraded to this setting, whereas an upgrade to learning from entailment, if at all possible, is non-trivial. An example of such a technique is the use of internal validation sets. Setting apart a subset of the training set is easy if all examples are independent; one can choose the subset randomly. When dependences between examples exist, however, it may be necessary to carefully choose the examples that are kept apart, because removing an example from the training set may cause relevant information about another example to be removed. The same problem occurs when one wants to mine a large database by learning from a sample of the database.

We can conclude from this that learning from interpretations offers better opportunities for upgrading techniques from attribute value learning to ILP than learning from entailment.

4.7.4 Limitations of Learning From Interpretations

As already mentioned, the assumption that all the relevant information for a prediction is localized in one example (together with the background knowledge) means that in the learning from interpretations setting one cannot learn predictors that make use of the information in multiple examples. Hence, recursive predicates such as `member` cannot be learned from interpretations. This means the setting is less powerful than learning from entailment.

An interesting observation, however, is that when one takes a look at the practical applications of ILP that have been reported on during the last few years, e.g. at the yearly workshops on ILP (De Raedt, 1996; Muggleton, 1997; Lavrač and Džeroski, 1997; Page, 1998) then it turns out that almost every application involves problems of the kind that can be handled by learning from interpretations. Luc De Raedt mentioned in his invited talk at the *Eighth International Conference on Inductive Logic Programming* that in the literature on ILP he has found only one single application where a recursive rule was found.

These points suggest that in practice the limitations of learning from interpretations are not as bad as they might seem at first sight.

4.8 Related Work

The learning from interpretations has its origin in the non-monotonic learning setting by Helft (1989). De Raedt and Džeroski (1994) showed how it can be used for concept learning and proved that in this setting (under certain assumptions) first order logic hypotheses are PAC-learnable with polynomial time complexity. De Raedt (1997) relates learning from interpretations with other settings within ILP (among which learning from entailment). De Raedt *et al.* (1998) illustrate the practical applicability of the learning from interpretations setting by means of the CLAUDIEN, ICL and TILDE systems. An illustration of the use of nonmonotonic ILP for other than prediction purposes (restructuring a deductive database) is (Blockeel and De Raedt, 1998a).

4.9 Conclusions

In this chapter we have compared different data representation formalisms. We have discussed the attribute value framework, which is essentially propositional, and the inductive logic programming framework, in which first order logic is used to represent hypotheses. Two settings for ILP were discussed: learning from interpretations and learning from entailment. We have compared the different settings with respect to their representational power and their efficiency, and related the representational power to learning from multiple relations in a relational database.

Our main conclusions are that learning from interpretations can be seen as situated somewhere in between the other settings, extending the attribute value framework towards ILP without giving up its efficiency. Due to its favorable position, the learning from interpretations framework is a good choice for upgrading the techniques discussed in the previous chapters to ILP.

Chapter 5

Decision Trees in First Order Logic

5.1 Introduction

Decision trees have mainly been employed within attribute value learning. Due to the focus on logic programming that has dominated the research on relational learning techniques during the latest decennium (in the form of inductive logic programming), relational hypotheses are almost always represented as first order rule sets. A few exceptions exist; we mention KATE (Manago, 1989), STRUCT (Watanabe and Rendell, 1991), ML-SMART (Bergadano and Giordana, 1988), SRT (Kramer, 1996) and TRITOP (Geibel and Wysotzki, 1997). Not all of these systems operate within a strict logical framework; e.g. KATE uses a frame-based representation language. There is also some variation in certain restrictions that are imposed. The decision tree format used by STRUCT and SRT is closest to the one we will propose in this chapter.

While STRUCT and SRT induce relational (also called structural) decision trees, the semantics of such trees have never been addressed explicitly. This is most probably due to the fact that they seem trivial. We show in this chapter that they are not. There are a few peculiarities in the semantics of first order logical decision trees that are easily overlooked, and these have interesting consequences with respect to the representational power of this formalism.

In this chapter we first introduce the notion of a first order logical decision tree. We then discuss how such a tree can be transformed into a rule set, and show that Quinlan's (1993a) method for turning propositional decision trees into rule sets does not work in the first order case. We further discuss the relationship with the flat logic programs that most ILP systems induce and with first order decision lists, and relate this to predicate invention and

induction of logical formulae with mixed quantification.

5.2 Setting

We use the learning from interpretations setting. We recall the problem specification as given in Chapter 4:

Given:

- a target variable Y
- a set of labelled examples E (each example is a Prolog program e labelled with a value y for the target variable)
- a language $\mathcal{L} \in$ Prolog,
- and a background theory B,

Find: a hypothesis $H \in \mathcal{L}$, such that for all labelled examples $(e, y) \in E$,

- $H \wedge e \wedge B \models label(y)$, and
- $\forall y' \neq y : H \wedge e \wedge B \not\models label(y')$

Throughout this chapter we will repeatedly refer to the following example.

Example 5.1 An engineer has to check a set of machines. A machine consists of several parts that may be in need of replacement. Some of these can be replaced by the engineer, others only by the manufacturer of the machine. If a machine contains worn parts that cannot be replaced by the engineer, it has to be sent back to the manufacturer (class `sendback`). If all the worn parts can be replaced, it is to be fixed (`fix`). If there are no worn parts, nothing needs to be done (`ok`).

Given the following set of examples (each example corresponds to one machine) and background knowledge:

Example 1	**Example 2**	**Example 3**	**Example 4**
label(fix)	*label(sendback)*	*label(sendback)*	*label(ok)*
`worn(gear)`	`worn(engine)`	`worn(wheel)`	
`worn(chain)`	`worn(chain)`		

Background knowledge
`replaceable(gear)`
`replaceable(chain)`
`not_replaceable(engine)`
`not_replaceable(wheel)`

a Prolog rule for the `sendback` class is:

```
label(send_back) :- worn(X), not_replaceable(X)
```

◇

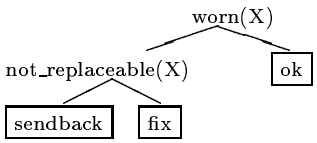


Figure 5.1: Logical decision tree encoding the target hypothesis of Example 5.1.

5.3 First Order Logical Decision Trees

5.3.1 Definition of First Order Logical Decision Trees

Definition 5.1 (FOLDT) *A first order logical decision tree (FOLDT) is a binary decision tree in which*

1. *the nodes of the tree contain a conjunction of literals, and*
2. *different nodes may share variables, under the following restriction: a variable that is introduced in a node (which means that it does not occur in higher nodes) must not occur in the right branch of that node.*

The need for the restriction in part (2) of Definition 5.1 follows from the semantics of the tree. A variable X that is introduced in a node, is existentially quantified within the conjunction of that node. The right subtree is only relevant when the conjunction fails ("there is no such X"), in which case further reference to X is meaningless.

Example 5.2 An example of a logical decision tree is shown in Figure 5.1. It encodes the target hypothesis of Example 5.1. ◇

Figure 5.2 shows how to use FOLDTs for prediction. We use the following notation: a tree T is either a leaf with label y, in which case we write $T =$ **leaf**(y), or it is an internal node with conjunction *conj*, left branch *left* and right branch *right*, in which case we write $T =$ **inode**(*conj, left, right*).[1]

Because an example e is a Prolog program, a test in a node corresponds to checking whether a query $\leftarrow C$ succeeds in $e \wedge B$ (with B the background knowledge). Note that it is not sufficient to define C as the conjunction *conj* in the node itself. Since *conj* may share variables with nodes higher in the tree, C consists of several conjunctions that occur in the path from the root to the current node. More specifically, C is of the form $Q \wedge conj$, where Q is

[1] This notation is slightly simplified with respect to the one from Chapter 3; in the original notation we would have written $T =$ **inode**(*conj*, {(`true`, *left*), (`false`, *right*)}).

procedure PREDICT(e : **example**) **returns** prediction:
    $Q := true$
    $N :=$ root
    while $N \neq$ **leaf**(y) **do**
        let $N =$ **inode**$(conj, left, right)$
        if $\leftarrow Q \wedge conj$ succeeds in $e \wedge B$
        then $Q := Q \wedge conj$
            $N :=left$
        else $N := right$
    return y

Figure 5.2: Making a prediction for an example using a FOLDT (with background knowledge B).

the conjunction of all the conjunctions that occur in those nodes on the path from the root to this node where the left branch was chosen. We call $\leftarrow Q$ the *associated query* of the node.

When an example is sorted to the left, Q is updated by adding *conj* to it. When sorting an example to the right, Q need not be updated: a failed test never introduces new variables.

Example 5.3 If in Figure 5.1 an example is sorted down the tree, in the node containing `not_replaceable(X)` the correct test to be performed is `worn(X), not_replaceable(X)`; it is not correct to test `not_replaceable(X)` on its own. ◇

5.3.2 Semantics of FOLDTs

Figure 5.3 shows how an equivalent logic program can be derived from a FOLDT. Where we use the term "logic programs" in this chapter, it refers to *normal* logic programs, i.e. programs that may contain negative literals in the body of clauses. When the latter is not allowed, we will explicitly refer to *definite* logic programs.

With each internal node in the tree a clause defining a newly invented nullary predicate is associated, as well as a query. This query can make use of the predicates defined in higher nodes. With leaves only a query is associated, no clause.

The queries are defined in such a way that the query associated with a node succeeds for an example if and only if that node is encountered when that example is sorted down the tree. Therefore, if a query associated with a

procedure ASSOCIATE(T : foldt, $\leftarrow Q$: query):
    if $T = \mathbf{inode}(conj, left, right)$ **then**
        assign a unique predicate p_i to this node
        assert $p_i \leftarrow Q, conj$
        ASSOCIATE($left$, $(\leftarrow Q, conj)$)
        ASSOCIATE($right$, $(\leftarrow Q, \neg p_i)$)
    else
        let $T = \mathbf{leaf}(k)$
        assert $label(k) \leftarrow Q$

procedure DERIVE_LOGIC_PROGRAM(T: foldt):
    ASSOCIATE(T, $\leftarrow$)

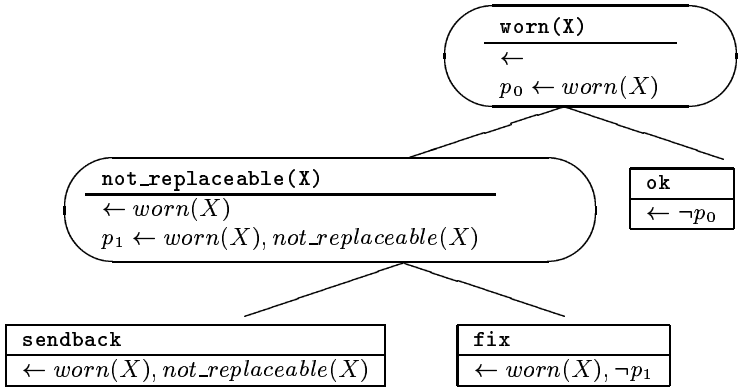
Figure 5.3: Mapping FOLDT's onto logic programs.

leaf succeeds, the leaf indicates the label of the example. The clauses define invented predicates that are needed to express these queries. Thus the queries associated with leaves, together with these clauses, form a logic program that is equivalent to the tree.

An important point is that the algorithm adds to a query the negation of the invented predicate p_i, and not the negation of the conjunction itself (see Figure 5.3: the query $\leftarrow Q, \neg p_i$ and not $\leftarrow Q, \neg conj$ is associated with the right subtree of T). Indeed, the queries of the left and right subtree should be complementary: for each example sorted into this node (i.e. $\leftarrow Q$ succeeds), exactly one of both queries should succeed. Now, $\leftarrow Q, conj$ (which is equivalent to $\leftarrow Q, p_i$) and $\leftarrow Q, \neg p_i$ are complementary, but $\leftarrow Q, conj$ and $\leftarrow Q, \neg conj$ are not, when $conj$ shares variables with Q. For instance, in the interpretation $\{q(1), p(1), q(2)\}$ both $\leftarrow q(X), p(X)$ and $\leftarrow q(X), \neg p(X)$ succeed.

Example 5.4 Figure 5.4 shows the result of applying the algorithm in Figure 5.3 to our running example. Consider in Figure 5.4 the node containing `not_replaceable(X)`. The query associated with the right subtree of this node contains $\neg$`p1` and not $\neg$`not_replaceable(X)`. Indeed, in the latter case the query would succeed if there is a worn part in the machine that is replaceable, while it ought to succeed if there are worn parts in the machine, but *all* of them are replaceable. ◇

Because a literal and its negation are not complementary, adding a literal is not equivalent to adding the negation of the literal while at the same time switching the branches. This means it may be interesting to allow negated literals in queries.



p_0	$\leftarrow$	worn(X)
p_1	$\leftarrow$	worn(X), not_replaceable(X)
label(sendback)	$\leftarrow$	worn(X), not_replaceable(X)
label(fix)	$\leftarrow$	worn(X), $\neg p_1$
label(ok)	$\leftarrow$	$\neg p_0$

Figure 5.4: The tree of Figure 5.1, with associated clauses and queries added; and the logic program derived from the tree.

Example 5.5 In our running example, `not_replaceable(X)` partitions the set $\{e_1, e_2, e_3\}$ (see Example 5.1) into $\{\{e_1\}, \{e_2, e_3\}\}$. `replaceable(X)` would partition it into $\{\{e_1, e_2\}, \{e_3\}\}$. ◇

This is an important difference with the propositional case, where a test (e.g. $X < 5$) and its negation ($X \geq 5$) always generate the same partition. In the first order context they may generate different partitions. This fact and its influence on the tree-to-ruleset conversion are new findings that have not been mentioned in existing literature on relational decision trees (Watanabe and Rendell, 1991; Kramer, 1996), but are important for a correct understanding of their semantics.

5.3.3 On the Expressiveness of FOLDTs and Other Formalisms

While the logic program that is equivalent to a decision tree typically contains auxiliary predicates p_i, in Prolog these can be avoided by using the cut operator. We then get a first order decision list (FODL):

```
label(sendback) :- worn(X), not_replaceable(X), !.
label(fix) :- worn(X), !.
label(ok).
```

In general, a tree can always be transformed into a decision list and vice versa. The following functions dl and tr define such mappings (@ represents concatenation of lists):[2]

$dl(T) = dl'(T, true)$
$dl'(\mathbf{leaf}(c), PC) = [(\texttt{label(}c\texttt{) :- } PC\texttt{, !})]$
$dl'(\mathbf{inode}(conj, left, right), PC) = dl'(left, (PC, conj))@dl'(right, PC)$

$tr([(\texttt{label(}c\texttt{) :- } conj\texttt{, !})|Rest] = \mathbf{inode}(conj, \mathbf{leaf}(c), tr(Rest))$
$tr([\texttt{label(}c\texttt{)}]) = \mathbf{leaf}(c)$

This establishes the equivalence with respect to expressiveness of FODLs and FOLDTs. We now turn to the relationship with logic programs.

We consider only non-recursive logic programs as hypotheses. The hypotheses essentially define the target predicate *label*. They may also define *invented* predicates which do not occur in the background theory. We assume here that the background is not changed by the learner and not part of the

[2] In *tr*, we make use of the fact that nodes can contain conjunctions of literals. If only one literal is allowed in each node, the conversion is still possible but more complex.

hypothesis. Then the hypotheses H_1 and H_2 are equivalent if and only if for all backgrounds B, $B \wedge H_1$ assigns the same label to any possible example as $B \wedge H_2$. We call a hypothesis in the form of a logic program *flat* if it contains no invented predicates, otherwise we call it *layered*.

A hypothesis in the form of a layered *definite* logic program can always be transformed into a flat definite logic program by unfolding calls to invented predicates. Layered *normal* logic programs, however, cannot always be transformed to flat normal logic programs in this way. Unfolding a negative literal for an invented predicate may introduce universally quantified variables in the body, which is beyond the expressive power of logic program clauses (by definition, variables not occurring in the head of a clause are existentially quantified in its body).

Example 5.6 Unfolding the layered logic program of our running example yields:

$$\begin{array}{lll} \texttt{label(ok)} & \leftarrow & \forall X : \neg\texttt{worn}(X) \\ \texttt{label(sendback)} & \leftarrow & \exists X : \texttt{worn}(X) \wedge \texttt{not\_replaceable}(X) \\ \texttt{label(fix)} & \leftarrow & \exists X : \texttt{worn}(X) \wedge \\ & & \forall Y : (\neg\texttt{worn}(Y) \vee \neg\texttt{not\_replaceable}(Y)) \end{array}$$

Since any flat logic program, when written in this format, only contains existential quantifiers (by definition of logic program clauses), no flat hypothesis exists that is equivalent to this theory (e.g. $\forall X \neg\texttt{worn}(X)$ cannot be written with only existential variables).[3] ◇

We conclude from the above that FOLDTs can always be converted to layered normal logic programs (Figure 5.3 gives the algorithm), but not always to flat normal logic programs.

Finally, observe that a flat logic program that predicts only one label for a single example (which is not a restrictive condition in the context of prediction) can always be transformed into an equivalent decision list by just adding a cut to the end of each clause.

As FODLs and FOLDTs can be converted into one another, and flat logic programs can be converted into FODLs or FOLDTs but not the other way around, we have the following property:

> *In the learning from interpretations setting, the set of theories that can be represented by FOLDTs is a strict superset of the set of*

[3] Using \+ to denote Prolog's unsound version of negation as failure (which does not check the groundness of its argument), one might remark that e.g. `label(fix) :- worn(X), \+ (worn(Y), not_replaceable(Y))` correctly computes the label `fix`. However, we do not call this a flat program. Operationally, \+ starts a subquery. Declaratively, the meaning is *label(fix)* $\leftarrow$ *worn(X)*, $\neg(\exists Y(worn(Y), not\_replaceable(Y)))$ which is beyond the expressive power of a normal clause.

theories that can be represented by flat normal logic programs, and is equivalent to the set of theories that can be represented by FODLs.

All this means that systems that induce trees or decision lists (examples of the latter are FFOIL (Quinlan, 1996) and FOIDL (Mooney and Califf, 1995)) can find theories that cannot be found by systems that induce flat (normal or definite) logic programs (e.g. FOIL (Quinlan, 1993b), PROGOL (Muggleton, 1995) and most other ILP systems). This extends the classical claim that the use of cuts allows for a more compact representation (see, e.g., (Mooney and Califf, 1995)) with the claim that also a greater expressivity is achieved. The same expressivity could be achieved by classical ILP systems if they allow negation and perform predicate invention (or if they allow Prolog's unsound negation as failure: \+ with as argument a conjunction of literals, which essentially amounts to the same).

In this respect we want to mention Bain and Muggleton's non-monotonic induction method (Bain and Muggleton, 1992). The hypotheses generated with this method have a structure similar to that of FOLDTs (when only two classes are involved), in that the induced theory is typically also layered through the use of invented predicates, and the invented predicates occur as negative literals in the clauses, accomodating exceptions to them. However, in Bain and Muggleton's framework the learning method is incremental and rule-based.

5.4 Related Work

This chapter is based on (Blockeel and De Raedt, 1998b).

Of the existing decision tree approaches to relational learning (Watanabe and Rendell, 1991; Kramer, 1996; Manago, 1989; Bergadano and Giordana, 1988; Boström, 1995; Geibel and Wysotzki, 1997; Bowers, 1998), STRUCT (Watanabe and Rendell, 1991) and SRT (Kramer, 1996) are most relevant to our approach; they both induce the kind of logical decision trees we have discussed. This work, however, has focused on induction techniques and has ignored the logical and representational aspects of decision trees, needed to fully understand the potential of this technique for first-order learning.

Boström's work (Boström, 1995) has in common with ours that he compared the covering and divide-and-conquer paradigms in the context of ILP. The algorithm he provides employs a divide-and-conquer approach and is in this respect similar to TDIDT. However, with his method the resulting theory is still a flat program (each leaf in the tree that is built contains one clause; by gathering all the leaves that predict one specific class, a logic program is obtained that defines that class).

Peter Geibel's TRITOP system (Geibel and Wysotzki, 1997) induces first order logical decision trees where different nodes cannot share variables. The

approach is still relational because complex conjunctions can occur in individual nodes. One could say that the system constructs propositional features in a relational domain, and uses these features as tests in the tree. The tree itself can thus be seen as propositional. A similar approach, learning from propositionalized first order knowledge, is described in (Kramer *et al.*, 1998).

The machine learning group at Bristol (Lloyd, Flach, Bowers) is studying structural decision trees in the context of the functional programming language Escher. Their approach is similar to ours in that the representation of an example resembles the interpretations we use. However, as in Geibel's approach, different nodes in a tree do not share variables. An early implementation of their relational decision tree learner is described by Bowers (1998).

The results concerning expressivity of trees and rule sets are quite different from those obtained for propositional learning systems. Rivest (1987) has compared the expressivity of DNF formulae, CNF formulae, decision lists and decision trees in the propositional case, and has shown that propositional decision lists are more expressive than the other formalisms, *given a fixed maximum on the length of rules and the depth of the tree*; i.e., the set of theories that can be represented by decision lists with rules of length at most k strictly includes the set of theories that can be represented by decision trees with maximal depth k, as well as the set of theories that can be represented in k-CNF or k-DNF format. Without such complexity bounds, propositional theories can always be transformed into any of the alternative formats, and all formats are equivalent in this sense.

5.5 Conclusions

Earlier in this text we have observed that the TDIDT approach is very successful in propositional learning and differs significantly from the covering approach. This suggests that in ILP, too, TDIDT might have advantages over the covering approach (this point was also raised and investigated by Boström (1995)). In an attempt to make the TDIDT paradigm more attractive to ILP we have investigated the logical aspects of first order decision trees. The resulting framework should provide a sound basis for first order logical decision tree induction.

Our investigation shows that first order logical decision trees are more expressive than the flat non-recursive logic programs typically induced by ILP systems for prediction tasks, and that this expressive power is related to the use of cuts, or the use of negation combined with predicate invention. This in turn relates our work to some of the work on induction of decision lists and predicate invention (Bain and Muggleton, 1992; Quinlan, 1996; Mooney and Califf, 1995), showing that these algorithms too have an expressivity advantage over algorithms inducing flat logic programs. These expressivity results are

specific for the first order case; they do not hold for propositional learning.

Chapter 6

Top-down Induction of First Order Logical Decision Trees

6.1 Introduction

In this chapter we discuss the induction of decision trees in the learning from interpretations setting, as implemented in the TILDE system (Blockeel and De Raedt, 1998b). This chapter builds on Chapter 5, where we introduced first order logical decision trees, and on Chapter 3, where we discussed top-down induction of decision trees in the predictive clustering framework.

We first present the general architecture of TILDE. Next, we discuss the features of the system in more detail. This discussion consists of two parts: first we look at the way in which techniques from propositional learning can be adapted or upgraded towards first order logic (Section 6.3); next, we discuss TILDE as an instantiation of the predictive clustering technique presented before (Section 6.4). After having discussed the algorithms used by TILDE, we illustrate the use of the system with a sample run in Section 6.5. The discussion of the implementation is concluded with some efficiency considerations (Section 6.6).

Section 6.7 consists of an empirical evaluation of TILDE, in which the system is compared with other inductive learners and the effect of certain implementation details is investigated. The focus of this evaluation is on the use of TILDE for classification, although other tasks are considered as well.

The chapter ends with a short discussion of related work and conclusions.

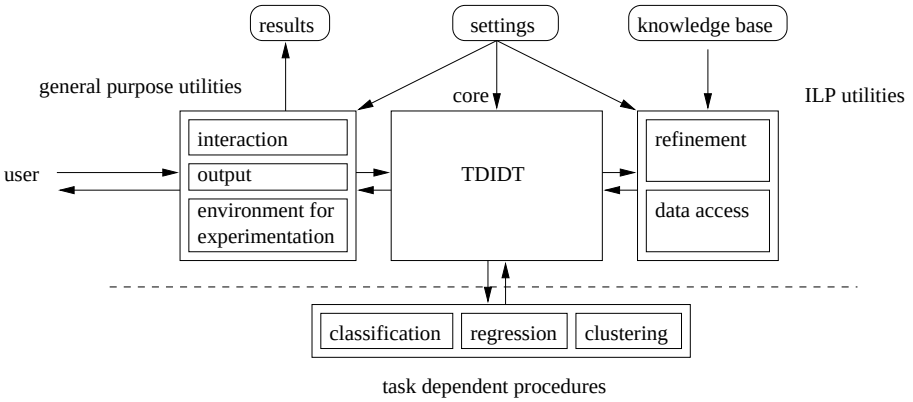


Figure 6.1: Architecture of the TILDE system. Arrows denote information flow.

6.2 Architecture of TILDE

The architecture of the TILDE system[1] is sketched in Figure 6.2.

At the core of the TILDE system is the generic TDIDT algorithm. The auxiliary modules can be divided into three groups:

- One group implements the ILP part of the system: it contains the code for applying a user-defined refinement operator to a clause (in order to generate the set of tests that is to be considered at a node), as well as the code for accessing the data (testing a clause in an example, obtaining the class of an example, ...). The modules in this group make direct use of Prolog functionality.
- A second group implements all the procedures that are specific to certain tasks. This group contains three modules: classification, regression and clustering. Each module instantiates the OPTIMAL_SPLIT, STOP_CRIT, INFO and PRUNE functions referred to by the generic TDIDT algorithm in Figure 3.4, defines quality criteria for trees, a post-pruning method, etc. TILDE always uses exactly one of these three modules, according to the mode it is in.
- A third group contains various auxiliary modules that are not directly related to any specific task or to the ILP character of the system; these include user interaction, procedures for writing trees in a readable format, conversion of trees to Prolog programs, facilities for testing hypotheses (e.g. cross-validation, evaluation on separate test set), etc.

[1] More specifically TILDE2.0. An earlier version, TILDE1.3, only performs classification.

The system takes two types of input: a settings file, specifying the various parameters of the system (which mode it should run in, the language bias, ...), and a knowledge base (divided into a set of examples and background knowledge). It generates one or more output files containing the results of the induction process.

The architecture emphasizes the similarity between the three tasks for which TILDE can be used. The part of the code that is task-specific is clearly isolated and turns out to be relatively small (a rough estimate based on the file sizes suggests this part is about 15% of the total code). The system has been designed so that new modes can be added relatively easily.

The TILDE system is implemented in Prolog, and runs within the MasterProLog engine (the former ProLog-by-BIM)[2]. From the above it will be clear that it is mainly the first group of modules (the ILP part of the code) that fully exploits the functionality offered by Prolog.

6.3 Upgrading TDIDT to First Order Logic

In this section we discuss the adaptations that have been made to the general TDIDT algorithm to upgrade it to the first order context. Figure 6.2 shows the basic TDIDT algorithm, but now in an ILP context, where the tests are first order conjunctions. We first discuss the way in which the computation of tests for nodes is adapted; next, we briefly discuss how a propositional discretization algorithm has been upgraded to the first order context.

6.3.1 Computing the Set of Tests for a Node

The main point where TILDE differs from propositional tree learners is the computation of the set of tests to be considered at a node. To this aim, it employs a user-defined refinement operator under θ-subsumption (Plotkin, 1970; Muggleton and De Raedt, 1994).

Definition 6.1 (θ-subsumption) *A clause c_1 θ-subsumes a clause c_2 (we write this as $c_1 \leq_\theta c_2$) if and only if there exists a variable substitution θ such that $Lits(c_1\theta) \subseteq Lits(c_2)$ with $Lits(c)$ the set of literals occurring in a clause c when it is written as a disjunction.*

Example 6.1 The clause $c_1 : p(X,Y) \leftarrow q(X,Y)$ θ-subsumes the clauses

$c_2 : p(X,Y) \leftarrow q(X,Y), r(X)$
$c_3 : p(X,a) \leftarrow q(X,a)$

[2] A compiled version exists that runs outside the MasterProLog interpreter; this version offers slightly less functionality.

procedure GROW_TREE(T: **tree**, E: **set of examples**, Q: **query**):
    $\leftarrow Q_b$:= OPTIMAL_SPLIT($\rho(\leftarrow Q)$, E)
    if STOP_CRIT($\leftarrow Q_b$, E)
    then T := **leaf**(INFO(E))
    else
        $conj := Q_b - Q$
        $E_1 := \{e \in E | \leftarrow Q_b$ succeeds in $e \wedge B\}$
        $E_2 := \{e \in E | \leftarrow Q_b$ fails in $e \wedge B\}$
        GROW_TREE(*left*, E_1, Q_b)
        GROW_TREE(*right*, E_2, Q)
        T := **inode**(*conj*, *left*, *right*)

procedure TILDE(T: **tree**, E: **set of examples**):
    GROW_TREE(T', E, true)
    PRUNE(T', T)

Figure 6.2: Algorithm for first-order logical decision tree induction.

because $Lits(c_1) \subseteq Lits(c_2)$ (θ is then the empty substitution) and c_3 is obtained by substituting the constant a for the variable Y. $\diamond$

Intermezzo 3 offers some background on θ-subsumption and why it is important in ILP.

Definition 6.2 (Refinement operator) *A refinement operator under θ-subsumption ρ maps a clause c onto a set of clauses, such that $\forall c' \in \rho(c), c \leq_\theta c'$.*

In TILDE the user-defined refinement operator ρ always consists of adding a conjunction to a clause; a variable substitution can be simulated be adding a unification literal (=). In order to refine a node with associated query $\leftarrow Q$, TILDE computes $\rho(\leftarrow Q)$ and chooses that query $\leftarrow Q_b$ in it that results in the best split. The conjunction put in the node consists of $Q_b - Q$, i.e. the literals that have been added to Q in order to produce Q_b.

Specifying the Basic Refinement Operator

The specific refinement operator that is to be used, is defined by the user in a PROGOL-like manner (Muggleton, 1995). A set of specifications of the form `rmode`(n: *conjunction*) is provided, indicating which conjunctions can be added to a query, the maximal number of times the conjunction can be added (n), and the modes and types of the variables in it.

Intermezzo 3: θ-subsumption

θ-subsumption, first defined by Plotkin (1970), is a crucial concept in ILP; it has been studied extensively and almost all ILP systems employ it in one way or another. We summarize the basics.
As Definition 6.1 states, $c_1 \leq_\theta c_2 \Leftrightarrow \exists\theta : Lits(c_1\theta) \subseteq Lits(c_2)$. Example 6.1 in the text suggests that if c_1 θ-subsumes c_2, it also entails it. Indeed, θ-subsumption can be seen as a sound but incomplete version of entailment. An example where c_1 entails c_2 but does not θ-subsume it is

$c_1 = p(Y) \leftarrow p(X), s(X,Y)$
$c_2 = p(Z) \leftarrow p(X), s(X,Y), s(Y,Z)$

Despite this shortcoming, θ-subsumption is popular in ILP because it is much cheaper to compute than entailment. It is useful because it imposes a quasi-order on a set of clauses, and this quasi-order is used to structure the search space.
A *quasi-order* $\leq$ is a reflexive and transitive relation that does not satisfy antisymmetry, i.e. it is possible that $a \leq b$ and $b \leq a$ without a and b being equal. We define the relation $\sim$ as follows: $a \sim b \Leftrightarrow a \leq b \wedge b \leq a$. The $\sim$ relation is symmetric and inherits the reflexivity and transitivity of $\leq$, hence it is an equivalence relation. Each quasi-order thus induces an equivalence relation in its domain, and a partial order on the equivalence classes.
We write the equivalence relation induced by $\leq_\theta$ as $\sim_\theta$. If $a \sim_\theta b$, we call a and b *syntactic variants*. Since θ-subsumption implies entailment, the $\sim_\theta$ relation implies logical equivalence. This means that in a set of syntactic variants only one formula needs to be tested in order to obtain the truth value of every formula in the set; or, in the context of TDIDT: we need generate only one formula for each set of syntactic variants.
A refinement operator under θ-subsumption ρ maps a clause c onto a set of clauses, such that $\forall c' \in \rho(c), c \leq_\theta c'$. A good refinement operator avoids generating clauses that are syntactic variants of one another.
The importance of refinement operators under θ-subsumption in ILP can most easily be seen by looking at how hypotheses are typically induced by machine learning systems. Usually, an inductive learner starts with a coarse hypothesis and generates a series of increasingly more refined hypotheses until an acceptable one is found. Refinement operators under θ-subsumption provide a formalization of this refinement process; they have been studied extensively (e.g. (van der Laag and Nienhuys-Cheng, 1998)).

The mode of an argument is indicated by a $+$, $-$ or $+-$ sign before a variable. $+$ stands for input: the variable should already occur in the associated query of the node where the test is put. $-$ stands for output: the variable has to be one that does not occur yet. $+-$ means that the argument can be both input and output; i.e., the variable can be a new one or an already existing one. Note that the names of the variables in the `rmode` facts are formal names; when the literal is added to a clause actual variable names are substituted for them.

Example 6.2 In the Machines example we introduced in Chapter 5, the following `rmode` specifications could be given:

```
rmode(5: worn(+-V)).
rmode(5: not_replaceable(+V)).
rmode(5: replaceable(+V)).
```

This `rmode` definition tells TILDE that a test in a node may consist of checking whether a component that has already been referred to is worn (e.g. `worn(X)` with `X` an already existing variable), checking whether there exists an worn component in the machine (e.g. `worn(Y)` with `Y` not occurring in the associated query), or testing whether a component that has already been referred to is replaceable. At most 5 literals of a certain type can occur on any path from root to leaf (this is indicated by the 5 in the `rmode` facts).

To make this a bit more concrete, look back at the tree in Figure 5.4. The left child of the root has as associated query $\leftarrow$`worn(X)`. The refinement operator ρ defined by the above specifications generates for this query

$$\begin{aligned}\rho(\leftarrow \texttt{worn(X)}) = \{\ & \leftarrow \texttt{worn(X)}, \texttt{replaceable(X)},\\ & \leftarrow \texttt{worn(X)}, \texttt{not\_replaceable(X)},\\ & \leftarrow \texttt{worn(X)}, \texttt{worn(Y)},\\ & \leftarrow \texttt{worn(X)}, \texttt{worn(X)}\}\end{aligned}$$

where the fourth clause, being in fact equal to the original clause, is filtered out. Hence, the three literals that are considered for this node are `replaceable(X)`, `not_replaceable(X)` and `worn(Y)`. Of these three the second one is preferred by the heuristic and filled in in the node.

◇

A conjunction can have multiple mode declarations, e.g. the following facts specify that at least one of the two arguments of a predicate `inside` has to be input:

```
rmode(5: inside(+V,+-W)).
rmode(5: inside(-V,+W)).
```

Types

TILDE allows the user to specify types for literals. When adding a new literal, the variables in it can only be unified with variables that have the same type. This often results in a huge reduction of the branching factor.

A declaration `typed_language(yes)` tells the system that it should take type information into account. For each predicate the types of its arguments are then given by facts of the form `type(`t_1`, ..., `t_a`)` with a the arity of the predicate. Multiple type specifications can be given for a predicate.

Example 6.3 Suppose the following `rmode`s are given:

```
rmode(5: atom(+-ID, +-Elem, -Charge)).
rmode(5: bond(+ID1,+-ID2,+-BID)).
rmode(5: +Charge<0).
```

If at some point a node with associated query
←`atom(A1,E1,C1), bond(A1, A2, B), atom(A2,E2,C2)`
is to be refined, a $<$ literal can be added in many ways: {`A1` $<$ `0`, `E1` $<$ `0`, `C1` $<$ `0`, `A2` $<$ `0`, `B` $<$ `0`, `E2` $<$ `0`, `C2` $<$ `0`}. Of these only `C1` $<$ `0` and `C2` $<$ `0` are useful: we only want to compare charges with 0, as the other variables are not numerical. The choice of variable names in the `rmode` specifications reflects this; however, while variable names might be suggestive to the user, TILDE does not take them into account. The problem is easily solved by adding type declarations, as follows:

```
typed_language(yes).
type(atom(id, elem, real)).
type(bond(id, id, bondtype)).
type(real<real).
```

TILDE will now only add a `<` literal if the variable to the left is of type `real`, which is only the case for $C1$ and $C2$ in the above example.

Note that these type specifications also have an influence on the number of `atom` and `bond` literals that have to be considered. For instance, since 7 variables occur in the associated query and the first two arguments of `atom` can be unified with already occurring variables or can be new, 64 different literals can be generated for `atom`; using types reduces this to 4. ◇

Generation of Constants

As the above example illustrates, it is possible to use literals with constant parameters instead of variables. Unfortunately, there are many situations where lots of constants can be useful, so that one could have a specification such as

```
rmode(5: +X < -1).
rmode(5: +X < -0.5).
rmode(5: +X < 0).
rmode(5: +X < 0.5).
rmode(5: +X < 1).
...
```

Fortunately, TILDE can generate suitable constants itself. It does this in the following way: for each example in the set of examples covered by a node, a query is run that generates one or more constants. Each of these constants is then filled in in the test that is to be put in this node. In order to keep the branching factor of the search space limited, maxima can be given for the number of examples TILDE should look at, as well as for the number of constants that can be generated from each example. We illustrate the constant generation algorithm with an example.

Example 6.4 We again use the Machines example. To tell TILDE that a test `worn(`c`)` can be used, with c being any constant for which `worn` could possibly succeed, the following `rmode` fact could be used:

```
rmode(5: #(15*5*X: worn(X), worn(X))).
%          a  d c    b        e
```

This specification means that in at most 15 (a) examples, TILDE should run `worn(X)` (b) and see which values `X` (c) can take; it should return at most 5 (d) values per example. Finally, the test `worn(X)` (e) will be put in the node, but with `X` changed into one of the constants: `worn(gear)`, `worn(engine)`, ... ◇

In the above example, the constant generating predicate is the same as the predicate that is to be filled in, but this need not be the case.[3] Another example of the use of constant generation, now in a continuous domain, is:

```
rmode(10: #(100*1*C: boundary(C), +X < C)).
```

In at most 100 examples one numeric boundary will be computed, and a test should consist of comparing an already occurring variable X with this boundary. The computation of a suitable boundary can be defined in background knowledge. It might be done by, e.g., a discretization algorithm (see further).

While the above syntax may be a bit awkward, it is very general and allows the generation of constants in many different settings. It is even possible to generate a whole literal (instead of only its constant parameters), one could

[3] When the generating predicate is the same as the predicate that is to be filled in, this method mimicks the *lazy evaluation* technique as implemented in PROGOL4.4 (Srinivasan and Camacho, 1996).

for instance generate a numerical computation such as `Y is 3.14*X*X` or a function `f(X,Y,Z)` where the definition of `f` is computed at runtime. As another example, the method for predicate invention that is incorporated in PROGOL4.4 (Khan *et al.*, 1998) can also be simulated with the `#`-construct.[4]

Lookahead

An important problem in ILP is that refinement of a clause by adding a single literal may result in little immediate improvement, although the literal may introduce new variables that are very relevant. The following example illustrates the problem.

Example 6.5 In Belgium a person can apply for a driver's license from the age of eightteen. When presented with a suitable set of examples, an inductive learner should be able to come up with the following rule:

```
label(can_apply) :- age(X), X>=18.
```

If the system learns rules top-down, i.e. it starts with an empty body and adds literals to it, then before finding the correct rule it has to generate one of the following rules:

```
label(can_apply) :- age(X).      (1)
label(can_apply) :- X>=18.       (2)
```

Unfortunately, none these rules make any sense. The body of rule (1) succeeds for each and every person (everyone has an age), hence addition of the literal `age(X)` does not yield any improvement over the empty rule. Rule (2) only imposes a constraint on a variable that has no meaning yet. In Prolog such a body always fails, in a constraint logic programming language it might always succeed; but in neither case will it yield any gain. A top-down rule learner may discard both rules because they do not seem to bring it any closer to a solution, thereby failing to find the solution when it is only one step away from it. ◇

While the problem is easiest to explain using a rule-based inductive learner, it also arises for tree-based systems. The problem is inherent to heuristic searches in general. For greedy systems, it may heavily influence the induction process. Although some systems have provisions for alleviating the problem in specific cases (e.g. FOIL (Quinlan, 1993b) automatically adds so-called determinate literals[5], such as the `age` literal in the above example), it has not been solved satisfactorily yet.

[4] This method consists of applying Srinivasan and Camacho's lazy evaluation technique to generate not a constant but a predicate.

[5] Given a clause, a literal to be added to that clause is called determinate if all its free variables can take at most one value.

One technique for coping with the problem is to make the learner *look ahead* in the refinement lattice. When a literal is added, the quality of the refinement can better be assessed by looking at the additional refinements that will become available after this one, and looking at how good these are. This technique is computationally expensive, but may lead to significant improvements in the induced theories.

There are several ways in which lookahead can be performed. One is to look at further refinements in order to have a better estimate for the current refinement. In that case, the heuristic value assigned to a refinement c' of a clause c is a function of c' and $\rho(c')$, with ρ the user-defined refinement operator. ρ itself does not change with this form of lookahead.

A second kind of lookahead is to redefine the refinement operator itself so that the two-step-refinements are incorporated in it. That is, if the original refinement operator (without lookahead) is ρ', then

$$\rho(c) = \rho'(c) \cup \bigcup_{c' \in \rho'(c)} \rho'(c') \tag{6.1}$$

This approach, as well as the former one, can be extended in the sense that the learner could look more than one level ahead.[6]

Example 6.6 Suppose that the test $X < 3$ causes the highest information gain, but that a heuristic using lookahead prefers $X < 4$ because it turns out that the test $X > 2$, in combination with $X < 4$, will lead to a higher gain than any test combined with $X < 3$. The difference between the two lookahead approaches is then that with the first approach, $X < 4$ is simply added as best test, while with the second approach (redefining ρ) one immediately adds $X < 4 \wedge X > 2$.

Both approaches are not equivalent: it is not guaranteed that after adding $X < 4$, the test $X > 2$ will turn out to be the best test to add next. Indeed, although it caused highest gain in combination with $X < 4$, the computation of the new test can use lookahead to find another test that in itself causes less gain but is more promising for further refinements. ◇

The TILDE system follows the second approach. It relies on the user to provide some information about when lookahead is needed, because in many cases the user has a better idea about this than what a learning system can derive on the basis of e.g. determinacy.

[6] One could object that the second approach to lookahead can always be simulated by adding to the background knowledge predicates that are equivalent to the combination of several other predicates. Lookahead would not be needed then. This is true, but if many combinations of literals are possible, then a background predicate must be provided for each combination. Especially when allowing lookahead of more than one level, the number of predicates may become huge. Our lookahead templates offer a much more flexible way of specifying which combinations are possible.

The effect of lookahead can be described as follows. First, we define a basic refinement operator ρ' that takes only the `rmode`s into account:[7]

$$\rho'(H \leftarrow B) = \{H \leftarrow B, C\theta | \quad \begin{array}{l}\texttt{rmode}(n : C) \text{ and} \\ \text{C has been used less than } n \text{ times to form } B \\ \text{and } (H \leftarrow B, C\theta) \text{ is mode- and type-conform } \}\end{array} \tag{6.2}$$

We now extend the operator ρ' so that lookahead is allowed, obtaining a new operator ρ.

The user can provide templates of the form `lookahead(`C_1`,` C_2`)`, specifying that whenever a conjunction is added matching C_1, the conjunction C_2 may be added as well. To formalize this, we define an operator λ that maps a set of clauses onto the set of clauses that is obtained by applying lookahead:

$$\lambda(S) = \{H \leftarrow B, C', C \mid \quad \begin{array}{l}(H \leftarrow B, C') \in S \wedge \exists\theta : C_1\theta = C' \wedge C_2\theta = C \\ \wedge \texttt{lookahead}(C_1, C_2)\}\end{array} \tag{6.3}$$

On the new clauses, lookahead can again be applied if the newly added conjunctions themselves match lookahead specifications, so that finally ρ can be defined as

$$\rho(Q) = \bigcup_{i=1}^{N} \lambda^i(\rho'(Q)) \tag{6.4}$$

where N is the maximum number of lookahead steps allowed.

Example 6.7 In the context of the Mesh data set (see Appendix A for a description), one could have the following specifications:

```
rmode(10:neighbour(+V1, -V2)).
lookahead(neighbour(V1, V2), long(V2)).
```

These would cause TILDE to add, in one refinement step, tests such as (with `E1` a bound variable and `E2` free):

```
neighbour(E1, E2)
neighbour(E1, E2), long(E2)
```

In other words, TILDE can (but need not) test a new edge at the same time it introduces it. ◇

[7] For convenience, we consider the $+$, $-$ and $+-$ symbols that occur in the `rmode` specifications to be meta-information; i.e. we treat e.g. `worn(+V)` as if it were `worn(V)` when applying a substitution, but still take the mode information into account when generating substitutions.

6.3.2 Discretization

Discretization is a technique that is used by symbolic induction methods in order to be able to generate tests on numerical data. It consists of converting a continuous domain into a discrete one. The motivation for discretizing numeric data is twofold and based on the findings in attribute value learning.

On the one hand, there is an efficiency concern. Some ILP systems (e.g., FOIL (Quinlan, 1993b)) generate numbers during the induction process itself, which may cause a lot of overhead: at each refinement step (a lot of) constants need to be generated. By discretizing numeric domains beforehand, as TILDE does, the induction process becomes much more efficient.

On the other hand, by discretizing the data, one may sometimes obtain higher accuracy rates (as the hypothesis is less likely to overfit the training data). Such results have been obtained by, e.g., Catlett (1991).

The discretization procedure that TILDE uses was developed and implemented by Luc De Raedt, Sašo Džeroski and Wim Van Laer (Van Laer *et al.*, 1997) for the ICL system (De Raedt and Van Laer, 1995), but could be incorporated in TILDE without significant modifications. The method only works for classification though; as of now TILDE does not contain any discretization algorithms that are usable for regression or clustering.

The approach followed in TILDE is that the user can identify declaratively the relevant queries and the variables for which the values are to be discretized. For instance, `to_be_discretized(atom(A,B,C,D), [D])` states that the fourth argument of the predicate `atom` should be discretized.

The resulting numeric attributes are then discretized using a simple modification of Fayyad and Irani's method. The details of this method can be found in (Fayyad and Irani, 1993) and (Dougherty *et al.*, 1995). In short, the algorithm finds a threshold that partitions a set of examples into two subsets such that the average class entropy of the subsets is as small as possible, as follows. Let $s(E)$ be the class entropy of a set of examples E:

$$s(E) = -\sum_{i=1}^{k} p(c_i, E) \log p(c_i, E) \tag{6.5}$$

($p(c_i, E)$ is the proportion of examples in E that have class c_i, k is the number of classes). If a threshold T for an attribute A partitions E into $E_1 = \{x \in E | x.A \leq T\}$ and $E_2 = E - E_1$, then the average class entropy after partitioning is

$$\frac{|E_1|}{|E|} s(E_1) + \frac{|E_2|}{|E|} s(E_2). \tag{6.6}$$

The threshold T is chosen so that this average entropy is minimal.[8] This

[8] This corresponds to the threshold that offers maximal information gain, as defined by Equation 3.3.

procedure is applied recursively on E_1 and E_2 until some stopping criterion is reached.

With respect to Fayyad and Irani's algorithm, two adaptations have been made. First, Fayyad and Irani propose a stopping criterion that is based on the minimal description length (MDL) principle, but for both ICL and TILDE this method was found to generate very few thresholds. Therefore TILDE's discretization procedure accepts a maximum number of thresholds as a parameter. This has the additional advantage that one can experiment with different numbers of thresholds.

A second adaptation made to Fayyad and Irani's method specifically concerns non-determinacy. Due to the fact that one example may have multiple or no values for a numeric attribute, we use sum of weights instead of number of examples in the appropriate places of Fayyad and Irani's formulae (in the attribute value case all values have weight 1 as each example has only one value for one attribute). The weight of an example in a set is equal to the number of values occurring in it that pass the test describing the set, divided by the total number of values in the example; it is 0 if there are no values. The sum of the weights of all values for one numeric attribute or query in one example always equals one, or zero when no values are given.

Example 6.8 Consider an example $e_1 = \{p(1), p(2), p(3)\}$, and some threshold $T = 2.5$. If each example had only one value for p, T would partition a set of examples S into S_1 (examples that have a value < 2.5) and S_2, the rest. In our context, e_1 has three values for p, two of which are smaller than 2.5, hence e_1 has weight $2/3$ in S_1, and $1/3$ in S_2. ◇

These weights can then be used in the formulae for class entropy by defining for any set S,

$$|S| = \sum_{e \in S} w_{e,S} \tag{6.7}$$

and defining $p(c_i, E)$ as

$$p(c_i, E) = \frac{|\{e \in E | class(e) = c_i\}|}{|E|} \tag{6.8}$$

Note that the use of weights is more or less arbitrary; other approaches could be followed. We found this approach to work well.

Aside from the generation of thresholds, there is the topic of how these thresholds should be used. We see several possibilities:

- Using inequalities to compare whether a value is less than a discretization threshold; this corresponds to an *inequality* test in the discrete domain.

	continuous domain	discrete domain
inequalities	$x < b_i$	$x_D < d_i$
equalities	$b_i \leq x < b_{i+1}$	$x_D = d_i$
intervals	$b_i \leq x < b_j$	$d_i \leq x_D < d_j$

Table 6.1: Comparison of tests in the continuous domain and the discrete domain.

- Checking whether a value lies in some interval bounded by two consecutive thresholds. Such an interval test corresponds with an *equality* test in the discretized domain.

- Checking whether a value lies in an interval bounded by nonconsecutive thresholds. This corresponds to an *interval* test in the discrete domain.

Table 6.1 gives an overview of the relationship between tests in the continuous domain using discretization thresholds, and tests in the discrete domain. We use the following notation. If x is a value of the continuous domain, x_D is the corresponding value in the discrete domain. The discrete domain is assumed to have n different values which we denote $d_1, \ldots, d_n$; each d_i corresponds to an interval $[b_i, b_{i+1})$ in the continuous domain.

Although allowing only inequality tests is complete (interval tests simply consist of multiple inequality tests), it seems better to explicitly allow interval tests that correspond to discrete equality tests. After all, most learning systems use tests such as $x = c$, and need not generate this test in two steps ($x \leq c \wedge x \geq c$).[9] Allowing intervals that correspond to discrete interval tests is a less obvious decision (not all learners generate interval tests such as $x \in [a, b)$), but seems an interesting option because of the following property. When the number of thresholds is increased, the new set of thresholds is a superset of the previous one. This means that the discrete values of the first discretization correspond to intervals of the second one. By using intervals in the discrete domain with n thresholds, all equality tests for discretizations with a number of thresholds smaller than n are generated as well.

6.4 Instantiations of TDIDT in TILDE

In this section we discuss how TILDE instantiates the procedure parameters of TDIDT for the different induction tasks, as well as some more general procedures that are not task-dependent.

[9] In fact, since an interval or equality test is equivalent to two inequality tests, this can be seen as a special case of lookahead.

6.4.1 Classification Trees

The subsystem of TILDE that induces classification trees is based upon the detailed description of C4.5 in Quinlan's *C4.5: Programs for Machine Learning* (Quinlan, 1993a). More specifically, TILDE inherits from C4.5 the following properties:

- OPTIMAL_SPLIT: the heuristic used for choosing the best split in a given node is by default gain ratio, although information gain can also be chosen by the user (see Section 3.3.1 for definitions). Quinlan (1993a) mentions that gain ratio usually performs slightly better than gain.
- STOP_CRIT: TILDE does not split a node when at least one of the following conditions is fulfilled:
  - the examples covered by it all have the same class,
  - no split can be found that yields any gain at all and for which both branches cover at least some minimal number of examples (by default 2)
- INFO: the information stored in a leaf is the modal class value among the examples covered by the leaf.
- PRUNE: for classification trees, TILDE offers two instantiations of the PRUNE procedure:
  - The C4.5 post-pruning technique, based on an non-empirical estimate of the predictive accuracy of a tree on unseen data (see Section 3.3.4).
  - Pruning based on the predictive accuracy of the tree on an internal validation set, as explained in Section 3.3.4. TILDE's pruning algorithm is shown in Figure 6.3. The QUALITY function is instantiated with the accuracy of a prediction on the validation set.

When running TILDE on a propositional data set, the main difference with C4.5 is that TILDE can only induce binary decision trees (conform to the definition of first order logical decision trees). When run on a propositional data set with only binary features, TILDE usually returns the same trees as C4.5 (small differences can be accounted for by the fact that when there are multiple best tests, TILDE may choose a different best test than C4.5).

6.4.2 Regression Trees

The algorithm for building regression trees is a special case of the clustering algorithm.

```
procedure COMBINE_QUALITIES(T1, T2: tree) returns real:
    w1 := |cov(T1)|
    w2 := |cov(T2)|
    return (w1 max({T1.p, T1.u}) + w2 max({T2.p, T2.u}))/(w1 + w2)

procedure COMPUTE_TREE_QUALITY(T: tree):
    if T = leaf(info)
        T.u := QUALITY(info)
        T.p := T.u
    else
        COMPUTE_TREE_QUALITY(T.left)
        COMPUTE_TREE_QUALITY(T.right)
        T.p := QUALITY(INFO(cov(T)))
        T.u := COMBINE_QUALITIES(T.left, T.right)

procedure CHANGE_NODES_INTO_LEAVES(T: tree):
    if T.p >= T.u then T := leaf(INFO(cov(T)))
    else
        CHANGE_NODES_INTO_LEAVES(T.left)
        CHANGE_NODES_INTO_LEAVES(T.right)

procedure PRUNE_TREE(T: tree):
    COMPUTE_TREE_QUALITY(T)
    CHANGE_NODES_INTO_LEAVES(T)
```

Figure 6.3: Pruning algorithm based on the use of validation sets. The algorithm works in two steps. First, for each node of the tree the quality of the node if it would be a leaf is recorded (p), as well as the quality of the node if it is not pruned but the subtree starting in it is pruned in an optimal way (u). In a second step, the tree is pruned in those nodes where $p > u$. QUALITY is a parameter of the algorithm; it yields the quality of a prediction on the validation set. $cov(T)$ denotes the set of examples in the training set covered by T.

- OPTIMAL_SPLIT: the heuristic used for deciding which split is best, is based on the difference between the mean of the two subsets created by the split. The greater this difference is, the better the split is considered to be. As we mentioned before, maximizing the difference between the means is equivalent to minimizing the variance within the subsets.
- STOP_CRIT: TILDE does not split a node if
  - no split can be found that reduces the variance of the target variable within the subsets and where each branch covers at least some minimal number of examples (by default 2)
  - no split can be found that causes a significant reduction of variance for the target variable. An F-test is used to test this (see Section 3.3.2). By default, the significance level is 1 (which means this test is turned off).
- INFO: the information stored in a leaf is the mean of the values of the examples covered by the leaf.
- PRUNE: the pruning algorithm in Figure 6.3 is used; the QUALITY function returns minus the mean squared error (MSE) of the prediction on the validation set (minus, because maximizing the quality should minimize the MSE).

The regression subsystem of TILDE is also mentioned as TILDE-RT in the literature (RT stands for regression trees). We will also use this term in this text, when we specifically refer to the regression subsystem of TILDE.

6.4.3 Clustering Trees

The algorithm for building clustering trees is very general and highly parametrized. For many applications the user needs to provide some code that defines domain-specific things. The reason why TILDE contains a separate mode for regression, even though it is in all respects a special case of clustering, is that for regression all these parameters can be instantiated automatically. Thus, the regression mode is much more user-friendly than the general clustering mode.

- OPTIMAL_SPLIT: this procedure is parametrized with a distance and a prototype function. By default, TILDE chooses a split such that the distance between the prototypes of the clusters is as large as possible. Two distance measures are predefined in TILDE:
  - Euclidean distance: this distance is only useful when the examples can be represented as points in an n-dimensional space. The user

needs to specify how the coordinates of an example are to be computed from its first order description. A prototype function is also predefined for this distance.

- the distance proposed by Ramon and Bruynooghe (1998): this is a first order distance measure. Jan Ramon implemented this distance and incorporated it in TILDE.

The user can instantiate the OPTIMAL_SPLIT procedure at several levels:

- one of the predefined distances can be chosen
- the user can define a distance, together with a prototype function (e.g., a distance that is specifically designed for the application at hand)
- the OPTIMAL_SPLIT procedure can be redefined entirely. This approach was taken with Ramon's distance, where a prototype function is not available (the prototype of a set can be computed theoretically, but in practice the computation is not feasible). The distance between two sets E_1 and E_2 is here defined as the average distance between all the examples in the sets:

$$D = \text{avg\_dist}(E_1 \times E_2) = \frac{\sum_{e_i \in E_1, e_j \in E_2} d(e_i, e_j)}{|E_1||E_2|} \tag{6.9}$$

Because this computation of D may be expensive,[10] when $E_1 \times E_2$ is large D is estimated by computing the average distance for a randomly chosen sample of fixed size:

$$\hat{D} = \text{avg\_dist}(S) \text{ with } S \subset E_1 \times E_2. \tag{6.10}$$

- STOP_CRIT: TILDE does not split a node if
  - no split can be found that reduces the variance of the subsets and where each branch covers at least some minimal number of examples (by default 2)
  - no split can be found that causes a significant reduction of variance. An F-test is used to test this (see Section 3.3.2). By default, the significance level is 1 (which means this test is turned off).
- INFO: by default, this is the identity function, i.e., the whole set of examples covered by the leaf is stored. This can be redefined by the user.

[10] Computing D has quadratic time complexity in the number of examples; this is to be avoided because the time complexity of the heuristic is crucial for the time complexity of the whole induction process, as we will see in Section 6.6.1.

- PRUNE: the pruning algorithm in Figure 6.3 is used; the QUALITY function returns minus the mean squared distance of the prototype of the covered training examples to the covered examples in the validation set.

The clustering subsystem of TILDE is also referred to as TIC (Top-down Induction of Clustering trees).

6.5 An Example of TILDE at Work

We now illustrate how TILDE works on the Machines example.

Data Format

A data set is presented to TILDE in the form of a set of interpretations. Each interpretation consists of a number of Prolog facts, surrounded by a `begin` and `end` line. Thus, the data for our running example are represented as follows:

```
begin(model(1)).
fix.
worn(gear).
worn(chain).
end(model(1)).

begin(model(2)).
sendback.
worn(engine).
worn(chain).
end(model(2)).

begin(model(3)).
sendback.
worn(wheel).
end(model(3)).

begin(model(4)).
ok.
end(model(4)).
```

The background knowledge is simply a Prolog program:

```
replaceable(gear).
replaceable(chain).
not_replaceable(engine).
not_replaceable(wheel).
```

Settings

The settings file includes information such as whether the task at hand is a classification, regression or clustering task; what the classes are, or which variables are to be predicted; the refinement operator specification; and so on. Most of these settings can be left to their default values for this application. A good settings file is:

```
minimal_cases(1).
classes([fix,sendback,ok]).

rmode(5: replaceable(+-X)).
rmode(5: not_replaceable(+-X)).
rmode(5: worn(+-X)).
```

The `minimal_cases` setting indicates how many examples should at least be covered by each leaf; it is by default 2, but for a small data set such as this one we prefer to make it 1.

Running TILDE with the above settings and input causes it to build a tree as shown in Figure 6.4. The figure shows the output TILDE writes to the screen, as well as a graphical representation of how the tree is built. Each step in the graphical representation shows the partial tree that has been built, the literals that are considered for addition to the tree, and how each literal would split the set of examples. E.g. $fss|o$ means that of four examples, one with class `fix` and two with class `sendback` are in the left branch, and one example with class `ok` is in the right branch. The best literal is indicated with an asterisk. The one that is barred would in principle be generated by the refinement operator, but is filtered out because it generates the same test as the one in the root node and hence is useless.

Output

The (slightly shortened) output file generated by TILDE for the above example is shown in Figure 6.5. It contains some statistics on the induction process and the induced hypothesis, as well as a representation of the hypothesis both as a first order logical decision tree and as a Prolog program.

6.6 Some Efficiency Considerations

6.6.1 Scalability

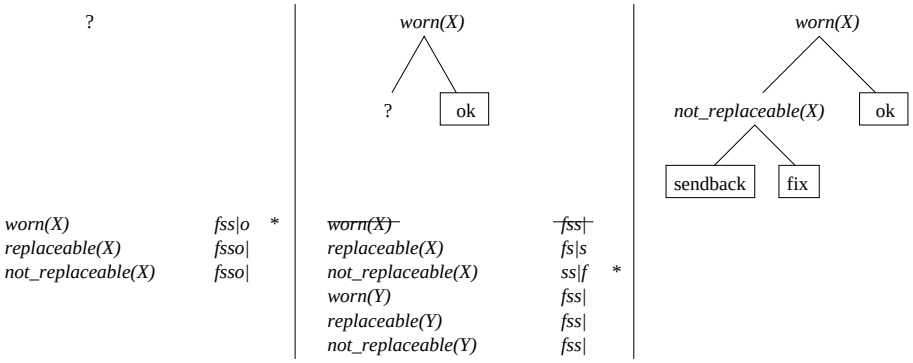
De Raedt and Džeroski (1994) have shown that in the learning from interpretations setting, learning first-order clausal theories is tractable. More specifically,

```
building tree...
true , replaceable(A)
true , not_replaceable(A)
true , worn(A)
[gain = 0.811278,gainratio = 1]
Considering true , worn(A) ...
+ Best test up till now.

Best test: true , worn(A)

(true , worn(A)) , replaceable(A)
[gain = 0.251629,gainratio = 0.274018]
(true , worn(A)) , replaceable(B)
(true , worn(A)) , not_replaceable(A)
[gain = 0.918296,gainratio = 1]
(true , worn(A)) , not_replaceable(B)
(true , worn(A)) , worn(B)
Considering (true , worn(A)) , replaceable(A) ...
+ Best test up till now.
Considering (true , worn(A)) , not_replaceable(A) ...
+ Best test up till now.

Best test: (true , worn(A)) , not_replaceable(A)
```



Refinement operator specification:

```
rmode(5: replaceable(+-X)).
rmode(5: not_replaceable(+-X)).
rmode(5: worn(+-X)).
```

Figure 6.4: TILDE illustrated on the running example. A screen dump of a run is shown, as well as a graphical representation of the tree-building process.

```
** Output of Tilde 2.0 **
Run on droopy : sparc SUNW,Ultra-2 running SunOS 5.6

Settings:
  heuristic(gainratio)
  classes([fix,sendback,ok])
  tilde_mode(classify)
  pruning(c45)
  minimal_cases(1)

Induction time: 0.04 seconds.

--------------------------

after pruning:

pruned_complexity : 2 nodes (2 literals)
pruned_training_accuracy : 1 = 4 / 4
pruned_global_accuracy : 1 = 4 / 4
pruned_C4.5_error_estimation : 2.5 (relative: 0.625)

--------------------------

Compact notation of pruned tree:

worn(A) ?
+--yes: not_replaceable(A) ?
|       +--yes: sendback [2 / 2]
|       +--no:  fix [1 / 1]
+--no:  ok [1 / 1]

--------------------------

Equivalent prolog program:

class(sendback) :- worn(A) , not_replaceable(A), !.
% 2 / 2 = 1.
class(fix) :- worn(A), !.
% 1 / 1 = 1.
class(ok).
% 1 / 1 = 1.
```

Figure 6.5: An output file generated by TILDE (slightly simplified).

given fixed bounds on the maximal length of clauses and the maximal size of literals, such theories are polynomial-sample polynomial-time PAC-learnable. This positive result is related directly to the learning from interpretations setting.

Quinlan (1986) has shown that induction of decision trees has time complexity $O(a \cdot m \cdot n)$ where a is the number of attributes of each example, m is the number of examples and n is the number of nodes in the tree. Since TILDE uses basically the same algorithm as C4.5, it can be expected to inherit the linearity in the number of examples and in the number of nodes. However, there are some differences that can affect its time complexity.

The main difference between TILDE and C4.5, as we already noted, is the generation of tests in a node. The number of tests to be considered in a node depends on the refinement operator. There is no theoretical bound on this, as it is possible to define refinement operators that cause an infinite branching factor (through the use of lookahead). In practice, useful refinement operators always generate a finite number of refinements, but even then this number may not be bounded: the number of refinements typically increases with the length of the associated query of the node (because the number of different ways in which new variables can be unified with already occurring ones depends on the number of variables already occurring). Also, the time for performing one single test on a single example depends on the complexity of that test (it is in the worst case exponential in the number of literals in the test).

A second difference is that Quinlan's derivation exploits the fact that the computation of the quality of a test is linear in the number of examples. This is easy to achieve in the case of classification or regression (e.g. a class entropy or variance can be computed in linear time) but for clustering some care needs to be taken, as for instance the discussion of Ramon's distance in Section 6.4 illustrates.

If there are n nodes, in each node t tests are performed on average, each test is performed on m examples, the average complexity of testing a single example is c and the computation of the heuristic for a single test takes time $h(m)$, then TILDE has time complexity $O(n \cdot t \cdot (m \cdot c + h(m)))$ (assuming the averages c and t exist). If one is willing to accept an upper bound on the complexity of the theory that is to be learned (which was done for the PAC-learning results) and defines a finite refinement operator, both the complexity of performing a single test on a single example and the number of tests are bounded and the averages do exist.

If care is taken that $h(m)$ is $O(m)$, then TILDE's time complexity is $O(n \cdot t \cdot m \cdot c)$. This means that, under fairly general conditions, the time complexity of TILDE is linear in the number of examples. The time complexity also depends on the global complexity of the theory and the branching factor of the refinement operator, which depend on the application domain.

6.6.2 Querying Examples Efficiently

Although TILDE is quite efficient according to ILP standards (see also Section 6.7), it is much less efficient than propositional systems. Preliminary comparisons with C4.5 (results not included in this text) indicate that the latter is often much faster, sometimes in the order of a factor of 1000.

There are many reasons why TILDE cannot be made as efficient as C4.5. Most of them are related to the fact that TILDE is an ILP system. The use of a relational representation for the examples complicates the querying algorithm. The set of tests to be considered at a node is not constant but depends on the node and its associated query, and hence needs to be computed at runtime. Also, performing one single test may take more time.

Another source of inefficiency is that TILDE is implemented in Prolog. Many algorithms incorporated in TILDE are essentially procedural, and Prolog often cannot execute these algorithms as efficiently as when they would be implemented in a lower level language.

There are, however, some points where the basic algorithm can be improved. A rather important one is the following.

As was explained earlier, in order to compute the quality of a test *conj* in a node, queries of the form $\leftarrow Q, conj$ must be sent to every example. However, it is clear that when *conj* does not share any variables with Q, it could be tested on its own. The extra computations involved in computing a variable substitution for Q can be very expensive.

Example 6.9 Suppose that $Q = \texttt{p(X,Y,Z)}, \texttt{q(X,U)}, \texttt{r(U,V)}, \texttt{s(V,Z)}$ and $conj = \texttt{a}$ (a literal of a nullary predicate). It is extremely simple to test whether `a` succeeds in a single example, but if the query $\leftarrow \texttt{p(X,Y,Z)}, \texttt{q(X,U)}, \texttt{r(U,V)}, \texttt{s(V,Z)}, \texttt{a}$ is executed in an example where `a` is false, the underlying Prolog engine backtracks on the `p,q,r,s` literals trying to make `a` succeed. There is no bound on the complexity of this query; if the example description is relatively large, it can be many times more expensive than just testing `a`. ◇

An algorithm has been implemented in TILDE that simplifies queries so that such unnecessary computations are avoided. To this aim, all literals in the clause are collected that are linked with the added conjunction along a path of literals that share variables. The literals in Q that are not collected cannot possibly influence the success of *conj* and hence are absent in the simplified query. The algorithm is shown in Figure 6.6.

6.7 Experimental Evaluation

In this section we describe the experiments performed to evaluate TILDE. We first give some general information, then discuss each experiment in detail.

procedure SIMPLIFY(Q, *conj*) **returns** query:
    V := variables of *conj*
    repeat
        L := literals in Q that contain variables in V
        V := V $\cup$ variables in L
    until V does not change anymore
    Q' := conjunction of all literals in L
    return $\leftarrow Q'$, *conj*

Figure 6.6: Simplification of queries.

6.7.1 Materials

All the experiments described in this text were run on Sun machines under the Solaris operating system. By default experiments were run on a Sun SPARC Ultra-2 at 168 MHz. In some cases a Sun SPARCstation-20 running at 100MHz was used; where this was the case it is mentioned explicitly in the text.

We now give a brief description of the data sets that have been used for our experiments. Detailed descriptions can be found in Appendix A. Most of the data sets are available on the Internet, at either the UCI machine learning data repository (Merz and Murphy, 1996) or the ILP data repository (Kazakov *et al.*, 1996).

- **Soybeans:** this database (Michalski and Chilausky, 1980) contains descriptions of diseased soybean plants. Every plant is described by 35 attributes. A small data set (46 examples, 4 classes) and a large one (307 examples, 19 classes) are available at the UCI data repository. The data sets are mainly used to evaluate clustering algorithms.

- **Iris:** a simple database of descriptions of iris plants, available at the UCI repository. It contains 3 classes of 50 examples each. There are 4 numerical attributes. The set is mainly used for unsupervised learning.

- **Mutagenesis:** this database (Srinivasan *et al.*, 1996), available at the ILP repository (Kazakov *et al.*, 1996), contains descriptions of molecules for which the mutagenic activity has to be predicted. Originally mutagenicity was measured by a real number, but in most experiments with ILP systems this has been discretized into two values (mutagenic or non-mutagenic), making the task a classification task.

  The data set consists of 230 molecules, which are divided into two subsets: regression-friendly (188 molecules) and regression-unfriendly (42

molecules). The term regression here refers to the use of linear regression, not regression trees. The names stem from the fact that experiments with linear regression yielded good results on some of the data but not on all.

- **Biodegradability:** a set of 62 molecules for which structural descriptions and molecular weights are given. The biodegradability of the molecules is to be predicted. This is a real number, but has been discretized into four values (fast, moderate, slow, resistant) in most past experiments. The dataset was provided to us by Sašo Džeroski but is not yet in the public domain.

- **Musk**: the aim is to predict for a set of molecules which ones are musk molecules and which ones are not. Each molecule can have a number of conformations, and the molecule is musk if and only if at least one of its conformations is musk. Introduced to the machine learning community by Dietterich *et al.* (1997), this problem was used to illustrate the so-called multiple-instance problem: an example is not represented by a tuple but by a set of tuples. Multiple-instance problems are hard to cope with for propositional learners.

  The Musk database consists of two data sets, a small one (320K, 476 tuples) and a large one (4.5MB, 6600 tuples). Each tuple consists of 166 numerical attributes. The database is a non-typical ILP database, because of its orientation towards numerical data and the almost propositional representation. It is available at the UCI data repository.

- **Mesh**: this data set, introduced in the ILP community by Dolšak and Muggleton (1992), has its origin in engineering. For many engineering applications, surfaces need to be approximated by a finite element mesh. Such a mesh needs to be fine in some places (in order to assure accuracy), and can be coarser in other places (which decreases the computation cost). The task is to learn rules that predict how fine a mesh should be, by studying a number of meshes.

  The data set consists of descriptions of 5 meshes. It is a typical ILP data set in that it contains structural information and a lot of background knowledge is available. It is available at the ILP data repository (Kazakov *et al.*, 1996).

- **Diterpenes**: a detailed description of this application can be found in (Džeroski *et al.*, 1998). The task is to identify substructures in diterpene molecules by looking at the ^{13}C NMR spectrogram of the molecule (peaks occurring in such a spectrogram may indicate the occurrence of certain substructures). This is a multiple class problem: there are 23 classes.

This problem is inherently relational, but propositional attributes can be defined (so-called engineered attributes) that are highly relevant.

This data set is not in the public domain. The data were kindly provided to us by Steffen Schulze-Kremer and Sašo Džeroski.

6.7.2 Building Classification Trees with TILDE

Aim of the experiment

The aim of this experiment is to compare TILDE's performance with that of other systems for the task of inducing classifiers. We want to investigate several aspects of its performance: the predictive accuracy of the hypothesis that is induced, its interpretability, and the efficiency with which it is induced.

Methodology

We have evaluated the classification subsystem of TILDE by performing experiments on several data sets, and comparing the obtained results with results published in the literature.

For all the experiments, TILDE's default parameters were used; only the refinement operator and number of thresholds for discretization, when applicable, were supplied manually. Full details on the experimental settings, as well as the datasets that were used (except for the Diterpenes dataset, which we cannot make public), are available at

`http://www.cs.kuleuven.ac.be/~ml/Tilde/Experiments/`

All reported results are obtained using ten-fold cross-validations.[11]

Materials

TILDE1.3 was used for these experiments. This is a stable version of TILDE that is available for academic purposes upon request. The data sets are Mutagenesis, Musk and Diterpenes. We chose these data sets because of the availability of published results.

Discussion of the results

Table 6.2 compares TILDE's performance on the Mutagenesis problem with that of FOIL (version 6.2) and PROGOL (actually P-Progol, Srinivasan's implementation in Prolog), as reported in (Srinivasan *et al.*, 1995) (four levels of

[11] Cross-validation is a method for estimating predictive accuracy. Performing an n-fold cross-validation means that a set of data is partitioned into n subsets, and n runs are performed where for each run a different subset is set apart as a test set, while the other $n-1$ subsets form the training set. The predictive accuracy for unseen examples is computed as the average predictive accuracy on the n test sets.

	Accuracies (%)				Times (s)				Complexity (literals)			
	B_1	B_2	B_3	B_4	B_1	B_2	B_3	B_4	B_1	B_2	B_3	B_4
PROGOL	76	81	83	88	117k	64k	42k	41k	24.3	11.2	11.1	9.9
FOIL	61	61	83	82	4950	9138	0.5	0.5	24	49	54	46
TILDE	75	79	85	86	41	170	142	352	8.8	14.4	12.8	19.9

Table 6.2: Accuracies, times and complexities of theories found by PROGOL, FOIL and TILDE for the Mutagenesis problem; averaged over ten-fold cross-validation. Times for TILDE were measured on a Sun SPARCstation-20, for the other systems on a Hewlett Packard 720. Because of the different hardware, times should be considered to be indicative rather than absolute.

Algorithm	% correct
iterated-discrim APR	89.2
GFS elim-kde APR	80.4
TILDE	79.4
backpropagation network	67.7
C4.5	58.8

Table 6.3: Comparison of accuracy of theories obtained with TILDE with those of other systems on the Musk dataset.

background knowledge B_i are distinguished there, each one being a superset of its predecessor, see Appendix A). For the numerical data, TILDE's discretization procedure was used. Lookahead was allowed when adding `bond`-literals (addition of a bond typically does not lead to any gain, but enables inspection of nearby atoms).

From the table it can be concluded that TILDE efficiently finds theories with high accuracy. The complexity of the induced theories is harder to compare, because TILDE uses a radically different format to represent the theory. When simply counting literals, TILDE's theories are about as complex as PROGOL's, but clearly simpler than FOIL's. (Converting the trees towards decision lists and then counting literals yields much larger numbers (respectively 20.4, 34.0, 53.3 and 73.5 literals for B_1 to B_4) due to duplication of many literals, but this comparison method is biased in favor of rule induction systems.) The fact that TILDE finds more compact theories than PROGOL on B_1, although PROGOL performs an exhaustive search (using the A*-algorithm), can be attributed to the greater expressivity of FOLDTs.

With the Musk dataset, the main challenge was its size. We used the largest of the two Musk data sets available at the UCI repository. As we noted before, this data set is not a typical ILP data set (it contains mainly numerical data) but cannot be handled well by propositional learners either. Dietterich *et al.*'s approach (Dietterich *et al.*, 1997) is to adapt propositional learning

	Prop	Rel	Both
FOIL	70.1	46.5	78.3
RIBL	79.0	86.5	91.2
TILDE	78.5 (1.3)	81.0 (1.0)	90.4 (0.6)

Table 6.4: Accuracy results on the Diterpenes data set, making use of propositional data, relational data or both; standard errors for TILDE are shown between parentheses.

algorithms to the multiple-instance problem in the specific case of learning single axis-parallel rectangles (APR's). For ILP systems no adaptations are necessary. Still, TILDE's performance is comparable with most other algorithms discussed in (Dietterich *et al.*, 1997), with only one (special-purpose) algorithm outperforming the others (Table 6.3). For the experiments with TILDE, all the numerical attributes were discretized.[12] The average running time of TILDE on one cross-validation step was about 2 hours.

For the Diterpenes data set, several versions of the data are distinguished: purely propositional data (containing engineered features), relational data (non-engineered), and both. Best performance up till now was achieved by the RIBL system (Emde and Wettschereck, 1996), an instance-based relational learner. Table 6.4 shows that TILDE achieves slightly lower accuracy than RIBL, but outperforms FOIL. Moreover, it returns a symbolic, interpretable (although complex) theory, in contrast to RIBL.

Conclusions

In all these experiments, we have compared TILDE's accuracies with the best known results, which were obtained with different systems. As far as predictive accuracy is concerned, TILDE does not outperform the best systems, but consistently performs almost as well. The complexity of the theories it yields is usually comparable with that of other systems, sometimes better. With respect to efficiency, the system seems to perform very well.

6.7.3 The Influence of Lookahead

Aim

We have argued that lookahead is a useful extension to TILDE; with this experiment we want to validate this claim. We want to investigate not only whether lookahead enables TILDE to find better hypotheses, but also how it affects TILDE's efficiency.

[12] The number of discretization bounds was determined by running experiments on the smaller dataset and choosing the number of bounds that works best on that set.

	Mutagenesis		Mesh	
	accuracy	time	accuracy	time
TILDE, no lookahead	74.6	23s	62.1	36s
TILDE, lookahead	77.0	539s	66.2	563s

Table 6.5: Comparison of TILDE's performance with and without lookahead on the Mutagenesis and Mesh data sets.

Methodology

We have tested the effect of lookahead on several data sets. For each data set we compared TILDE's performance with lookahead to its performance without lookahead.

The datasets repeatedly were partitioned randomly into 10 subsets. Two ten-fold cross-validations were run based on each such partition; one without allowing lookahead, and one with lookahead. For each single partition the accuracy of TILDE with and without lookahead was compared.

Materials

TILDE1.3 was employed for this experiment. We used two ILP data sets: Mutagenesis and Mesh. These two were chosen because they are widely used as ILP benchmarks, and because they contain structural data where properties of neighboring substructures (atoms or edges) are important for classification, but the link to a neighbor itself (`bond` and `neighbour` predicates) provides little or no gain (therefore lookahead is important).

Discussion

In Figure 6.7, each dot represents one partition; dots above the straight line are those partitions where accuracy with lookahead was higher than without lookahead. For both the Mutagenesis and Mesh datasets, lookahead invariably yields an increase in predictive accuracy except in one case (where it stays the same). The hypothesis that lookahead does not yield improvement on these data sets can be rejected at the 1% level.

Table 6.5 compares the average running times needed by TILDE for inducing a single hypothesis. The table confirms that lookahead is computationally expensive, but comparing the times for Mutagenesis with those of PROGOL in Table 6.2 suggests that it is still much cheaper than, e.g., performing an exhaustive search (which PROGOL does).

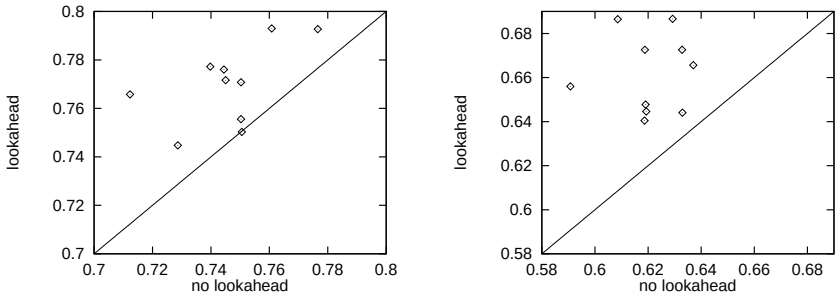


Figure 6.7: Comparison of TILDE's performance with and without lookahead, (a) on the Mutagenesis data; (b) on the Mesh data.

Conclusions

The experiments show that the ability to use lookahead can improve the performance of TILDE. Its computational complexity also increases, but is still acceptable. By letting the user control the lookahead, it is possible to keep the computational complexity to a strict minimum and only use lookahead where it really is useful.

6.7.4 The Influence of Discretization

Aim

The aim of this experiment is to study empirically how discretization influences TILDE's performance with respect to efficiency and predictive accuracy.

Methodology

The most frequently occurring approaches to number handling in symbolic learning consist of allowing tests of the form `V` $<$ v, where v can be either any value in the domain of `V`, or one of a set of thresholds returned by a discretization procedure.

We have run TILDE with these different languages, but also with some alternative languages (allowing equality tests or interval tests), adapted to the specific data set. Each run consists of a ten-fold cross-validation.

Materials

TILDE1.3 was employed for these experiments. We used the Musk and Diterpenes data sets, because both contain non-determinate numerical data, which

makes them fit to test our discretization procedure on.

Discussion

For the Musk data set we tested discretization with inequalities and equalities, for a various number of thresholds. By using discrete equality tests we approximate the setting of Dietterich *et al.* (1997), who learn axis-parallel rectangles. For the Diterpenes data set, inequalities and interval tests were compared, as well as not using any discretization at all, again for a various number of thresholds.

In Figure 6.8, predictive accuracies are plotted against the maximal number of thresholds that was given, for each setting and for each data set. In the Musk domain, the curves for inequalities and equalities are quite similar; using inequalities seems to perform slightly better. In the Diterpenes domain, the effect of discretization is very different according to how the discretization thresholds are used: using interval tests increases performance, while using only inequality tests decreases it. The effect also depends on the number of thresholds that is chosen.

Figure 6.9 shows running times on the Diterpenes dataset. There are no surprises here: discretization yields an efficiency gain, and the number of threshold affects the induction time (linearly when inequality tests are used, quadratically for interval tests; this was expected since the number of intervals increases quadratically with the number of thresholds).

Conclusions

Our conclusions are that the way in which discretization results are used (discrete (in)equalities, intervals) significantly influences the accuracy of the induced theory, as well as the efficiency of the induction process. Using discretization does not guarantee better performance, but may make it possible. It is up to the user to choose a suitable approach.

6.7.5 Regression

Aim

The aim of this experiment is to evaluate the regression subsystem of TILDE.

Methodology

We have evaluated TILDE-RT by running it on data sets and comparing the results with those of other regression systems. Due to limited availability of

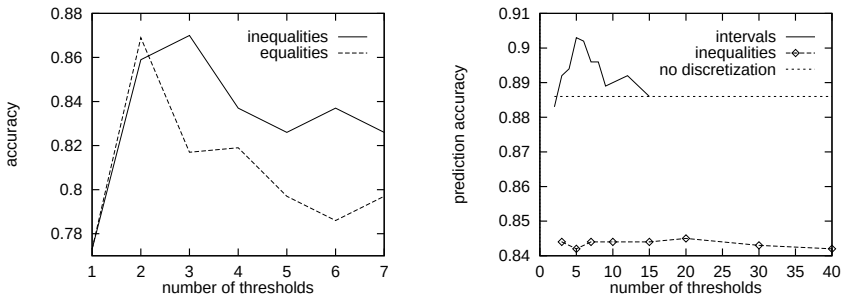


Figure 6.8: Influence of number of thresholds on accuracy: (a) Musk dataset, comparing equalities and inequalities; (b) Diterpenes dataset, comparing intervals with inequalities and no discretization at all.

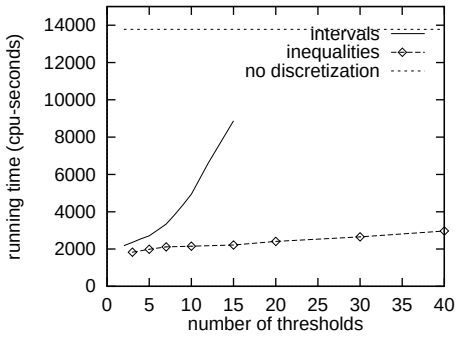


Figure 6.9: Comparison of running times for the different approaches (Diterpenes dataset).

leave-one-out TILDE	classification	acc. = 0.532
leave-one-out TILDE-RT	regression	RE = 0.740
leave-one-out TILDE-RT	classif. via regression	acc. = 0.565
6-fold cross-val. SRT	regression	RE = 0.34
6-fold cross-val. TILDE-RT	regression	RE = 1.13

Table 6.6: Comparison of regression and classification on the biodegradability data. RE = relative error of predictions; acc. = proportion of predictions that are correct.

results on regression in ILP for which the data sets are accessible, this comparison is very limited. We therefore also discretized the predictions afterwards and compare with results obtained by classifiers.

Materials

We used the TILDE-RT algorithm as implemented in TILDE2.0. The system was run on the Biodegradability data set. For each molecule biodegradability was predicted on a logarithmic scale (since there is huge variation in the original values). Given the small size of the data set, no validation set was used; instead we used the F-test stopping criterion (significance level 0.01).

Discussion

Table 6.6 compares TILDE-RT's performance with TILDE's (classification, leave-one-out) and SRT's (regression, sixfold). The SRT experiments were performed by Stefan Kramer.

The comparison with SRT is based on the relative error of the predictions. As the table shows, TILDE-RT scores worse than SRT for this criterion. For comparing TILDE-RT with TILDE, we have discretized TILDE-RT's predictions into four values, corresponding to the four classes that are usually distinguished when the problem is handled as a classification problem. According to this quality criterion, TILDE-RT turns out to score quite well.

Conclusions

These results indicate that there is clearly room for improvement with respect to using TILDE for regression. A possible reason for the fact that it performs worse than SRT is that SRT is specialized in regression and is much more sophisticated. Note that TILDE-RT is just a trivial instantiation of TIC. While SRT's approach is computationally more complex, this certainly seems to pay off with respect to accuracy.

The fact that in a classification setting TILDE-RT scores as high as TILDE is rather surprising, given the comparison with SRT. It suggests that, while TILDE-RT is not optimal with respect to regression, the approach may be competitive with classical approaches to classification, in domains where the classification task is derived from a regression task.[13] These results provide an argument in favor of using regression systems for regression problems, and against turning the regression problem into a classification problem and then using a classifier.

6.7.6 Clustering

Aim

The aim of this experiment is to evaluate the clustering subsystem of TILDE (TIC) with respect to its ability to form (in an unsupervised manner) clusters that are useable for flexible prediction and for classification.

Methodology

We have run TIC on several data sets, comparing its performance with other unsupervised learners. Two evaluation criteria were used: the ability to identify predefined classes (unsupervised classification), and the coherence of the clusters with respect to all attributes (i.e. how well can the values of attributes be predicted if one knows the cluster an instance belongs to).

For all the experiments euclidean distances were computed from all numerical attributes, except when stated otherwise. For the Soybeans data sets all nominal attributes were converted into numbers first. All the reported results were obtained over a ten-fold cross-validation.

For unsupervised classification, the system was evaluated as follows: learning is unsupervised, but classes are assumed to be known at evaluation time (the class of a test example is compared with the majority class of the leaf the example is sorted into).

For flexible prediction, we followed the following strategy: using the training set a clustering tree is induced. Then, all examples of the test set are sorted in this hierarchy, and the prediction for all of their attributes is evaluated. For each attribute, the value that occurs most frequently in a leaf is predicted for all test examples sorted into that leaf.

These experiments were performed in co-operation with Jan Ramon.

[13] One might wonder whether it is not simply the case that TILDE performs badly on this data set. A comparison with other learners on this data set has shown that TILDE's performance is at par with that of most other classification systems. Hence, our conclusions remain valid.

	Soybeans		Iris	
	Avg. accuracy	avg. tree size	Avg. accuracy	avg. tree size
TIC	97%	3.9 nodes	92 %	15 nodes
TILDE	100 %	3 nodes	94%	4 nodes

Table 6.7: Comparing TIC with a supervised learner.

Materials

We used TIC, as implemented in TILDE2.0. Relevant data sets are Soybeans and Iris. These are data sets that have been used for clustering before.

Discussion

Unsupervised Classification: We applied TIC to the small Soybeans and Iris databases, performing ten-fold cross-validations. Learning is unsupervised, but classes are assumed to be known at evaluation time (the class of a test example is compared with the majority class of the leaf the example is sorted into). Table 6.7 compares the results with those obtained by TILDE in a supervised setting.

We see that TIC obtains high accuracies for these problems. The only clustering result we know of is the one of COBWEB, which obtained 100% on the Soybeans data set (this does not differ significantly from TIC's accuracy). TILDE's accuracies don't differ much from those of TIC which induced the hierarchy without knowledge of the classes. Tree sizes are smaller though. This confirms our earlier remark that clustering in the instance space I yields more and smaller clusters than clustering in the prediction space P.

Flexible prediction: We used the large Soybeans database, with pruning. The quality criterion used for the pruning is the intra-cluster variation SS_W (see Equation 3.15).

Table 6.8 summarizes the accuracies obtained for each attribute and compares with the accuracy of majority prediction. The high accuracies show that most attributes can be predicted very well, which means the clusters are very coherent. The mean accuracy of 81.6% is slightly lower than the 83±2% reported by Fisher (1996), but again the difference is not statistically significant.

Note that both this and the previous evaluation method aim at measuring the usefulness of the clustering for predictions. The descriptive quality of the clustering is not measured, although the size of the tree for the Soybeans database indicates that subclusters of the ideal clusters are found, hence from a descriptive point of view the clustering is not optimal. A better stopping criterion or post-pruning method might help here.

name	range	default	acc.	name	range	default	acc.
date	0-6	21.2%	46.3%	plant_stand	0-1	52.1%	85.0%
precip	0-2	68.4%	79.2%	temp	0-2	58.3%	75.6%
hail	0-1	68.7%	71.3%	crop_hist	0-3	32.2%	45.0%
area_damaged	0-3	32.9%	54.4%	severity	0-2	49.2%	63.2%
seed_tmt	0-2	45.6%	51.1%	germination	0-2	32.2%	45.0%
plant_growth	0-1	65.8%	96.4%	leaves	0-1	89.3%	96.4%
leafspots_halo	0-2	49.5%	85.3%	leafspots_marg	0-2	52.2%	86.6%
leafspots_size	0-2	47.8%	87.0%	leaf_shread	0-1	75.9%	81.4%
leaf_malf	0-1	87.3%	88.3%	leaf_mild	0-2	83.7%	88.9%
stem	0-1	54.1%	98.4%	lodging	0-1	80.7%	80.0%
stem_cankers	0-3	58.3%	90.6%	canker_lesion	0-3	49.1%	88.9%
fruiting_bodies	0-1	73.6%	84.3%	external_decay	0-2	75.6%	91.5%
mycelium	0-1	95.8%	96.1%	int_discolor	0-2	86.6%	95.4%
sclerotia	0-1	93.2%	96.1%	fruit_pods	0-3	62.7%	91.2%
fruit_spots	0-4	53.4%	87.0%	seed	0-1	73.9%	85.7%
mold_growth	0-1	80.5%	86.6%	seed_discolor	0-1	79.5%	84.0%
seed_size	0-1	81.8%	88.6%	shriveling	0-1	83.4%	87.9%
roots	0-2	84.7%	95.8%				

Table 6.8: Prediction of all attributes together in the Soybeans data set.

Conclusions

From a predictive clustering point of view, TIC performs approximately as well as other clustering systems. From a descriptive point of view, it performs worse; it tends to find overly specific clusters.

6.7.7 The Effect of Pruning on Clustering

Aim

In this experiment we evaluate the effect of our post-pruning method in TIC. Note that the effect of post-pruning has been studied extensively in the classification and regression context (see e.g. (Quinlan, 1993a; Breiman *et al.*, 1984)), but much less within clustering (an exception is (Fisher, 1996), but the pruning method described there differs significantly from ours). A related task within the clustering field is "cutting" a hierarchical clustering at some level to derive a flat clustering (see, e.g., (Kirsten and Wrobel, 1998)). However, the aim of finding an optimal flat clustering differs from that of finding an optimal hierarchical clustering, so the quality criteria (and hence the techniques) differ.

Methodology

The clustering subsystem of TILDE is run with and without pruning, and the results are compared. Since the effect of pruning might depend on the size of the validation set, we experiment with different validation set sizes. Ten-fold cross-validations are performed in all cases. In each run the algorithm divides the learning set in a training set and a validation set. Clustering trees are built

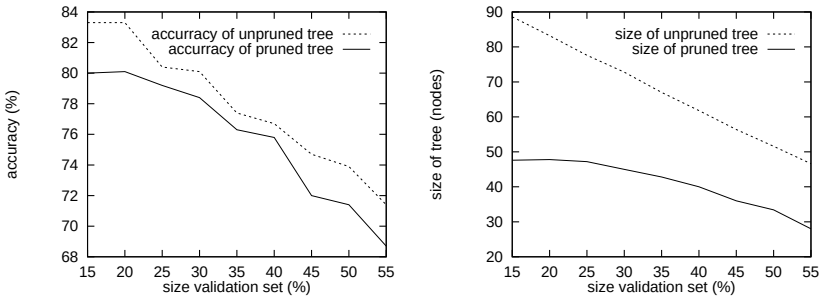


Figure 6.10: Soybeans: a) Accuracy before and after pruning; b) number of nodes before and after pruning.

and pruned in an unsupervised manner. The clustering hierarchy before and after pruning is evaluated by trying to predict the class of each of the examples in the test set (i.e., during evaluation the classes are assumed to be known).

This experiment was performed in co-operation with Jan Ramon.

Materials

We have applied TIC, as implemented in TILDE2.0, to two databases: large Soybeans and Mutagenesis. For the Mutagenesis experiments, the tests allowed in the trees can make use of structural information only (Background B_2), though the heuristics (the distances) make use of the numerical information included in Background B_3.

Discussion

In Figure 6.10, the average accuracy of the clustering hierarchies before and after pruning is plotted against the size of the validation set (this size is a parameter of TIC), and the same is done for the tree complexity. The same results for the Mutagenesis database are summarized in Figure 6.11.

We see that for Soybeans TIC's pruning method results in a slight decrease in accuracy but a large decrease in the number of nodes. The pruning strategy seems relatively stable w.r.t. the size of the validation set. The Mutagenesis experiment confirms these findings (though the decrease in accuracy is less clear here).

Conclusions

These experiments show that our post-pruning method for predictive clustering is useful: it decreases the size of the tree without its accuracy suffering from this too much.

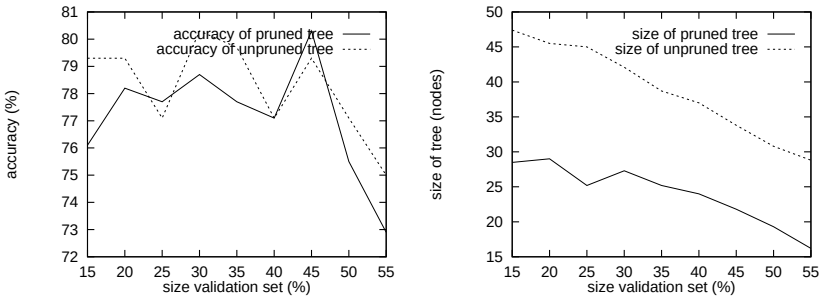


Figure 6.11: Mutagenesis: Accuracy and size of the clustering trees.

6.7.8 Handling Missing Information

Aim

Earlier we have explained how clustering based predictive induction allows for more robust induction than the classical induction of classification trees, because the heuristics can make use of other information than only the target variable. More specifically, when many class values are missing or noisy, the "clustering in I" technique should yield better theories than the "clustering in P" technique. With this experiment we try to validate this claim.

Methodology

We have experimented with leaving out information from the Mutagenesis data set. In one experiment only class information is used to compute the distance, in another experiment all three numerical variables available in Background B_3 are used. Like for the pruning experiment, the tests allowed in the trees only make use of Background B_2.

Information was left out from the data as follows: if the proportion of available information is desired to be p, for each example and for each variable there is a probability p that the value is available. The presence of information on one variable is thus independent of the presence of information on another variable.

Materials

A predecessor of the current TIC called C0.5 (De Raedt and Blockeel, 1997), was used for these experiments. Only one data set is used: Mutagenesis.

available numerical data	logmutag	all three
100%	0.80	0.81
50%	0.78	0.79
25%	0.72	0.77
10%	0.67	0.74

Table 6.9: Classification accuracies obtained for Mutagenesis with several distance functions, and on several levels of missing information.

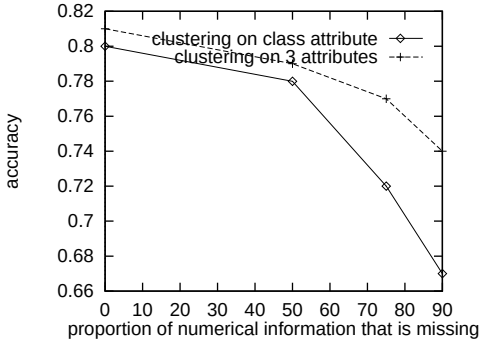


Figure 6.12: Evolution of predictive accuracy in the presence of missing values.

Discussion

Table 6.9 and Figure 6.12 show how the performance of the system degrades with a decreasing amount of information. It can be seen that performance degrades less quickly when 3 numerical variables are used by the heuristics, than when only the class information is used. Note that this experiment is similar in spirits to the ones performed with COLA (Emde, 1994). A difference is that COLA performs unsupervised clustering, while TILDE can use class information when it is available (and hence is a bit more informed than COLA).

Conclusions

This experiment confirms our hypothesis that the use of other than just class information may cause a system to perform better in the presense of missing class values.

6.8 Related work

This chapter compiles and extends the descriptions of TILDE and its auxiliary algorithms that appeared in (Blockeel and De Raedt, 1998b), (Blockeel and

De Raedt, 1997) and (Blockeel *et al.*, 1998b). Most experimental results have also appeared in these papers. (Džeroski *et al.*, 1998) perform an extensive comparison of different ILP systems on the Diterpenes data set, which contains some experiments performed with TILDE that are not included in this text. The conclusions in that article are similar to ours. Ramon and Bruynooghe (1998) have performed experiments with TIC using their first-order distance and report promising results. The TILDE-RT system has also been used for experiments with reinforcement learning in structured domains (Džeroski *et al.*, 1998).

We have discussed how TILDE upgrades propositional techniques to the first order logic context. This upgrading methodology is not specific for TILDE, nor for induction of decision trees. It can also be used for rule induction, discovery of association rules, and other kinds of discovery in the learning from interpretations setting. Systems such as ICL (De Raedt and Van Laer, 1995) and WARMR (Dehaspe and De Raedt, 1997) are illustrations of this. Both learn from interpretations and upgrade propositional techniques. ICL learns first order rule sets, upgrading the techniques used in CN2, and WARMR learns a first order equivalent of association rules ("association rules over multiple relations"). WARMR has been designed specifically for large databases and employs an efficient algorithm that is an upgrade of APRIORI (Agrawal *et al.*, 1996).

TILDE is of course strongly related to other tree induction systems. As mentioned, it borrows techniques from C4.5 (Quinlan, 1993a) and CART (Breiman *et al.*, 1984). The difference with these systems is that TILDE works with first order representations. In this respect it is closer to STRUCT (Watanabe and Rendell, 1991) and SRT (Kramer, 1996). The main differences with these systems are that TILDE learns from interpretations, is more general (in the sense that the other systems focus on classification, respectively regression, while TILDE can do both and in addition perform unsupervised learning and flexible prediction) and that its structure emphasizes the similarities between a) propositional learning and learning from interpretations and b) different induction tasks. Moreover, TILDE returns hypotheses with a clear semantics (as explained in Chapter 5), in contrast with STRUCT and SRT where this issue is not discussed. Finally, TILDE makes use of modern ILP technology (discretization, lookahead, ...) and well-understood techniques originating in propositional learning.

A well-known first-order clusterer is KBG (Bisson, 1992a). Differences are that KBG uses a fixed first-order similarity measure, and that it is an agglomerative clustering algorithm while TILDE is a divisive one. The divisive nature of TILDE makes it as efficient for clustering as classical TDIDT algorithms.[14]

[14] Assuming that distances between clusters can be computed efficiently, i.e. with time complexity linear in the number of examples.

A final difference with KBG is that TILDE directly obtains logical descriptions of the different clusters through the use of the logical decision tree format. For KBG, these descriptions have to be derived in a separate step because the clustering process only produces the clusters (i.e. sets of examples) and not their description.

Other approaches to first order clustering include Kietz's and Morik's Kluster (Kietz and Morik, 1994), (Thompson and Langley, 1991), (Ketterlin *et al.*, 1995), (Yoo and Fisher, 1991) and (Kirsten and Wrobel, 1998).

The instance-based learner RIBL (Emde and Wettschereck, 1996) is related to TILDE via the link between instance-based learning and predictive clustering that was mentioned in Chapter 2. RIBL uses an advanced first order distance metric that might be a good candidate for incorporation in TILDE.

6.9 Conclusions

In this chapter we have presented the inductive logic programming system TILDE. We have discussed both the way in which it upgrades propositional TDIDT to the first order logic context, and the way in which it generalizes over different induction tasks. TILDE is the first ILP system that can perform all these different tasks. Experiments have shown that the system is fast, according to ILP standards, and usually yields hypotheses of high quality (both with respect to interpretability and correctness). Its main shortcoming seems to be located in the regression subsystem, where a comparison with SRT suggests that it can be improved. We point out that currently the regression subsystem is a trivial instantiation of the clustering subsystem that does not contain any specialized techniques yet; this may account for the difference in performance between SRT and TILDE.

Several factors contribute to the overall good performance of TILDE:

- the success of TDIDT in propositional learning;
- the usefulness of learning from interpretations to upgrade propositional techniques;
- the fact that, thanks to the locality assumption, learning from interpretations allows to partially preserve the efficiency of propositional learning;
- the fact that the locality assumption does not severely limit the practical applicability of the system.

A stable version of TILDE, TILDE1.3, is available for academic purposes (see `http://www.cs.kuleuven.ac.be/~ml/Tilde/`). TILDE1.3 only induces classification trees, no regression or clustering trees. TILDE2.0, which does incorporate these features, is only a prototype and not publicly available yet.

Chapter 7

Scaling up TILDE Towards Large Data Sets

7.1 Introduction

Scalability is an important issue in data mining: a good data mining system needs to be able to handle large data sets. In the previous chapters we have shown that it is possible to upgrade techniques and efficiency results from propositional learning to inductive logic programming by learning from interpretations. By upgrading Quinlan's result on the time complexity of learning decision trees, we have shown that TILDE can be expected to scale up well, at least as far as time complexity is concerned.

However, time complexity is not the only limitation on the size of the data sets that an algorithm can handle; memory restrictions also need to be taken into consideration. Even though at first sight memory may seem to be becoming less important, as current hardware often offers hundreds of megabytes of internal memory, the size of the databases that people want to mine also grows, and it does not seem realistic to say that within a few years machines will exist with so much internal memory that they can load a whole database at once.

Still, loading all data at once is what most machine learning and data mining systems do nowadays, and ILP systems are no exception to that. In the propositional learning community, people have recently started taking the problem of mining large databases seriously, and have devised algorithms for mining large databases efficiently without having to load huge amounts of data into main memory. Examples are (Agrawal *et al.*, 1996) and (Mehta *et al.*, 1996), where algorithms are given for the discovery of association rules, respectively induction of decision trees, that work with data stored in an external database and minimize the database access that is needed, for instance by doing as many

computations as possible off-line.

In this chapter we discuss a re-implementation of TILDE that is based on the decision tree algorithm proposed by Mehta *et al.* (1996). This work can be seen as one more illustration of how propositional techniques can be upgraded to the first order logic setting in the learning from interpretations framework. It also demonstrates the feasibility of using TILDE for learning from large data sets.

The re-implementation is only a prototype; it focuses on classification and does not provide all the functionality of the original TILDE system. It is worked out sufficiently, however, for us to illustrate its feasibility and evaluate it experimentally.

In Section 7.2 we discuss the alternative implementation of TILDE and compare it with the classical implementation. We next discuss the influence the implementation has on the feasibility of certain optimizations (Section 7.3). In Section 7.4 we present some experiments to validate our claims. Some of these experiments involve data sets of 100MB or more, illustrating that, contrary to a seemingly wide-spread belief within the machine learning community, the use of ILP techniques is not confined to small databases. We end the chapter with some related work (Section 7.5) and conclusions (Section 7.6).

7.2 Different Implementations of TILDE

We compare two different implementations of TILDE: one is a straightforward implementation, following closely the TDIDT algorithm. The other is a more sophisticated implementation that aims specifically at handling large data sets; it is based on the work by Mehta *et al.* (1996). Both implementations are described using classification terminology, but the same algorithms can be applied for regression and clustering.

A straightforward implementation: TILDE*classic*

The original TILDE implementation was presented in the previous chapter. In this chapter we will refer to it as TILDE*classic*. This implementation is based directly on the algorithm shown in Figure 6.2, which is the most straightforward way of implementing TDIDT. Noteworthy characteristics are that the tree is built depth-first, and that the best test is chosen by enumerating the possible tests and for each test computing its quality (to this aim the test needs to be evaluated on every single example), as is shown in Figure 7.1. The procedure BEST_TEST is an instantiation of the OPTIMAL_SPLIT procedure in Figure 6.2 for induction of classifiers, using the information gain criterion.

Note that with this implementation, it is crucial that fetching an example from the database in order to query it is done as efficiently as possible, because

```
/* WACE = weighted average class entropy */
function WACE(A, B: array[class] of natural) returns real:
    T_A := Σ_c A[c]
    T_B := Σ_c B[c]
    S := Σ_c A[c] log(A[c]/T_A) + Σ_c B[c] log(B[c]/T_B)
    return S/(T_A + T_B)

procedure BEST_TEST(S: set of queries, E: set of examples) returns query:
    for each refinement ← Q_i in S:
        /* counter[true] and counter[false] are class distributions,
           i.e. arrays mapping classes onto their frequencies */
        for each class c : counter[true][c] := 0, counter[false][c] := 0
        for each example e in E:
            if ← Q_i succeeds in e
            then increase counter[true][class(e)] by 1
            else increase counter[false][class(e)] by 1
        s_i := WACE(counter[true], counter[false])
    Q_b := that Q_i for which s_i is minimal /* highest gain */
    return Q_b
```

Figure 7.1: Computation of the best test Q_b in TILDE*classic*.

this operation is inside the innermost loop. For this reason, TILDE*classic* loads all data into main memory when it starts up. Localization is then achieved by means of the module system of the Prolog engine in which TILDE runs. Each example is loaded into a different module, and accessing an example is done by changing the currently active module, which is a very cheap operation. Testing an example involves running a query in the small database consisting of the example together with the background knowledge; the data about other examples is not visible.

An alternative is to load all the examples into one module; no example selection is necessary then, and all data can always be accessed directly. The disadvantage is that the relevant data need to be looked up in a large set of data, so that a good indexing scheme is necessary in order to make this approach efficient. We will return to this alternative in the section on experiments.

We point out that, when examples are loaded into different modules, TILDE-*classic* partially exploits the locality assumption (in that it handles each individual example independently from the others, but still loads all the examples in main memory). It does not exploit this assumption at all when all the examples are loaded into one module.

A more sophisticated implementation: TILDE*LDS*

Mehta *et al.* (1996) have proposed an alternative implementation of TDIDT that is oriented towards mining large databases. With their approach, the database is accessed less intensively, which results in an important efficiency gain. We have adopted this approach for an alternative implementation of TILDE, which we call TILDE*LDS* (*LDS* stands for *Large Data Sets*).

The alternative algorithm is shown in Figure 7.2. It differs from TILDE-*classic* in that the tree is now built breadth-first, and examples are loaded into main memory one at a time.

The algorithm works level-wise. Each iteration through the **while** loop will expand one level of the decision tree. S contains all nodes at this level of the decision tree. To expand this level, the algorithm considers all nodes N in S. For each refinement in each node, a separate counter (to compute class distributions) is kept. The algorithm makes one pass through the data, during which for each example that belongs to a non-leaf node N it tests all refinements for N on the example and updates the corresponding counters.

Note that while for TILDE*classic* the example loop was inside the refinement loop, the opposite is true now. This minimizes the number of times a new example must be loaded, which is an expensive operation (in contrast with the previous approach where all examples were in main memory and examples only had to be "selected" in order to access them, examples are now loaded from disk). In the current implementation each example needs to be loaded at most once per level of the tree ("at most" because once it is in a leaf it need

```
function BEST_TEST(N: node) returns query:
    ← Q := associated_query(N)
    for each refinement ← Q_i of ← Q:
        CD_l := counter[N,i,true]
        CD_r := counter[N,i,false]
        s_i := WACE(CD_l, CD_r)
    Q_b := that Q_i for which s_i is minimal
    return ← Q_b

procedure TILDELDS(T: tree):
    S := {T}
    while S ≠ φ do
        /* add one level to the tree */
        for each example e that is not covered by a leaf node:
            load e
            N := the node in S that covers e
            ← Q := associated_query(N)
            for each refinement ← Q_i of ← Q:
                if ← Q_i succeeds in e
                then increase counter[N,i,true][class(e)] by 1
                else increase counter[N,i,false][class(e)] by 1
        for each node N ∈ S :
            remove N from S
            ← Q_b := BEST_TEST(N)
            if STOP_CRIT(← Q_b)
            then N := leaf(MODAL_CLASS (N))
            else
                ← Q := associated_query(N)
                conj := Q_b − Q
                N := inode(conj, left, right)
                add left and right to S
```

Figure 7.2: The TILDE*LDS* algorithm. The WACE function is defined in Figure 7.1. The STOP_CRIT and MODAL_CLASS functions are the instantiations of STOP_CRIT and INFO for classification as mentioned in Chapter 6.

not be loaded anymore), hence the total number of passes through the data file is equal to the depth of the tree, which is the same as was obtained for propositional learning algorithms (Mehta *et al.*, 1996).

The disadvantage of this algorithm is that a four-dimensional array of counters needs to be stored instead of a two-dimensional one (as in TILDE*classic*), because different counters are kept for each refinement in each node.[1]

Care has been taken to implement TILDE*LDS* in such a way that the size of the data set that can be handled is not restricted by internal memory (in contrast to TILDE*classic*). Whenever information needs to be stored the size of which depends on the size of the data set, this information is stored on disk.[2] When processing a certain level of the tree, the space complexity of TILDE*LDS* therefore has a component $O(r \cdot n)$ with n the number of nodes on that level and r the (average) number of refinements of those nodes (because counters are kept for each refinement in each node), but is constant in the number of examples m. This contrasts with TILDE*classic* where the space complexity has a component $O(m)$ (because all examples are loaded at once).

While memory now restricts the number of refinements that can be considered in each node and the maximal size of the tree, this restriction is unimportant in practice, as the number of refinements and the tree size are usually much smaller than the upper bounds imposed by the available memory. Therefore TILDE*LDS* typically consumes less memory than TILDE*classic*, and may be preferable even when the latter is also feasible.

7.3 Optimizations

The difference in the way the example loop and the refinement loop are nested does not only influence the amount of data access that is necessary. It also has an effect on the kind of optimizations that can be performed.

The way in which refinements and examples are processed in TILDE*classic* and TILDE*LDS* is illustrated in Figure 7.3. The figure shows how TILDE*classic* chooses one refinement, tests it on all examples, then goes on to the next refinement, while TILDE*LDS* chooses one example, tests all refinements on it, then goes on to the next example.

We have seen one opportunity for optimizing queries already: in Section 6.6.2 it was shown how some parts of a query can be removed because they cannot possibly be relevant for this test. While the query simplification algorithm is relatively expensive, it pays off when the query is to be run on many examples.

[1] One of the dimensions fortunately does not depend on the application, as it can only take the values `true` and `false`.

[2] The results of all queries for each example are stored in this manner, so that when the best query is chosen after one pass through the data, these results can be retrieved from the auxiliary file, avoiding a second pass through the data.

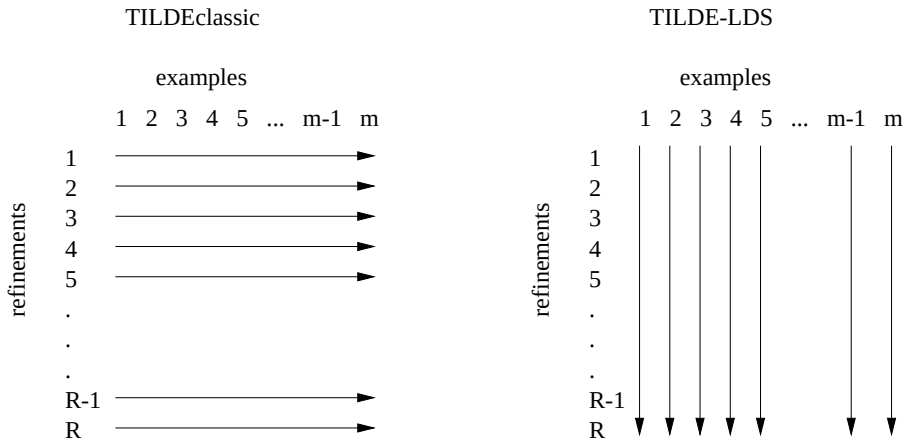


Figure 7.3: The difference between TILDE*classic* and TILDE*LDS* in the way they process the examples and refinements.

We recall that the algorithm changes a refinement $Q, conj_i$ into a simpler form $Q', conj_i$, where Q' only contains those literals in Q that can influence the outcome of $conj_i$. Now, when a query Q is simplified into Q', this Q' can be passed on horizontally in Figure 7.3; only when a new row is started (a new refinement is processed), a new Q' needs to be computed. This is necessary because although Q stays the same, the simplification Q' is computed from both Q and $conj_i$, and $conj_i$ is different for each row.

From Figure 7.3 it is clear that TILDE*classic* can just compute Q' at the beginning of a row and then use it to process the whole row before computing a new one; TILDE*LDS*, on the contrary, needs to recompute Q' for each example. Since the simplification of a query may be more expensive than what is gained by testing the simpler query on a single example (this strongly depends on the application), in some cases it may be better not to use simplified queries. An alternative is to compute each Q' once and store them in an array indexed by refinement identifiers, from which they are retrieved when needed. This seems the only efficient way of using query simplification in TILDE*LDS*.

While query simplification becomes harder to use in TILDE*LDS*, this alternative implementation creates other opportunities for optimization of the querying process. It is now easier to pass information about a single example from one refinement to another (vertically, in Figure 7.3). Note that each test is of the form $Q, conj_i$, so that a large part of the test (Q) is the same for each refinement. Since the refinements are all tested on the same example, all answer substitutions for Q could be computed once, and then for each refinement one only need test $conj_i$, with its variables instantiated according to the

```
/* alternative for the second for each loop in Figure 7.2 */
for each refinement ← Q_i of ← Q:
    conj_i := Q − Q_i
    S := set of all conj_i
    for each answer substitution θ for ← Q:
        for each conj_i in S:
            if conj_iθ succeeds
            then remove conj_i from S
                    increase counter[N, i, true] by 1
    for each conj_i in S: increase counter[N,i,false] by 1
```

Figure 7.4: Interleaved computation of answer substitutions for Q and the success of each refinement on a single example.

answer substitutions for Q.

We refer to this technique of storing answer substitutions for later use as *caching*. Note that this technique is similar to tabling, as used in logic programming. We use the more general term caching because tabling might suggest that partial results are stored and retrieved when needed within the execution of one single query. This is not the case in our implementation; partial results are propagated from one query to another similar one.

The caching technique as described above has the disadvantage that the size of one set of answer substitutions can be very large. Moreover, unless the set is stored in an indexed structure, retrieving answer substitutions still takes time linear in the size of the set. Fortunately, it is possible to avoid storing the substitutions explicitly, by interleaving the computation of answer substitutions θ_j for Q and the computation of the truth value of $conj_i\theta_j$; i.e., starting with a set S of all $conj_i$, the system finds one answer substitution θ_1 for Q, computes truth values for all $conj_i\theta_1$, removes from S all $conj_i$ for which $conj_i\theta_1$ is true, then continues looking for θ_2, applies it to the smaller S, and so on until all answer substitutions for Q have been found. The conjunctions that are still in S at that moment are those that fail on the example. Figure 7.4 gives an algorithm.

This technique provides the advantage of our caching technique (the answer substitutions of Q are only computed once) without having to store a large set of data, and is also faster because a linear search in the list of answer substitutions is avoided. It may incur some overhead, depending on the exact implementation; to make it maximally efficient it should be implemented at the level of the Prolog execution engine.

Note that, in order to use this caching technique in TILDE*classic*, it would be

	TILDE*classic*	TILDE*LDS*
simplified query	easy	feasible
caching	not feasible	easy
both	not feasible	feasible

Table 7.1: Overview of the different optimizations.

necessary to store an array of sets of answer substitutions indexed by example identifiers. Given the potentially large size of one set of answer substitutions and the potentially huge number of examples, this is clearly infeasible.

A summary of all this is given in Table 7.1. Leaving the feasibility of loading all data into main memory out of consideration, the choice between TILDE*classic* and TILDE*LDS* should be based on which optimization is expected to work best: query simplification or caching.

It is possible, but rather complicated, to combine the two optimizations. Instead of one Q, there are then $k \leq R$ different simplified queries Q' for which answer substitutions are to be computed (with R the total number of refinements). A possible procedure is: compute Q' for each refinement, store the different Q'_i together with a set S_i of all the refinement indexes for which they are relevant, and apply the algorithm in Figure 7.4 for each S_i (*mutatis mutandis*; Q becomes Q' etc. in the algorithm).

Whether computing the answer substitutions for k simpler queries is more efficient than computing the answer substitutions for one more complex query, depends on k and on how much simpler the simplified queries are. This is domain-dependent.

The possibility of avoiding recomputation of the answer substitutions of Q is an important advantage of TILDE*LDS*, and may in itself offset the advantage that TILDE*classic* has by computing a simplified query. Moreover, TILDE*LDS* is the only implementation that in principle allows either optimization, or even both together. In their current implementations, TILDE*classic* exploits query simplification, while TILDE*LDS* can employ either the caching technique or the alternative algorithm in Figure 7.4.

We can draw some interesting conclusions from this discussion. First of all, in the propositional learning field the difference between the classical version of TDIDT and the level-wise version is not so important, except when the data set is not loaded in main memory. In the ILP context, this is quite different: due to the interaction between the conjunction in a node and the associated query of a node, optimization opportunities are very different for both implementations (and it seems that the level-wise version offers the better opportunities).

Second, the optimizations we have identified should also be applicable to other ILP systems. Many ILP systems are structured so that the example loop is inside the refinement loop. Switching the loops might yield significant

speed-ups in some cases. This certainly applies to systems that learn from interpretations (CLAUDIEN (De Raedt and Dehaspe, 1997), ICL (De Raedt and Van Laer, 1995)). For systems learning from entailment the optimization seems harder to accomplish but is probably not impossible; a system such as PROGOL (Muggleton, 1995) might benefit from it.

7.4 Experiments

In this experimental section we try to validate our claims about time complexity empirically, and explore some influences on scalability. More specifically, we want to:

- validate the claim that when the localization assumption is exploited, induction time is linear in the number of examples (*ceteris paribus*, i.e., we control for other influences on induction time such as the size of the tree);
- study the influence of localization on induction time (by quantifying the amount of localization and investigating its effect on the induction time);
- investigate how the induction time varies with the size of the data set in more practical situations (if we do *not* control other influences; i.e. a larger data set may cause the learner to induce a more complex theory, which in itself has an effect on the induction time).

These experiments have been performed in co-operation with Nico Jacobs.

7.4.1 Data Sets

We briefly describe the data sets that are used in these experiments; for a more detailed description see Appendix A.

RoboCup

This set contains data about soccer games played by software agents training for the RoboCup competition (Kitano *et al.*, 1997). It contains 88594 examples and is 100MB large. Each example consists of a description of the state of the soccer terrain as observed by one specific player on a single moment. This description includes the identity of the player, the positions of all players and of the ball, the time at which the example was recorded, the action the player performed, and the time at which this action was executed.

While this data set would allow rather complicated theories to be constructed, for our experiments the language bias was very simple and consisted of a propositional language (only high-level commands are learned). This use of

the data set reflects the learning tasks considered until now by the people who are using it, see (Jacobs *et al.*, 1998). It does not influence the validity of our results for relational languages, because the propositions are defined by the background knowledge and their truth values are computed at runtime, so the query that is really executed is relational. For instance, the proposition `have_ball`, indicating whether some player of the team has the ball in its possession, is computed from the position of the player and of the ball.

Poker

The Poker data sets are artificially created data sets where each example is a description of a hand of five cards, together with a name for the hand (pair, three of a kind, ...). The aim is to learn definitions for several poker concepts from a set of examples. The classes that are considered here are `nought`, `pair`, `two_pairs`, `three_of_a_kind`, `full_house`, `flush` and `four_of_a_kind`. This is a simplification of the real poker domain, where more classes exist and it is necessary to distinguish between e.g. a pair of queens and a pair of kings; but this simplified version suffices to illustrate the relevant topics and keeps learning times sufficiently low to allow for reasonably extensive experiments.

An interesting property of this data set is that some classes, e.g. `four_of_a_kind`, are very rare, hence a large data set is needed to learn these classes (assuming the data are generated randomly).

Mutagenesis

For the description of the Mutagenesis data set we refer to Section 6.7.1 or Appendix A. In these experiments we used Background B_1, i.e., only structural information about the molecules (the atoms and bonds occurring in them) is available.

7.4.2 Materials and Settings

All experiments were performed with the two implementations of TILDE we discussed: TILDE*classic* and TILDE*LDS*. These programs are implemented in Prolog and run under the MasterProLog engine. The hardware we used is a Sun Ultra-2 at 168 MHz, running the Solaris system (except when stated otherwise).

Both TILDE*classic* and TILDE*LDS* offer the possibility to precompile the data file. We exploited this feature for all our experiments. For TILDE*LDS* this raises the problem that in order to load one example at a time, a different object file has to be created for each example (MasterProLog offers no predicates for loading only a part of an object file). This can be rather impractical. For this reason several examples are usually compiled into one object file; a

parameter called *granularity* (G, by default 10) controls how many examples can be included in one object file. Object files are loaded one by one by TILDE*LDS*, which means that G examples (instead of one) at a time are loaded into main memory. Because of this, G has an influence on the efficiency of TILDE*LDS*; in a sense it affects the amount of localization in the data. This effect is investigated in our experiments.

7.4.3 Experiment 1: Time Complexity

Aim of the Experiment

As mentioned before, induction of trees with TILDE*LDS* should in principle have a time complexity that is linear in the number of examples. With our first experiment we empirically test whether our implementation indeed exhibits this property. We also compare it with other approaches where the locality assumption is exploited less or not at all.

We distinguish the following approaches:

- loading all data at once in main memory without exploiting the locality assumption (the standard ILP approach);
- loading all data at once in main memory, exploiting the locality assumption (this is what TILDE*classic* does);
- loading examples one at a time in main memory; this is what TILDE*LDS* does.

To the best of our knowledge all ILP systems that do not learn from interpretations follow the first approach (with the exception of a few systems that access an external database directly instead of loading the data into main memory, e.g. RDT/DB (Morik and Brockhausen, 1996); but these systems still do not make a locality assumption). We can easily simulate this approach with TILDE*classic* by specifying all information about the examples as background knowledge. For the background knowledge no locality assumption can be made, since all background knowledge is potentially relevant for each example.

The performance of a Prolog system that works with a large database is improved significantly if indexes are built for the predicates. On the other hand, adding indexes for predicates creates some overhead with respect to the internal space that is needed, and a lot of overhead for the compiler. The MasterProLog system by default indexes all predicates, but this indexing can be switched off. We have performed experiments for the standard ILP approach both with and without indexing (thus, the first approach in the above list is actually subdivided into "indexed" and "not indexed").

Methodology

Since the aim of this experiment is to determine the influence of the number of examples (and only that) on time and space complexity, we want to control as much as possible other factors that might also have an influence. We have seen in Section 6.6.1 that these other factors include the number of nodes n, the average number of tests per node t and the average complexity of performing one test on one single example c. c depends on both the complexity of the queries themselves and on the example sizes.

When varying the number of examples for our experiments, we want to keep these factors constant. This means that first of all the refinement operator should be the same for all the experiments. This is automatically the case if the user does not change the refinement operator specification between consecutive experiments.

The other factors can be kept constant by ensuring that the same tree is built in each experiment, and that the average complexity of the examples does not change. In order to achieve this, we adopt the following methodology. We create, from a small data set, larger data sets by including each single example several times. By ensuring that all the examples occur an equal number of times in the resulting data set, the class distribution, average complexity of testing a query on an example etc. are all kept constant. In other words, all variation due to the influence of individual examples is removed.

Because the class distribution stays the same, the test that is chosen in each node also stays the same. This is necessary to ensure that the same tree is grown, but not sufficient: the stopping criterion needs to be adapted as well so that a node that cannot be split further for the small data set is not split when using the larger data set either. In order to achieve this, the minimal number of examples that have to be covered by each leaf (which is a parameter of TILDE) is increased proportionally to the size of the data set.

By following this methodology, the mentioned unwanted influences are filtered out of the results.

Materials

We used the Mutagenesis data set for this experiment. Other materials are as described in Section 7.4.2.

Setup of the Experiment

Four different versions of TILDE are compared:

- TILDE*classic* without locality assumption, without indexing
- TILDE*classic* without locality assumption, with indexing

- TILDE*classic* with locality assumption
- TILDE*LDS*

The first three "versions" are actually the same version of TILDE as far as the implementation of the learning algorithm is concerned, but differ in the way the data are represented and in the way the underlying Prolog system handles them.

Each TILDE version was first run on the original data set, then on data sets that contain each original example 2^n times, with n ranging from 1 to 9.

For each run on each data set we have recorded the following:

- the time needed for the induction process itself (in CPU-seconds)
- the time needed to compile the data (in CPU-seconds). The different systems compile the data in different ways (e.g. according to whether indexes need to be built). As compilation of the data need only be done once, even if afterwards several runs of the induction system are done, compilation time may seem less relevant. Still, it is important to see how the compilation scales up, since it is not really useful to have an induction method that scales linearly if it needs a preprocessing step that scales super-linearly.

Discussion of the Results

Table 7.2 gives an overview of the time each TILDE version needed to induce a tree for each set, as well as the time it took to compile the data into the correct format. Some properties of the data sets are also included.

Note that only TILDE*LDS* scales up well to large data sets. The other versions of TILDE had problems loading or compiling the data from a multiplication factor of 16 or 32 on.

The results are shown graphically in Figure 7.5. Note that both the number of examples and time are indicated on a logarithmic scale. Care must be taken when interpreting these graphs: a straight line does not indicate a linear relationship between the variables. Indeed, if $\log y = n * \log x$, then $y = x^n$. This means the slope of the line should be 1 in order to have a linear relationship, while 2 indicates a quadratic relationship, and so on. In order to make it easier to recognize a linear relationship (slope 1), the function $y = x$ has been drawn on the graphs as a reference; each line parallel with this one indicates a linear relationship.

The graphs and tables show that induction time is linear in the number of examples for TILDE*LDS*, for TILDE*classic* with locality, and for TILDE*classic* without locality but with indexing. For TILDE*classic* without locality or indexing the induction time increases quadratically with the number of examples.

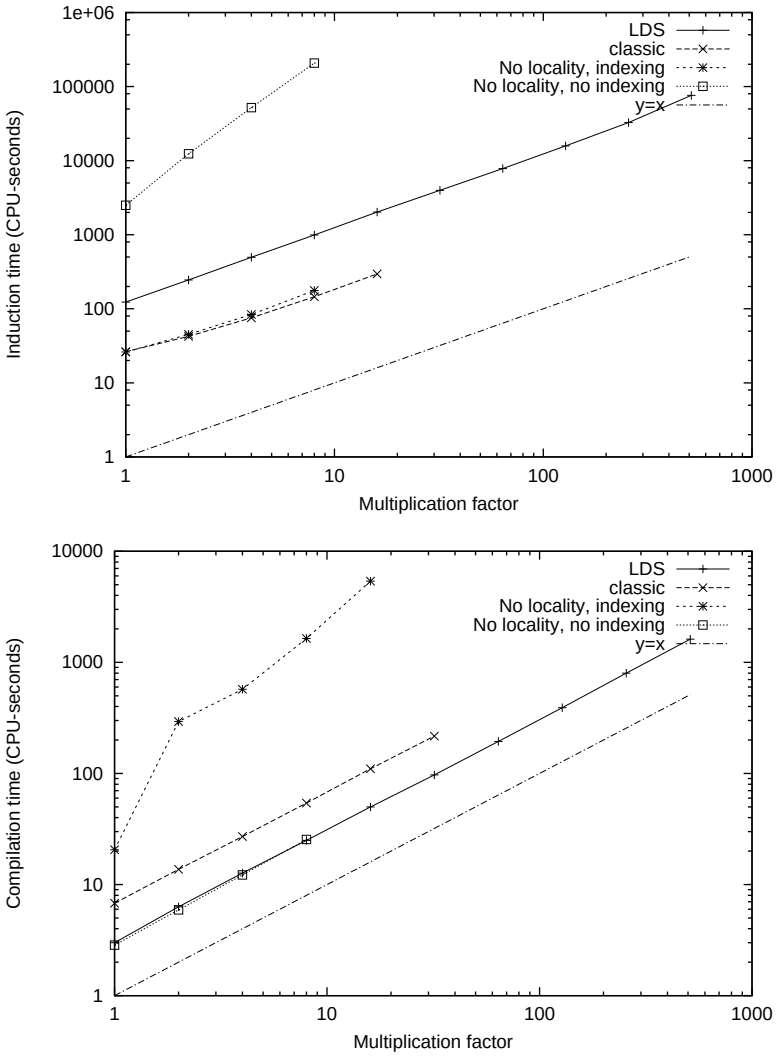


Figure 7.5: Scaling properties of TILDE*LDS* in terms of number of examples.

Properties of the example sets										
mult.factor	1	2	4	8	16	32	64	128	256	512
#examples	188	376	752	1504	3k	6k	12k	24k	48k	96k
#facts	10512	21k	42k	84k	168k	336k	673k	1.3M	2.7M	5.4M
size (MB)	0.25	0.5	1	2	4	8	16	32	65	130
Compilation time (CPU-seconds)										
LDS	3	6.3	12.7	25	50	97	194	391	799	1619
classic	6.8	13.7	27	54	110	217	-	-	-	-
–loc, +ind	20.6	293	572	1640	5381	18k	-	-	-	-
–loc, –ind	2.8	5.9	12.2	25.5	52.3	-	-	-	-	-
Induction time (CPU-seconds)										
LDS	123	245	496	992	2026	3980	7816	16k	33k	76k
classic	26.3	42.5	75.4	149	296	-	-	-	-	-
–loc, +ind	26.1	45.2	83.9	177	-	-	-	-	-	-
–loc, –ind	2501	12k	52k	208k	-	-	-	-	-	-

Table 7.2: Comparison of different TILDE versions on Mutagenesis: TILDE*LDS*, TILDE*classic*, TILDE*classic* without localization but with indexing (–loc, +ind) and TILDE*classic* without localization and without indexing (–loc, –ind).

This is not unexpected, as in this setting the time needed to run a test on one single example increases with the size of the data set.

With respect to compilation times, we note that all are linear in the size of the data set, except TILDE*classic* without locality and with indexing. This is in correspondence with the fact that building an index for the predicates in a deductive database is an expensive operation, super-linear in the size of the database.[3]

Furthermore, the experiments confirm that TILDE*classic* with locality scales as well as TILDE*LDS* with respect to time complexity, but for large data sets runs into problems because it cannot load all the data.

Conclusions

Observing that without indexing induction time increases quadratically, and with indexing compilation time increases quadratically, we conclude that the locality assumption is indeed crucial to our linearity results, and that loading only a few examples at a time in main memory makes it possible to handle much larger data sets.

7.4.4 Experiment 2: The Effect of Localization

Aim of the experiment

In the previous experiment we studied the effect of the number of examples on time complexity, and observed that this effect is different according to whether

[3]It is not clear to what extent the expensiveness of the operation is typical for the MasterProLog compiler, and to what extent it is inherent to the indexing task itself.

the locality assumption is made. In this experiment we do not just distinguish between localized and not localized, but consider gradual changes in localization, and thus try to quantify the effect of localization on the induction time.

Methodology

We can test the influence of localization on the efficiency of TILDE*LDS* by varying the granularity parameter G in TILDE*LDS*. G is the number of examples that are loaded into main memory at the same time. Localization of information is stronger when G is smaller.

The effect of G was tested by running TILDE*LDS* successively on the same data set, under the same circumstances, but with different values for G. In these experiments G ranged from 1 to 200. For each value of G both compilation and induction were performed ten times; the reported times are the means of these ten runs.

Materials

We have used three data sets: a RoboCup data set with 10000 examples, a Poker data set containing 3000 examples, and the Mutagenesis data set with a multiplication factor of 8 (i.e. 1504 examples). The data sets were chosen to contain a sufficient number of examples to make it possible to let G vary over a relatively broad range, but not more (to limit the experimentation time).

Other materials are as described in Section 7.4.2.

Discussion of the Results

Induction times and compilation times are plotted versus granularity in Figure 7.6. It can be seen from these plots that induction time increases approximately linearly with granularity. For very small granularities, too, the induction time can increase. We suspect that this effect can be attributed to an overhead of disk access (loading many small files, instead of fewer larger files). A similar effect is seen when we look at the compilation times: these decrease when the granularity increases, but asymptotically approach a constant. This again suggests an overhead caused by compiling many small files instead of one large file. The fact that the observed effect is smallest for Mutagenesis, where individual examples are larger, increases the plausibility of this explanation.

Conclusions

This experiment clearly shows that the performance of TILDE*LDS* strongly depends on G, and that a reasonably small value for G is preferable. It thus confirms the hypothesis that localization of information is advantageous with respect to time complexity.

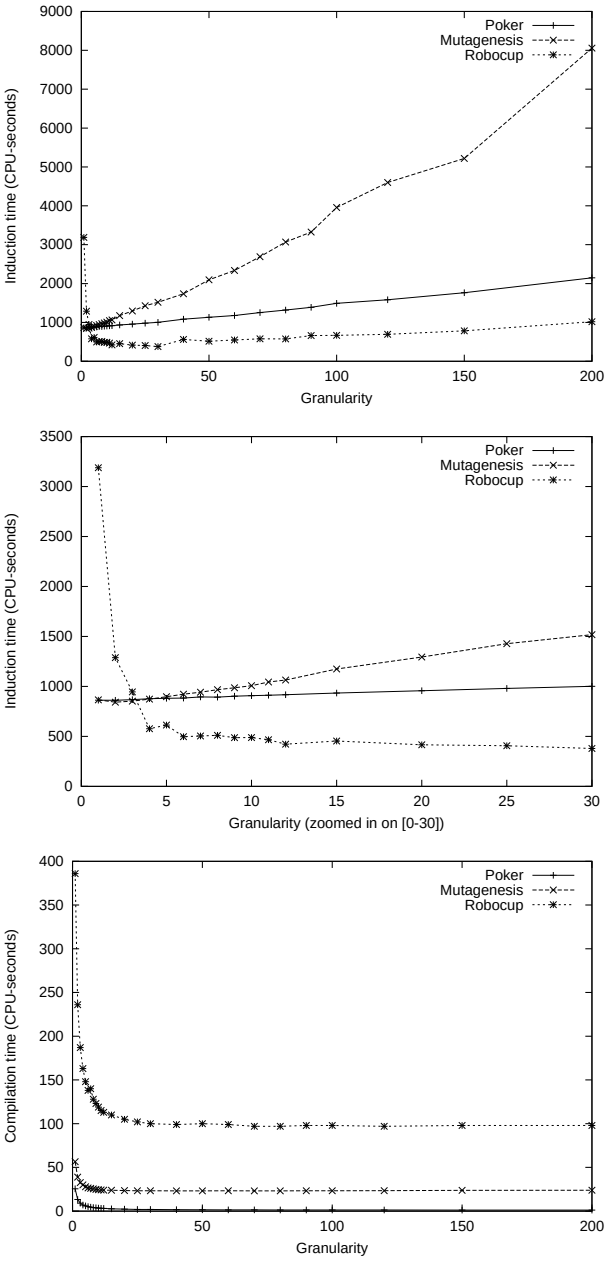


Figure 7.6: The effect of granularity on induction and compilation time.

7.4.5 Experiment 3: Practical Scaling Properties

Aim of the experiment

With this experiment we want to measure how well TILDE*LDS* scales up in practice, without controlling any influences. This means that the tree that is induced is not guaranteed to be the same one or have the same size, and that a natural variation is allowed with respect to the complexity of the examples as well as the complexity of the queries. This experiment is thus meant to mimic the situations that arise in practice.

Since different trees may be grown on different data sets, the quality of these trees may differ. We investigate this as well.

Methodology

The methodology we follow is to choose some domain and then create data sets with different sizes for this domain. TILDE*LDS* is then run on each data set, and for each run the induction time is recorded, as well as the quality of the tree (according to different criteria, see below).

Materials

Data sets from two domains were used: RoboCup and Poker. These domains were chosen because large data sets were available for them. For each domain several data sets of increasing size were created.

Whereas induction times have been measured on both data sets, predictive accuracy has been measured only for the Poker data set. This was done using a separate test set of 100,000 examples, which was the same for all the hypotheses. For the RoboCup data set no test set was constructed, because interpretability of the hypotheses by domain experts is the main evaluation criterion for this application (these theories are used for verification of the behavior of agents, see (Jacobs *et al.*, 1998)).

The RoboCup experiments have been run on a SUN SPARCstation-20 at 100 MHz; for the Poker experiments a SUN Ultra-2 at 168 MHz was used.

Discussion of the Results

Table 7.3 shows the consumed CPU-times in function of the number of examples, as well as the predictive accuracy. These figures are plotted in Figure 7.7. Note that the CPU-time graph is again plotted on a double logarithmic scale.

With respect to accuracy, the Poker hypotheses show the expected behavior: when more data are available, the hypotheses can predict very rare classes (for which no examples occur in smaller data sets), which results in higher accuracy.

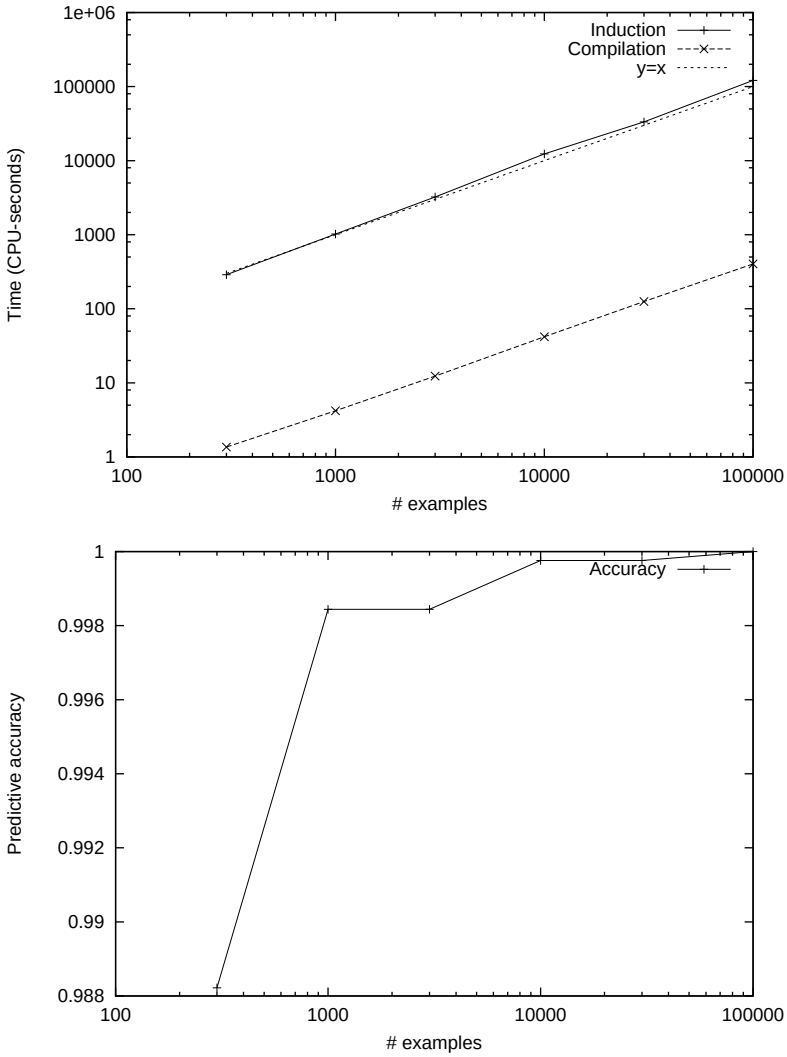


Figure 7.7: Consumed CPU-time and accuracy of hypotheses produced by TILDE*LDS* in the Poker domain, plotted against the number of examples.

#examples	300	1000	3000	10000	30000	100000
compilation (CPU-s)	1.36	4.20	12.36	41.94	125.47	402.63
induction (CPU-s)	288	1021	3231	12325	33394	121266
accuracy	0.98822	0.99844	0.99844	0.99976	0.99976	1.0

Table 7.3: Consumed CPU-time and accuracy of hypotheses produced by TILDE*LDS* in the Poker domain.

#examples	10k	20k	30k	40k	50k	60k	70k	80k	89k
compilation	274	522	862	1120	1302	1793	1964	2373	2615
induction	1448	4429	7678	9285	6607	13665	29113	28504	50353
	±44	±83	±154	±552	±704	±441	±304	±657	±3063

Table 7.4: Consumed CPU-time of hypotheses produced by TILDE*LDS* in the RoboCup domain; for induction times standard errors are added.

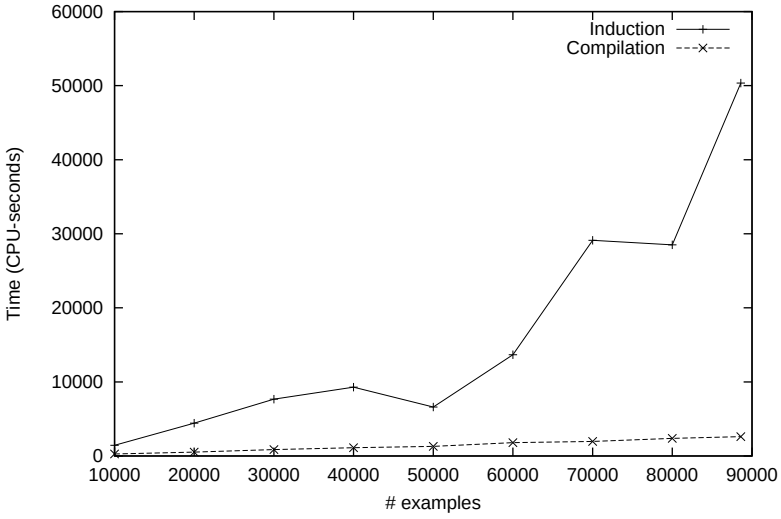


Figure 7.8: Consumed CPU-time for TILDE*LDS* in the RoboCup domain, plotted against the number of examples.

The graphs further show that in the Poker domain, TILDE*LDS* scales up linearly, even though more accurate (and slightly more complex) theories are found for larger data sets.

In the RoboCup domain, the induced hypotheses were the same for all runs except the 10000 examples run. In this single case the hypothesis was more simple and, according to the domain expert, less informative than for the other runs. This suggests that in this domain a relatively small set of examples (20000) suffices to learn from.

It is harder to see how TILDE*LDS* scales up for the RoboCup data. Since the same tree is returned in all runs except the 10000 examples run, one would expect the induction times to grow linearly. However, the observed curve does not seem linear, although it does not show a clear tendency to be super-linear either. Because large variations in induction time were observed, we performed these runs 10 times; the estimated mean induction times are reported together with their standard errors. The standard errors alone cannot explain the observed deviations, nor can variations in example complexity (all examples are of equal complexity in this domain).

A possible explanation is the fact that the Prolog engine performs a number of tasks that are not controlled by TILDE, such as garbage collection. In specific cases, the Prolog engine may perform many garbage collections before expanding its memory space (this happens when the amount of free memory after garbage collection is always just above some threshold), and the time needed for these garbage collections is included in the measured CPU-times. The MasterProLog engine is known to sometimes exhibit such behavior (cf. Bart Demoen, personal communication).

In order to sort this out, TILDE*LDS* would have to be reimplemented in a language of lower level than Prolog in order to have full control over all computations that occur. Such a reimplementation is planned (but not within this thesis work).

Conclusions

We stress that any results concerning the evolution of tree complexity and accuracy when more data become available are necessarily domain-dependent, and one should be cautious when generalizing them. It seems safe, though, to conclude from our experiments that the linear scaling property has at least a reasonable chance of occurring in practice.

7.5 Related Work

The text of this chapter is based on (Blockeel *et al.*, 1998a).

This work is closely related to efforts in the propositional learning field to increase the capability of machine learning systems to handle large databases. It has been influenced more specifically by a tutorial on data mining by Usama Fayyad, in which the work of Mehta and others was mentioned (Mehta *et al.*, 1996; Shafer *et al.*, 1996). They were the first to propose the level-wise tree building algorithm we adopted, and to implement it in the SLIQ (Mehta *et al.*, 1996) and SPRINT (Shafer *et al.*, 1996) systems. The main difference with our approach is that SLIQ and SPRINT learn from one single relation, while TILDE*LDS* can learn from multiple relations.

Related work inside ILP includes the RDT/DB system (Morik and Brockhausen, 1996), which presents the first approach to coupling an ILP system with a relational database management system (RDBMS). Being an ILP system, RDT/DB also learns from multiple relations. The approach followed is that a logical test that is to be performed is converted into an SQL query and sent to an external relational database management system. This approach is essentially different from ours, in that it exploits as much as possible the power of the RDBMS to efficiently evaluate queries. Also, there is no need for preprocessing the data. Disadvantages are that for each query an external database is accessed, which is slow, and that it is less flexible with respect to background knowledge. Some of the issues involved in coupling ILP systems with relational databases are discussed in (Blockeel and De Raedt, 1996).

We also mention the KEPLER system (Wrobel *et al.*, 1996), a data mining tool that provides a framework for applying a broad range of data mining systems to data sets; this includes ILP systems. KEPLER was deliberately designed to be very open, and systems using the learning from interpretations setting can be plugged into it as easily as other systems.

Of the systems using the learning from interpretations setting (De Raedt and Van Laer, 1995; De Raedt and Dehaspe, 1997; Dehaspe and De Raedt, 1997), the WARMR system (finding association rules over multiple relations; see also (Dehaspe and Toivonen, 1998)) is most closely related to the work described in this chapter, in the sense that there, too, an effort was made to adapt the system for large databases.

More loosely related work inside ILP would include all efforts to make ILP systems more efficient. Since most of this work concerns ILP systems that learn from entailment, the way in which this is done usually differs substantially from what we describe in this paper. For instance, the PROGOL system (Muggleton, 1995) has recently been extended with caching and other efficiency improvements (Cussens, 1997). Another direction of work is the use of sampling techniques, see e.g. (Srinivasan, 1998; Sebag, 1998).

7.6 Conclusions

We have argued and demonstrated empirically that the use of ILP is not limited to small databases, as is often assumed. Mining databases of over a hundred megabytes was shown to be feasible, and this does not seem to be a limit.

The positive results that have been obtained are due mainly to the use of the learning from interpretations setting, which makes the link with propositional learning more clear. This made it easier to upgrade the work by Mehta *et al.* (1996), which turned out to be crucial for handling large data sets. Incorporating the same technique in a system that uses the classical ILP setting seems much more difficult.

The currently available results suggest that the alternative implementation may be preferable to the original one, even for data sets that can be handled by the latter. First of all, it uses less memory and second, it offers interesting optimization opportunities. Such opportunities would also become available for other ILP systems if they were re-implemented in a similar fashion as TILDE.

Although we obtained our results only for a specific kind of data mining (induction of decision trees), the results are generalizable not only to other approaches within the classification context (e.g. rule based approaches) but also to other inductive tasks within the learning from interpretations setting, such as clustering, regression and induction of association rules.

Chapter 8

Conclusions

This work started out from the observation that many sophisticated techniques for machine learning and data mining exist, but most of them are set in the attribute value learning framework. This limits the application of these techniques to those domains where the data can be represented as vectors of fixed length, and hypotheses are essentially propositional.

Inductive logic programming, on the other hand, provides a richer description language for both data and hypotheses, namely first order predicate logic. However, the field is younger and many of the techniques existing in attribute value learning do not have a counterpart in inductive logic programming.

The aim of this work was to port some of the techniques in attribute value learning to inductive logic programming. The techniques we focused on are all decision tree based techniques. To achieve our goal, we have worked in several steps:

- We have formulated a general framework for predictive induction that we call predictive clustering, and that encompasses more specific techniques such as classification, regression, unsupervised learning and flexible prediction. We have furthermore shown that induction of decision trees can be defined at this general level, and that this technique reduces to the classical approaches for building classification or regression trees by instantiating a few parameters.

- While propositional rule sets have a first order counterpart in the form of Prolog programs, propositional decision trees do not have such a counterpart. For that reason, we have defined and studied first order logical decision trees.

- Finally, we have used these first order logical decision trees to upgrade the predictive clustering framework to inductive logic programming. This

has resulted in the implementation of the multi-purpose inductive logic programming system TILDE, which can perform classification, regression, and several kinds of clustering.

We have evaluated our approach by running TILDE on a broad range of applications, empirically studying its performance with respect to the predictive accuracy of the hypotheses it finds, the simplicity and understandability of these hypotheses, and the efficiency with which they can be induced.

Our main conclusions are that the sophisticated attribute learning techniques that we have focused on can indeed be upgraded to the inductive logic programming framework. The upgraded versions inherit many desirable properties of their propositional counterparts, such as the ability to induce hypotheses of high quality (with respect to both predictive accuracy and simplicity), high efficiency and good scalability properties. This is reflected by the fact that TILDE can compete with many current state-of-the-art systems for inductive logic programming, even though most of these are more oriented towards specific tasks and can handle only a subset of the tasks TILDE can handle.

An interesting side-result of this research is that several first order representation formalisms have been related to one another with respect to their expressiveness. It turns out that in the first order context decision lists and decision trees have an advantage over flat logic programs in this respect.

Further work can be identified in several directions. First of all, the clustering and regression subsystems can be improved. They are trivial instantiations of our general predictive clustering approach. While this general approach provides a sound basis, and indeed its trivial instantiations work quite well already, in order to achieve the same performance as highly specialized systems more specialized techniques should be incorporated.

A second direction for future work is a further study of the predictive clustering framework. We have argued that this framework suggests several interesting opportunities for improving the behavior of inductive learners, but these have not been studied in detail yet. A lot of work can be done in this area.

Finally, we plan to implement a version of the TILDE system in C or some other low-level language. The current system is written in Prolog. This has enabled a fast development of the system and created an ideal environment for testing different versions and making it highly parametrized. However, an implementation in C would enable a better comparison with other inductive learners with respect to efficiency, and may also yield more precise experimental results.

Bibliography

[Agrawal *et al.*, 1996] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. The MIT Press, 1996.

[Aha *et al.*, 1991] D. Aha, D. Kibler, and M. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[Bain and Muggleton, 1992] M. Bain and S. Muggleton. Non-monotonic learning. In S. Muggleton, editor, *Inductive logic programming*, pages 145–161. Academic Press, London, 1992.

[Bergadano and Giordana, 1988] F. Bergadano and A. Giordana. A knowledge intensive approach to concept induction. In *Proceedings of the 5th International Workshop on Machine Learning*. Morgan Kaufmann, 1988.

[Bergadano *et al.*, 1997] F. Bergadano, D. Gunetti, F. Neri, and G. Ruffo. ILP data analysis in adaptive system and network management, December 1997. In Periodic Progress Report of ESPRIT LTR Project 20237 (ILP2).

[Bisson, 1992a] G. Bisson. Conceptual clustering in a first order logic representation. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 458–462. John Wiley & Sons, 1992.

[Bisson, 1992b] G. Bisson. Learning in FOL with a similarity measure. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-92)*. AAAI Press, 1992.

[Blockeel and De Raedt, 1996] H. Blockeel and L. De Raedt. Relational knowledge discovery in databases. In *Proceedings of the 6th International Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 199–212. Springer-Verlag, 1996.

[Blockeel and De Raedt, 1997] H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–85. Springer-Verlag, 1997.

[Blockeel and De Raedt, 1998a] H. Blockeel and L. De Raedt. Isidd: An interactive system for inductive database design. *Applied Artificial Intelligence*, 12(5):385–421, July-August 1998.

[Blockeel and De Raedt, 1998b] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.

[Blockeel *et al.*, 1998a] H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 1998. To appear.

[Blockeel *et al.*, 1998b] H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning*, pages 55–63, 1998. `http://www.cs.kuleuven.ac.be/~ml/PS/ML98-56.ps`.

[Bloedorn and Michalski, 1996] E. Bloedorn and R. Michalski. The AQ17-DCI system for data-driven constructive induction and its application to the analysis of world economics. In Z. Raś and M. Michalewicz, editors, *Foundations of Intelligent Systems*, volume 1079 of *Lecture Notes in Artificial Intelligence*, pages 108–117. Springer-Verlag, 1996.

[Boström, 1995] H. Boström. Covering vs. divide-and-conquer for top-down induction of logic programs. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995.

[Bowers, 1998] A.F. Bowers. Early experiments with a higher-order decision-tree learner. In J. Lloyd, editor, *Proceedings of the JICSLP '98 post-conference workshop and CompulogNet Area Meeting on Computational Logic and Machine Learning*, pages 42–49, 1998.

[Bratko, 1990] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1990. 2nd Edition.

[Breiman *et al.*, 1984] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

[Breiman, 1996] L. Breiman. Some properties of splitting criteria. *Machine Learning*, 24:41–47, 1996.

[Buntine and Niblett, 1992] W. Buntine and T. Niblett. A further comparison of splitting rules for decision tree induction. *Machine Learning*, 8:75–85, 1992.

[Catlett, 1991] J. Catlett. On changing continuous attributes into ordered discrete attributes. In Yves Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 164–178. Springer-Verlag, 1991.

[Chapman and Kaelbling, 1991] D. Chapman and L.P. Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1991.

[Clark and Niblett, 1989] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.

[Clocksin and Mellish, 1981] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.

[Cohen and Page, 1995] W.W. Cohen and D. Page. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Computing*, 13, 1995.

[Cohen, 1995] W.W. Cohen. Pac-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, 2:541–573, 1995.

[Cussens, 1997] J. Cussens. Part-of-speech tagging using progol. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 93–108. Springer-Verlag, 1997.

[De Raedt and Blockeel, 1997] L. De Raedt and H. Blockeel. Using logical decision trees for clustering. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 133–141. Springer-Verlag, 1997.

[De Raedt and Bruynooghe, 1993] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.

[De Raedt and Dehaspe, 1997] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

[De Raedt and Džeroski, 1994] L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[De Raedt and Van Laer, 1995] L. De Raedt and W. Van Laer. Inductive constraint logic. In Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann, editors, *Proceedings of the 6th International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.

[De Raedt *et al.*, 1995] L. De Raedt, L. Dehaspe, W Van Laer, H. Blockeel, and M. Bruynooghe. On the duality of CNF and DNF, or how to learn CNF using a DNF learner. Unpublished, 1995.

[De Raedt *et al.*, 1997] L. De Raedt, P. Idestam-Almquist, and G. Sablon. θ-subsumption for structural matching. In *Proceedings of the 9th European Conference on Machine Learning*, pages 73–84. Springer-Verlag, 1997.

[De Raedt *et al.*, 1998] L. De Raedt, H. Blockeel, L. Dehaspe, and W. Van Laer. Three companions for first order data mining. In S. Džeroski and N. Lavrač, editors, *Inductive Logic Programming for Knowledge Discovery in Databases*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1998. To appear.

[De Raedt, 1996] L. De Raedt, editor. *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 1996.

[De Raedt, 1997] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.

[De Raedt, 1998] L. De Raedt. Attribute-value learning versus inductive logic programming: the missing links (extended abstract). In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 1–8. Springer-Verlag, 1998.

[Dehaspe and De Raedt, 1997] L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 125–132. Springer-Verlag, 1997.

[Dehaspe and Toivonen, 1998] L. Dehaspe and H. Toivonen. Frequent query discovery: a unifying ILP approach to association rule mining. *Data Mining and Knowledge Discovery*, 1998. To appear.

[Dietterich *et al.*, 1997] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2):31–71, 1997.

[Dolšak and Muggleton, 1992] B. Dolšak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive logic programming*, pages 453–472. Academic Press, 1992.

[Domingos, 1996] P. Domingos. Unifying instance based and rule based induction. *Machine Learning*, 24(2):141–168, 1996.

[Domingos, 1998] P. Domingos. Data mining with RISE and CWS. In F. Esposito, R.S. Michalski, and L. Saitta, editors, *Proceedings of the 4th International Workshop on Multistrategy Learning*, pages 1–12, 1998.

[Dougherty *et al.*, 1995] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In A. Prieditis and S. Russell, editors, *Proc. Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 1995.

[Džeroski *et al.*, 1998] S. Džeroski, S. Schulze-Kremer, K. R. Heidtke, K. Siems, D. Wettschereck, and H. Blockeel. Diterpene structure elucidation from 13C NMR spectra with inductive logic programming. *Applied Artificial Intelligence*, 12(5):363–384, July-August 1998.

[Džeroski *et al.*, 1992] S. Džeroski, S. Muggleton, and S. Russell. PAC-learnability of determinate logic programs. In *Proceedings of the 5th ACM workshop on Computational Learning Theory*, pages 128–135, 1992.

[Džeroski *et al.*, 1998] S. Džeroski, L. De Raedt, and H. Blockeel. Relational reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann, 1998.

[Elmasri and Navathe, 1989] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, 2nd edition, 1989.

[Emde and Wettschereck, 1996] W. Emde and D. Wettschereck. Relational instance-based learning. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning*, pages 122–130. Morgan Kaufmann, 1996.

[Emde, 1994] W. Emde. Inductive learning of characteristic concept descriptions. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 51–70, Sankt Augustin, Germany, 1994. Gesellschaft für Mathematik und Datenverarbeitung MBH.

[Fayyad and Irani, 1993] U.M. Fayyad and K.B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1022–1027, San Mateo, CA, 1993. Morgan Kaufmann.

[Fayyad *et al.*, 1996] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. The MIT Press, 1996.

[Fisher and Langley, 1985] D. Fisher and P. Langley. Approaches to conceptual clustering. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 691–697, Los Altos, CA, 1985. Morgan Kaufmann.

[Fisher, 1987] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.

[Fisher, 1996] D. H. Fisher. Iterative optimization and simplification of hierarchical clusterings. *Journal of Artificial Intelligence Research*, 4:147–179, 1996.

[Frawley *et al.*, 1991] W. Frawley, G. Piatetsky-Shapiro, and C. Matheus. Knowledge discovery in databases: an overview. In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 1–27. Cambridge, MA: MIT Press, 1991.

[Geibel and Wysotzki, 1997] P. Geibel and F. Wysotzki. A logical framework for graph theoretical decision tree learning. In N. Lavrač and S. Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 173–180, 1997.

[Gluck and Corter, 1985] M.A. Gluck and J.E. Corter. Information, uncertainty, and the utility of categories. In *Proceedings of the Seventh Annaul Conference of the Cognitive Science Society*, pages 283–287, Hillsdale, NJ, 1985. Lawrence Erlbaum.

[Hartigan, 1975] J.A. Hartigan. *Clustering Algorithms*. Wiley New York, 1975.

[Helft, 1989] N. Helft. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–156. Morgan Kaufmann, 1989.

[Hutchinson, 1997] A. Hutchinson. Metrics on terms and clauses. In *Proceedings of the 9th European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence, pages 138–145. Springer-Verlag, 1997.

[Jacobs *et al.*, 1998] N. Jacobs, K. Driessens, and L. De Raedt. Using ILP systems for verification and validation of multi agent systems. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446, pages 145–154. Springer-Verlag, 1998.

[Karalič and Bratko, 1997] A. Karalič and I. Bratko. First order regression. *Machine Learning*, 26:147–176, 1997.

[Kazakov *et al.*, 1996] D. Kazakov, L. Popelinsky, and O. Stepankova. ILP datasets page [`http://www.gmd.de/ml-archive/datasets/ilp-res.html`], 1996.

[Ketterlin *et al.*, 1995] A. Ketterlin, P. Gancarski, and J.J. Korczak. Conceptual clustering in structured databases : a practical approach. In *Proceedings of KDD-95*, 1995.

[Khan *et al.*, 1998] K. Khan, S. Muggleton, and R. Parson. Repeat learning using predicate invention. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, pages 165–174, 1998.

[Kietz and Morik, 1994] J.U. Kietz and K.. Morik. A polynomial approach to the constructive induction of structural knowledge. *Machine Learning*, 14:193–217, 1994.

[Kirsten and Wrobel, 1998] M. Kirsten and S. Wrobel. Relational distance-based clustering. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 261–270. Springer-Verlag, 1998.

[Kitano *et al.*, 1997] H. Kitano, M. Veloso, H. Matsubara, M. Tambe, S. Coradeschi, I. Noda, P. Stone, E. Osawa, and M. Asada. The robocup synthetic agent challenge 97. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 24–29. Morgan Kaufmann, 1997.

[Kowalski, 1979] R. Kowalski. *Logic for problem solving*. North-Holland, 1979.

[Kramer *et al.*, 1998] S. Kramer, B. Pfahringer, and C. Helma. Stochastic propositionalization of non-determinate background knowledge. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, pages 80–94, 1998.

[Kramer, 1996] S. Kramer. Structural regression trees. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[Langley and Sage, 1984] P. Langley and S. Sage. Conceptual clustering as discrimination learning. In *Proceedings of the Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 95–98, London, Ontario, Canada, 1984.

[Langley, 1996] P. Langley. *Elements of Machine Learning.* Morgan Kaufmann, 1996.

[Lavrač and Džeroski, 1997] N. Lavrač and S. Džeroski, editors. *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1997.

[Lloyd, 1987] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 2nd edition, 1987.

[Manago, 1989] M. Manago. Knowledge intensive induction. In A. M. Segre, editor, *Proceedings of the 6th International Workshop on Machine Learning*, pages 151–155. Morgan Kaufmann, 1989.

[Mehta *et al.*, 1996] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proceedings of the Fifth International Conference on Extending Database Technology*, 1996.

[Merz and Murphy, 1996]
C.J. Merz and P.M. Murphy. UCI repository of machine learning databases [`http://www.ics.uci.edu/~mlearn/mlrepository.html`] , 1996. Irvine, CA: University of California, Department of Information and Computer Science.

[Michalski and Chilausky, 1980] R.S. Michalski and R.L. Chilausky. Learning by being told and learning from examples: an experimental comparaison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy analysis and information systems*, 4, 1980.

[Michalski and Stepp, 1983] R.S. Michalski and R.E. Stepp. Learning from observation: conceptual clustering. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 1. Tioga Publishing Company, 1983.

[Michalski *et al.*, 1986] R. Michalski, I. Mozetič, J. Hong, and N. Lavrač. The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86)*, pages 1041–1045, Philadelphia, PA, 1986.

[Mingers, 1989] J. Mingers. An empirical comparison of selection measures for decision tree induction. *Machine Learning*, 3:319–342, 1989.

[Mitchell, 1997] T. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[Mooney and Califf, 1995] R.J. Mooney and M.E. Califf. Induction of first-order decision lists: Results on learning the past tense of english verbs. *Journal of Artificial Intelligence Research*, pages 1–23, 1995.

[Mooney, 1995] R.J. Mooney. Encouraging experimental results on learning cnf. *Machine Learning*, 19:79–92, 1995.

[Morik and Brockhausen, 1996] K. Morik and P. Brockhausen. A multistrategy approach to relational discovery in databases. In R.S. Michalski and Wnek J., editors, *Proceedings of the 3rd International Workshop on Multistrategy Learning*, pages 17–28, 1996.

[Muggleton and De Raedt, 1994] S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[Muggleton *et al.*, 1996] S. Muggleton, D. Page, and A. Srinivasan. An initial experiment into stereochemistry-based drug design using inductive logic programming. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 25–40. Springer-Verlag, 1996.

[Muggleton, 1995] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13, 1995.

[Muggleton, 1996] S. Muggleton. Learning from positive data. In S. Muggleton, editor, *Proceedings of the 6th Inductive Logic Programming Workshop*, 1996.

[Muggleton, 1997] S. Muggleton, editor. *Proceedings of the 6th International Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1997.

[Nienhuys-Cheng, 1997] Shan-Hwei Nienhuys-Cheng. Distance between herbrand interpretations: A measure for approximations to a target concept. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.

[Page, 1998] D. Page, editor. *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1998.

[Piatetsky-Shapiro and Frawley, 1991] G. Piatetsky-Shapiro and W. Frawley, editors. *Knowledge discovery in databases*. The MIT Press, 1991.

[Plotkin, 1970] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[Quinlan, 1986] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Quinlan, 1993a] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann, 1993.

[Quinlan, 1993b] J.R. Quinlan. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1993.

[Quinlan, 1996] J. R. Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, October 1996.

[Ramon and Bruynooghe, 1998] J. Ramon and M. Bruynooghe. A framework for defining distances between first-order logic objects. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 271–280. Springer-Verlag, 1998.

[Ramon *et al.*, 1998] J. Ramon, M. Bruynooghe, and W. Van Laer. Distance measures between atoms. In *Proceedings of the CompulogNet Area Meeting on 'Computational Logic and Machine Learning'*, pages 35–41, 1998.

[Rissanen, 1978] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465–471, 1978.

[Rivest, 1987] R.L. Rivest. Learning decision lists. *Machine Learning*, 2:229–246, 1987.

[Russell and Norvig, 1995] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[Sebag, 1998] M. Sebag. A stochastic simple similarity. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 95–105. Springer-Verlag, 1998.

[Shafer *et al.*, 1996] J.C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 22th International Conference on Very Large Databases*, 1996.

[Sneath and Sokal, 1973] P. Sneath and R. Sokal. *Numerical Taxonomy: the Principles and Practice of Numerical Classification*. Freeman San Francisco, 1973.

[Srinivasan and Camacho, 1996] A. Srinivasan and R.C. Camacho. Experiments in numerical reasoning with ILP. Technical Report PRG-TR-22-96, Oxford University Computing Laboratory, Oxford, 1996. Accepted to appear in the Journal of Logic Programming, Special Issue on ILP.

[Srinivasan and King, 1996] A. Srinivasan and R.D. King. Feature construction with inductive logic programming: A study of quantitative predictions of biological activity aided by structural attributes. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 89–104. Springer-Verlag, 1996.

[Srinivasan *et al.*, 1995] A. Srinivasan, S.H. Muggleton, and R.D. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 1995.

[Srinivasan *et al.*, 1996] A. Srinivasan, S.H. Muggleton, M.J.E. Sternberg, and R.D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85, 1996.

[Srinivasan, 1998] A. Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 1998.

[Stahl, 1996] I. Stahl. Predicate invention in inductive logic programming. In L. De Raedt, editor, *Advances in inductive logic programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*, pages 34–47. IOS Press, 1996.

[Sterling and Shapiro, 1986] Leon Sterling and Ehud Shapiro. *The art of Prolog.* The MIT Press, 1986.

[Thompson and Langley, 1991] K. Thompson and P. Langley. Concept formation in structured domains. In D. Fisher, M. Pazzani, and P. Langley, editors, *Concept formation: knowledge and experience in unsupervised learning.* Morgan Kaufmann, 1991.

[Utgoff, 1989] P.E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4(2):161–186, 1989.

[Valiant, 1984] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[van der Laag and Nienhuys-Cheng, 1998] Patrick R. J. van der Laag and Shan-Hwei Nienhuys-Cheng. Completeness and properness of refinement operators in inductive logic programming. *Journal of Logic Programming*, 34(3):201–225, 1998.

[Van Laer *et al.*, 1997] W. Van Laer, L. De Raedt, and S. Džeroski. On multi-class problems and discretization in inductive logic programming. In Zbigniew W. Ras and Andrzej Skowron, editors, *Proceedings of the 10th International Symposium on Methodologies for Intelligent Systems (ISMIS97)*, volume 1325 of *Lecture Notes in Artificial Intelligence*, pages 277–286. Springer-Verlag, 1997.

[Watanabe and Rendell, 1991] L. Watanabe and L. Rendell. Learning structural decision trees from examples. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 770–776, 1991.

[Webb, 1996] G.I. Webb. Further experimental evidence against the utility of occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417, 1996.

[Wilson and Martinez, 1997] D.R. Wilson and T.R. Martinez. Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, 6:1–34, 1997.

[Wnek and Michalski, 1993] J. Wnek and R. Michalski. Hypothesis-driven constructive induction in AQ17-HCI: A method and experiments. *Machine Learning*, 14(2):139–168, 1993.

[Wrobel *et al.*, 1996] S. Wrobel, D. Wettschereck, E. Sommer, and W. Emde. Extensibility in data mining systems. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*. AAAI Press, 1996.

[Yoo and Fisher, 1991] J. Yoo and D. Fisher. Concept formation over explanations and problem-solving experience. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 630 – 636. Morgan Kaufmann, 1991.

Appendix A

Data Sets

In this appendix we describe the data sets that we have used.

A.1 Soybeans

This database (Michalski and Chilausky, 1980) contains descriptions of diseased soybean plants. Every plant is described by 35 attributes. The plants have to be classified into one of several classes, each class being a specific disease. A small data set (46 examples, 4 classes) and a large one (307 examples, 19 classes) are available at the UCI data repository (Merz and Murphy, 1996). A single example from this data set, represented in TILDE's input format, is given in Figure A.1.

A.2 Iris

The Iris database, available at the UCI repository, contains descriptions of iris plants that are to be classified into one of three classes. For each class there are 50 examples. There are 4 numerical attributes. The problem is very easy to solve for supervised learners, hence the database is used mostly for unsupervised learners. The evaluation criterion is then, whether the learner finds three clusters that correspond to the classes.

Some examples of the Iris data set are shown in Figure A.2.

A.3 Mutagenesis

This database (Srinivasan *et al.*, 1996), available at the ILP repository (Kazakov *et al.*, 1996), is probably the most popular database that has ever been used

```
begin(model(16)).
  ziekte(2).
  date(5).
  plant_stand(0).
  precip(0).
  temp(2).
  hail(1).
  crop_hist(3).
  area_damaged(3).
  severity(1).
  seed_tmt(1).
  germination(2).
  plant_growth(1).
  leaves(1).
  leafspots_halo(0).
  leafspots_marg(2).
  leafspot_size(2).
  leaf_shread(0).
  leaf_malf(0).
  leaf_mild(0).
  stem(1).
  lodging(0).
  stem_cankers(0).
  canker_lesion(3).
  fruiting_bodies(0).
  external_decay(0).
  mycelium(0).
  int_discolor(2).
  sclerotia(1).
  fruit_pods(0).
  fruit_spots(4).
  seed(0).
  mold_growth(0).
  seed_discolor(0).
  seed_size(0).
  shriveling(0).
end(model(16)).
```

Figure A.1: An example from the Soybeans data set.

```
begin(model(1)).
  sl(5.1).
  sw(3.5).
  pl(1.4).
  pw(0.2).
  irissetosa.
end(model(1)).

begin(model(2)).
  sl(4.9).
  sw(3.0).
  pl(1.4).
  pw(0.2).
  irissetosa.
end(model(2)).

begin(model(51)).
  sl(7.0).
  sw(3.2).
  pl(4.7).
  pw(1.4).
  irisversicolor.
end(model(51)).
```

Figure A.2: Examples from the Iris data set.

in ILP. It contains descriptions of molecules for which the mutagenic activity has to be predicted (mutagenicity is the ability to cause DNA to mutate, which is a possible cause of cancer). Originally mutagenicity was measured by a real number, but in most experiments with ILP systems this has been discretized into two values (positive and negative, i.e. mutagenic or non-mutagenic), so that the task becomes a classification task.

The data set consists of 230 molecules, which are divided into two subsets: regression-friendly (188 molecules) and regression-unfriendly (42 molecules). The term regression in this respect refers to the use of linear regression, not regression trees. The 42 regression-unfriendly molecules are those where propositional linear regression methods did not yield good results; these are the most interesting ones for ILP (although the set of 188 molecules has been used more often).

Srinivasan (Srinivasan *et al.*, 1995) introduces four levels of "background" knowledge, each of which is a superset of its predecessor:

- Background B_1 contains only non-numerical structural descriptions: the atoms and bonds in the molecules.
- In Background B_2 the partial charges of the atoms in the molecules are also available.
- Background B_3 adds to the structural description of a molecule two numerical attributes called *lumo* and *logp*. These attributes are known to be highly relevant for mutagenicity.
- Background B_4 contains higher level submolecular structures such as benzene rings, phenanthrene structures, ...

Note that we are using the term "background knowledge" somewhat inconsistently here. The different backgrounds actually provide richer descriptions of the examples, they do not only provide general domain knowledge. As mentioned in Chapter 4 the difference between these two is not made in the normal ILP setting: the examples are single facts there, and all the rest is background knowledge. That explains why the different levels of description were called "backgrounds" in (Srinivasan *et al.*, 1995). Due to the frequent occurrence of these terms in the literature we use them as well.

To illustrate what a description of one molecule looks like, Figure A.3 shows a part of such a description. The level of detail is that of B_2. Each `atom` fact states the existence of a single atom, the element it belongs to (e.g. `c` means it is a carbon atom), its type (atoms of the same element are further classified into different types according to their conformation) and its partial charge. A `bond` literal indicates which atoms are bound, and what type of bond it is (single, double, aromatic, ...). The atoms `d1_1` to `d1_6` can be seen to form a benzene ring in this description.

```
begin(model(1)).
   pos.
   atom(d1_1,c,22,-0.117).
   atom(d1_2,c,22,-0.117).
   atom(d1_3,c,22,-0.117).
   atom(d1_4,c,195,-0.087).
   atom(d1_5,c,195,0.013).
   atom(d1_6,c,22,-0.117).
   (...)
   atom(d1_25,o,40,-0.388).
   atom(d1_26,o,40,-0.388).

   bond(d1_1,d1_2,7).
   bond(d1_2,d1_3,7).
   bond(d1_3,d1_4,7).
   bond(d1_4,d1_5,7).
   bond(d1_5,d1_6,7).
   bond(d1_6,d1_1,7).
   bond(d1_1,d1_7,1).
   bond(d1_2,d1_8,1).
   bond(d1_3,d1_9,1).
   (...)
   bond(d1_24,d1_19,1).
   bond(d1_24,d1_25,2).
   bond(d1_24,d1_26,2).
end(model(1)).
```

Figure A.3: The Prolog representation of one example in the Mutagenesis data set. The `atom` facts enumerate the atoms in the molecule. For each atom its element (e.g. carbon), type (e.g. carbon can occur in several configurations; each type corresponds to one specific configuration) and partial charge. The `bond` facts enumerate all the bonds between the atoms (the last argument is the type of the bond: single, double, aromatic, etc.). `pos` denotes that the molecule belongs to the positive class (i.e. is mutagenic).

```
typed_language(yes).
type(atom(id, element, type, charge)).
type(bond(id, id, bondtype)).
type(X =< X) :- X = charge.
type(X=X).
type(member(_, _)).
type(discretized(_,_,_)).

/* discretization */

discretization(bounds(100)).
to_be_discretized(atom(_, _, _, Ch), [Ch]).

/* refinement operator */

rmode(20: #(230*37*T: atom(_, _, T, _), atom(A, E, T, Ch))).
rmode(20: #(230*9*E: atom(_, E, _, _), atom(A, E, T, Ch))).
rmode(20: #(1*100*C:
            (discretized(atom(_, _, _, X), [X], L), member(C, L)),
            (atom(-A, E, T, Ch), Ch =< C))).
rmode(20: bond(+A1, -A2, BT)).

lookahead(bond(A1, A2, BT), #(1*6*C: member(C, [1,2,3,4,5,7]),
                                     BT=C)).
lookahead(bond(A1, A2, BT), #(230*9*E: atom(A2, E, _, _),
                                       atom(A2, E, _, _))).
lookahead(bond(A1, A2, BT), #(230*37*T: atom(A2, _, T, _),
                                        atom(A2, _, T, _))).
```

Figure A.4: A typical settings file for the Mutagenesis data set.

Figure A.4 shows a typical settings file for running TILDE on this data set. The settings file allows the system to make use of B_2: atoms, bonds, and partial charges.

A.4 Biodegradability

This data set consists of 62 molecules of which structural descriptions and molecular weights are given. The representation of the examples is very similar to that used for the Mutagenesis data set.

The aim is to predict the biodegradability of the molecules. This is expressed as (we cite Kramer (1996)) "the halfrate of surface water aerobic

```
begin(model('MUSK-211')).
  testid('MUSK-211').
  musk.
  conformation('211_1+1', 46, -108, -60, -69, ..., -112, 96).
  conformation('211_1+10', 41, -188, -145,  ...   ...   ...).
  (...)
  conformation('211_1+9', ...   ...   ...  ...   , -113, 97).
end(model('MUSK-211')).
```

Figure A.5: A part of one example from the Musk data set: the molecule called MUSK-211. It has several conformations, referred to as `211_1+1` etc.

aqueous biodegradation in hours". This is a real number, but has been discretized into four values (fast, moderate, slow, resistant) in most past experiments.

This data set was kindly provided to us by Sašo Džeroski and is used with permission, but is not yet in the public domain.

A.5 Musk

Two Musk data sets are available at the UCI data repository: a small one (320kB, 476 tuples) and a large one (4.5MB, 6600 tuples). Each tuple consists of 166 numerical attributes. The data sets represent sets of molecules, and the task is to predict which molecules are musk molecules and which are not.

The Musk data set was introduced to the machine learning community by Dietterich *et al.* (1997), who used the problem to illustrate the so-called multiple-instance problem: an example is not represented by a single tuple but by a set of tuples, only one of which may be relevant. Multiple-instance problems are hard to cope with for propositional learners due to this property.

In the case of the Musk database, molecules are represented by sets of tuples; each tuple represents one possible conformation of the molecule. A molecule is musk if at least one of its conformations is musk.

While the database is not a typical ILP database, because of its orientation towards numerical data and the almost propositional representation, learning in a multiple-instance context can be seen as the simplest form of ILP, as argued by De Raedt (1998). The examples can easily be represented by interpretations, each interpretation simply containing a set of tuples, as Figure A.5 illustrates. Due to practical limitations on the arity of predicates in our Prolog engine we could not use this representation and resorted to the one illustrated in Figure A.6, which makes the link with the original tuples less clear.

Figure A.7 illustrates what a settings file could look like; it is taken from

```
begin(model('MUSK-211')).
  testid('MUSK-211').
  musk.
  df1('211_1+1',46).
  df2('211_1+1',-108).
  df3('211_1+1',-60).
  df4('211_1+1',-69).
  df5('211_1+1',-117).
  df6('211_1+1',49).
  (...)
  df163('211_1+1',156).
  df164('211_1+1',-50).
  df165('211_1+1',-112).
  df166('211_1+1',96).
  df1('211_1+10',41).
  df2('211_1+10',-188).
  df3('211_1+10',-145).
  (...)
  df165('211_1+9',-113).
  df166('211_1+9',97).
end(model('MUSK-211')).
```

Figure A.6: The same molecule as shown in Figure A.6, but using a different representation. Each conformation is described by a single fact $\texttt{df}_i$ for each of its 166 numerical attributes.

one of our experiments with the large Musk data set.

A.6 Mesh

This data set, introduced in the ILP community by Dolšak and Muggleton (1992) and available at the ILP data repository (Kazakov *et al.*, 1996), has its origin in engineering. For many engineering applications, surfaces need to be approximated by a finite element mesh. Such a mesh needs to be fine in some places (in order to assure accuracy), and can be coarser in other places (which decreases the computation cost). The task is to learn rules that predict how fine a mesh should be, by studying a number of meshes.

The data set consists of descriptions of 5 meshes. It is a typical ILP data set in that it contains structural information and a lot of background knowledge is available. On the other hand, it is in essence a regression task (which is less typical for ILP): the resolution of a mesh is a natural number. Most ILP systems just consider the different resolutions to be different classes and learn a classifier.

Due to the fact that the resolution of one edge may depend on its neighbors, the localization of information is relatively bad for this data set. For our experiments with TILDE we have put all the information in the background and kept only the resolution of each edge in the example description itself (together with its identification). A representative part of the data set that is thus obtained is shown in Figure A.8. A sample settings file is shown in Figure A.9. The `tilde_mode(regression)` fact in the file tells TILDE that it should perform regression (by default TILDE performs classification). The `classes` setting is replaced by the `euclid` setting, which tells TILDE on what variable it should perform regression.

A.7 Diterpenes

A detailed description of this application can be found in (Džeroski *et al.*, 1998). The task is to identify substructures in diterpene molecules by looking at the ^{13}C NMR spectrogram of the molecule (peaks occurring in such a spectrogram may indicate the occurrence of certain substructures). There are 23 classes. The entire data set consists of 1503 examples.

This problem is inherently relational, but propositional attributes can be defined (so-called engineered attributes) that are highly relevant.

This data set is not in the public domain. The data were kindly provided to us by Steffen Schulze-Kremer and Sašo Džeroski and are used with permission.

```
classes([musk,nonmusk]).

discretization(bounds(3)).
to_be_discretized(df1(X, N), [N]).
(...)
to_be_discretized(df166(X, N), [N]).

typed_language(yes).
type(df1(conf,int)).
(...)
type(df166(conf,int)).
type(findinterval(_,interval)).
type(between(int,interval)).

interval1([A], [A, inf]) :- !.
interval1([A,B|C], [A,B]).
interval1([A|B], X) :- interval1(B, X).

interval([A|_], [-inf, A]).
interval(L, X) :- interval1(L, X).

findinterval(DF, Int) :-
        F =.. [DF, X, Y],
        discretized(F, [Y], L),
        sort(L, L1),
        interval(L1, Int).

between(X, [-inf, B]) :- X =< B, !.
between(X, [A, inf]) :- A =< X, !.
between(X, [A,B]) :- A =< X, X =< B.

lookahead(df1(Conf,Arg), #(1*100*I: findinterval(df1, I),
                                    between(Arg, I))).
(...)
lookahead(df166(Conf,Arg), #(1*100*I: findinterval(df166, I),
                                      between(Arg, I))).

rmode(5: df1(+X, Y)).
(...)
rmode(5: df166(+X, Y)).

root(df1(X,_)).
```

Figure A.7: A settings file for the Musk data set.

```
begin(background).

  long(a1).  long(a34).  long(a54).  long(b19).  (...)
  usual(a3).  usual(a39).  usual(b11).   (...)
  short(a6).  short(a9).  short(a11).  short(a13).    (...)
  circuit(c15).  circuit(c16).  circuit(c17).   (...)
  half_circuit(a36).  half_circuit(a37).  (...)
  (...)

  neighbour(A,B) :- neighbour_xy_r(A,B); neighbour_yz_r(A,B);
                    neighbour_zx_r(A,B).
  neighbour(A,B) :- neighbour_xy_r(B,A); neighbour_yz_r(B,A);
                    neighbour_zx_r(B,A).
  neighbour_xy_r(a34,a35).
  neighbour_xy_r(a35,a26).
  (...)
  opp(A,B) :- opposite(A,B); opposite(B,A).
  opposite(a11,a3).
  opposite(a9,a3).
  opposite(a31,a25).
  (...)

end(background).

begin(model(a1)).
   resolution(17).
   structure(a).
   mesh_id(a1).
end(model(a1)).

begin(model(a2)).
   resolution(1).
   structure(a).
   mesh_id(a2).
end(model(a2)).

begin(model(a3)).
   resolution(8).
   structure(a).
   mesh_id(a3).
end(model(a3)).
```

Figure A.8: Data representation in the Mesh data set.

```
tilde_mode(regression).
euclid(resolution(X), X).

property(E, long(E)).
property(E, usual(E)).
property(E, short(E)).
(...)
property(E, two_side_loaded(E)).
property(E, cont_loaded(E)).

/* lookahead specifications */

lookahead(opp(E1,E2), X) :- property(E2, X).
lookahead(neighbour(E1,E2), X) :- property(E2, X).
lookahead(eq(E1,E2), X) :- property(E2, X).

rmode(5: long(+E)).
rmode(5: usual(+E)).
(...)
rmode(5: two_side_loaded(+E)).
rmode(5: cont_loaded(+E)).

rmode(5: neighbour(+E, -E2)).
rmode(5: opp(+E, -E2)).
rmode(5: eq(+E, -E2)).

root(mesh_id(E)).
```

Figure A.9: A typical settings file for the Mesh data set. The task is defined as regression on the first argument of the `resolution` predicate.

A.8 RoboCup

This is a data set containing data about soccer games played by software agents training for the RoboCup competition (Kitano *et al.*, 1997). It contains 88594 examples and is 100MB large. Each example consists of a description of the state of the soccer terrain as observed by one specific player on a single moment. This description includes the identity of the player, the positions of all players and of the ball, the time at which the example was recorded, the action the player performed, and the time at which this action was executed. Figure A.10 shows one example.

The classes are high-level representations of the actions of the agents; they are not represented explicitly in an example but computed from the description of the example.

A.9 Poker

The Poker application was first used in (Blockeel *et al.*, 1998a). It consists of several data sets of different size, all generated automatically using a program that mimics the way in which cards are assigned to hands (i.e. the class distributions are as they really occur in Poker games). In the classification problem we consider, there are 7 classes: `pair`, `double_pair`, `three_of_a_kind`, `full_house`, `flush`, `four_of_a_kind`, `nought`. This is a simplification of the real poker domain, where one distinguishes e.g. pair of kings and pair of eights, etc.

Using a program that randomly generates examples for this domain has the advantage that one can easily create multiple training sets of increasing size, an independent test set, etc., which offers a lot of flexibility for experimentation.

An interesting property of this data set is that some classes are very rare, hence a large data set is needed to learn these classes (assuming the data are generated randomly).

Figure A.11 illustrates how one example in the Poker domain can be represented. An example of a typical settings file is given in Figure A.12.

```
begin(model(e71)).
   player(my,1,-48.804436,-0.16494742,339).
   player(my,2,-34.39789,1.0097091,362).
   player(my,3,-32.628735,-18.981379,304).
   player(my,4,-27.1478,1.3262547,362).
   player(my,5,-31.55078,18.985638,362).
   player(my,6,-41.653893,15.659259,357).
   player(my,7,-48.964966,25.731588,352).
   player(my,8,-18.363993,3.815975,362).
   player(my,9,-22.757153,32.208805,347).
   player(my,10,-12.914384,11.456045,362).
   player(my,11,-10.190831,14.468359,18).
   player(other,1,-4.242554,11.635328,314).
   player(other,2,0.0,0.0,0).
   player(other,3,-13.048958,23.604038,299).
   player(other,4,0.0,0.0,0).
   player(other,5,2.4806643,9.412553,341).
   player(other,6,-9.907758,2.6764495,362).
   player(other,7,0.0,0.0,0).
   player(other,8,0.0,0.0,0).
   player(other,9,-4.2189126,9.296844,339).
   player(other,10,0.4492856,11.43235,158).
   player(other,11,0.0,0.0,0).
   ball(-32.503292,0.81057936,362).
   mynumber(5).
   rctime(362).
   turn(137.4931640625).
   actiontime(362).
end(model(e71)).
```

Figure A.10: The Prolog representation of one example in the RoboCup data set. A fact such as `player(other,3,-13.048958,23.604038,299)` means that player 3 of the other team was last seen at position (-13,23.6) at time 299. A position of (0,0) means that that player has never been observed by the player that has generated this model. The action performed currently by this player is `turn(137.4931640625)`: it is turning towards the ball.

```
begin(model(4)).
  card(7,spades).
  card(queen,hearts).
  card(9,clubs).
  card(9,spades).
  card(ace,diamonds).
  pair.
end(model(4)).
```

Figure A.11: An example from the Poker data set.

```
classes([nought,pair,double_pair,three,full_house,flush1,four]).

typed_language(yes).
type(card(rank,suit)).
type(X \= X).

rmode(card(-X,-Y)).
rmode(+X \= +Y).

max_lookahead(2).
lookahead(card(X,Y), card(-U,-V)).
lookahead(card(X,Y), X \= +Z).
lookahead(card(X,Y), Y \= +Z).
lookahead((X \= Y), (+U \= +V)).
```

Figure A.12: A typical settings file for the Poker data set.

Index