Data Mining using Logic

Tightening the Gap between Logic and Specialized Algorithms

Siegfried Nijssen
snijssen@liacs.nl

November 16, 2000

Preface

I remember very well one seminarium that I visited last year: *the seminarium data mining*. It was in this seminarium that I came to learn about Association Rules and Inductive Logic Programming. Association rules were the topic of the first lessons, Inductive Logic Programming was discussed in the later sessions. One thing stroke me immediately: while in the first lessons, all concern was on efficiency and on finding those rules which were interesting, these concerns were completely dropped in the sessions about ILP. Instead, a model was introduced there that was so general that it could not possibly be very efficient. I wondered whether there was not a possibility of merging these two ends together, whether there was not something like a way in between these two counterparts. When it came to choosing a topic for my Master's Thesis, I considered this a nice topic to investigate. The result of this investigation is what you are reading now.

During the project, I have learned a lot. More than can ever be reflected in this thesis. Among the events that I will remember, is a visit to a Dagstuhl Seminar on Data Analysis in Germany. It gave me an impression of the world in which real researchers live. They appeared to be hard working people who rise early in the morning and go to sleep late in the morning. The talking they do in the mean time is really interesting.

The project itself has been the most instructive. Some periods have been very difficult, as I had the impression that no progress was made. These periods have however provided some useful clues to prevent such deadlocks in the future.

I would like to thank prof.dr. J. Kok from Universiteit Leiden for the opportunity he offered to attend the Dagstuhl Seminar; I am grateful for the time he spent in conducting me. I whish to thank dr. W. Kosters from Universiteit Leiden for proofreading this document; this enabled me to improve the quality considerably. Furthermore my expressions of thanks go to Erica, for the good company she was during the months of my Master's Thesis project, and especially to my parents and brothers who gave me the necessary mental support.

Siegfried Nijssen,
Den Haag, Leiden, 25 October 2000.

Summary

Although Machine Learning is an old topic, it has gained renewed attention the last few years as part of the *data mining* process, which is the process of discovering hidden patterns in large amounts of (raw) data. Among the algorithms which are popular, are Association Rule and Decision Tree algorithms. Some reasons for their popularity are:

- the existence of efficient implementations;
- the interpretability of their results.

One class of algorithms within the Machine Learning community which has received less attention in the data mining process, is the class of *Inductive Logic Programming* (ILP) algorithms. These are algorithms which generate a set of logic formulas that can be used to predict or describe a set of data. In situations where data recedes in multiple tables, such an algorithm can be shown to be a necessity. Although the results of ILP algorithms are perfectly interpretable, a major problem with these algorithms is their inefficiency. Some causes for this are the following:

- the inherent complexity of logic. It can be shown that many problems concerning logic formulas are insolvable or NP complete;
- most algorithms depend on the full-fledged logic programming language PROLOG, which is not known for its efficiency;
- many of the algorithms are only predictive in nature and based on an exhaustive search.

Some of these issues have been tackled by the WARMR [Dehaspe, 1998] and TILDE [Blockeel, 1998] algorithms. WARMR is a descriptive algorithm which is based on efficient Association Rule algorithms. TILDE is a predictive algorithm based on decision trees and thus performs a heuristic search.

The first two problems however remain, and, as it appears, also a part of the third problem has not been solved satisfactory by the above mentioned algorithms. In this thesis for each of the problems the possible solutions are investigated, where the main idea is that for some subset of the full-fledged algorithms an efficient implementation may exist.

To circumvent the complexity of logic, some alternatives for the unprovable logic implication relation are investigated. It is found that only the most easy formulas, formulas in which one predicate can only occur once, are practically usable in some situations. Other relations are still found to be NP complete or have graph isomorphism complexity. The choice is made to neglect the problem and introduce a few ad-hoc methods to avoid some of the most frequent situations.

To show that also algorithms without full-fledged PROLOG are practically usable, two algorithms are developped and tested based on WARMR and TILDE: FARMER and GILDER. Although they still exploit some notions which are an inheritance of PROLOG, only a small subset of PROLOG is necessary during the training phase. In return, the resemblance of these algorithms with the popular APRIORI and C4.5 algorithms is shown to be much larger.

Although TILDE is based on decision tree algorithms, it is argued that it still uses some hidden exhaustive search. Both from a complexity point of view as from an accuracy point of view, this is found to be disadvantageous, and a mechanism is introduced to prevent TILDE's cousin GILDER from performing such an exhaustive search too much.

To prove the applicability of FARMER and GILDER, both algorithms are applied to a large financial dataset. Although not all experiments have convincing results, the algorithms show their applicability in these experiments.

During the development of the algorithms, the choice is made to restrict the original algorithms as little as possible. During the project, however, the idea gained strength that much more restrictions could be added to handle real-world databases even more efficiently. In several sections an idea is given about such further restrictions, some of which are inspired by the work of other researchers that was unknown at the beginning of the project.

Contents

Chapter 1

Introduction

In this Thesis some keywords are repeatedly used, for example ‘data mining’, ‘machine learning’ and ‘logic’. This chapter introduces these terms and makes clear which is the place in the “real world” of the algorithms that are discussed in this Thesis. Furthermore, an overview is given of what the reader can expect in the other chapters.

1.1 Overview

In the first section, the concept of *Data Mining* is introduced. An overview is given of its stages. In this Thesis, one of these stages is considered in much detail: the knowledge acquiring stage. The second section discusses some of the tasks which may be performed there. The third section introduces logic and gives an impression why logic can be useful in the data mining process. The fourth section hightlights the need for speed, while the fifth section gives an overview of the other chapters.

1.2 Data Mining

1.2.1 What is it?

During the last decade, computers have conquered the Western world. When you look carefully, they are everywhere. Just buy a candy bar at your local grocery, or look television and heat your favorite pizza: probably some computer is involved in the process. And it is not over yet. High-tech highways are on their way. The cars will be able to drive on their own, and special devices register the cars passing. Let us concentrate on the computers which collect data. Insurance companies, groceries, government, researchers: all have as hobby the collection of data these days. However, once they have collected it, a new question arises: what to do with it? All those data are useless without useful application. That is what is *data mining* about: the discovery of unintended, hidden patterns from huge amounts of sometimes unordered data.

For the moment, let us look at the grocery. Every minute some client might be buying a set of products, and the transaction information is stored. There could be a lot of interesting information in these data. If some product is always bought together with another product, the grocery could anticipate on this behaviour by adjusting its prices. However, the amount of transaction information which is gathered in, say, a month can be very large. Most of the time, it is almost impossible for a human to analyze all the transactions every month. Therefore, computers are introduced not only as a means of collecting data, but also as an aid for analyzing the data. The computer should gather information from the data which is interesting for its human possessor. Because this might sound like looking for a needle in a haystack, the activity was given its name *data mining*. The whole process is called *Knowledge Discovery in Databases* (KDD), which was a term introduced in [Fayyad, 1996].

1.2.2 Stages in the KDD process

The KDD process can be devided into the following stages:

- the collection of data: this is what banks and supermarkets, do when they build a database. It is also a good idea to determine at this moment which knowledge may be interesting, so that this can be taken into account in the next stages;
- the preprocessing of data: most of the time databases are not yet usable for machine learning algorithms. Major problems are for example missing and corrupted values in the tables which have to be corrected first. Sometimes the available data is insufficient, in that case some enrichment has to be performed;
- machine learning: some algorithm is executed to compute those patterns which are considered to be interesting;
- presentation of results: the results of the algorithm should be made understandable, and some indications should be given of the consequences of these results.

Of course, it is not obligatory to perform these stages in sequential order. The application of an algorithm could reveil some problems with the data, which then have to be solved first.

The stage which is considered in most detail here is the third stage (the machine learning stage). The algorithms which are discussed, have however also some influence on the other stages. The proposed algorithms are all based on 'logic', which allows them to treat a larger set of databases than most other algorithms. This should be taken into account in all other stages. In the preprocessing stage, it is for example of course better to take into account which possibilities the algorithm in the next stage offers.

1.3 Which is the task of the algorithms?

What kind of information might be useful for humans? Of course, that depends on the application. There is no such universal interesting information. However, some methods are more common than others. Traditionally, statistics is the main source of information. It is used to answer questions like:

How many customers buy pizza and candy bars at the same time?

For a lot of situations, statistics provides sufficient interesting information. The main problem of statistics is that it presupposes that the user of the system knows which questions to ask the system. The amount of products can be so large that an even more interesting question becomes *which products* are bought together frequently[1]. Recently, algorithms which perform this task have become a popular topic for research. The most well-known algorithm is APRIORI [Agrawal, 1996], which discovers sets of items which are often bought together.

The point of view previously given is merely descriptive. Given some data, one searches patterns which occur often in the data. Other interesting information is more predictive in nature. For example, an insurance company might have a database of clients and their profit to the company. Probably the insurance company is interested to discover a means of identifying the good clients. Given a database of existing clients, the task of the algorithm is to discover a mechanism which can be used to automatically determine whether a client is good or bad. A well-known mechanism to perform such a *classification task* is the decision tree, which accomplishes this by repeatedly asking questions about the client. The construction of decision trees has been studied exhaustively, for example in [Quinlan, 1993a]. Both APRIORI and decision trees will return in the remainder of this document.

1.4 The need for (first order) logic

Most of the algorithms which have been developed for data mining purposes are *propositional*. This means that all the input of the algorithm has to be put into one table. A grocery database which contains transactions only, or an insurance database which contains information on clients, is propositional.

Although this is sufficient in a lot of situations, it is rather restricted. Assume that there is more information: the grocery has a database with product data, or the insurance company also has a database with agency data. It would be a pity if we could not exploit that information too. Otherwise, maybe better predictors could result or even more interesting patterns would be found.

(First order) logic is a formal language which can be used to express knowledge that concerns true or false facts. Using this language, the following set of examples can be expressed in a mathematical clearly defined way:

[1] according to some criterion

> "Everybody who has a car, also has a driver's license."
> "There are people who like orange juice and don't like coffee."
> "Somebody is a good customer if he lives in the neighboorhood and shops every day."

As can be seen from these examples, first order logic adds the possibility to link tables together, and thus exactly adds the feature which is desired to link together databases. The main topic which is studied in this document, is the extension of well-known algorithms, such as APRIORI and decision trees, with first order logic. For example, the extended APRIORI algorithm could find patterns like:

> "Many people buy cheese and drinks from the same country."

1.5 The need for speed

A lot of databases suffer from one important factor: their size. It takes a lot of time to process them. First order logic has another problem: its expressiveness. The number of patterns expressible in logic is infinite in most cases, or very large otherwise. By combining logic with large databases, a setup seems to result which is hardly usable or completely intractable. Of course, this is not the case, but surely care has to be taken during the construction of the extended algorithms. This is the main challenge which is faced in this Thesis: not only to construct algorithms which are first order extensions of well-known algorithms, but also to create reasonably fast implementations.

1.6 What can be expected in this Thesis?

The remainder of this document is structured as follows:

- Chapter 2 summarizes some theory of logic which is necessary when reading this document.
- Chapter 3 explains the notion of Inductive Logic Programming and gives some example systems, which all have contributed ideas to the algorithms discussed in the next chapters.
- Chapter 4 explains the APRIORI and WARMR algorithms and the relation of the latter to ILP.
- Chapter 5 introduces a new WARMR-like algorithm and compares it to the original WARMR algorithm.
- Chapter 6 introduces the theory of decision trees. It summarizes the TILDE algorithm.
- Chapter 7 discusses a new TILDE-like algorithm, and compares it to other classification algorithms.

- Chapter 8 discusses an experiment which has been done using the two algorithms.
- Chapter 9 concludes.

Chapter 2

Logic

The representation language which is used throughout this thesis, is based on first-order logic and especially on clausal logic. In this chapter some fundamentals of logics are discussed. The main focus is on implication relations between formulas and, consequently, on equivalence relations. These relations are the formalization of what people mean when they say that one conclusion can be drawn from the other or that two reasonings mean the same. Especially when formulas are automatically generated, which will be the case for all systems discussed in this thesis, it is important to have a means of filtering formulas which do not add any information as they mean the same as other generated formulas.

2.1 Overview

The chapter is organized as follows. In the first section, a simplified definition of general first-order logic formulas is given. In the second section some of the most well-known restrictions, such as clausal logic, are explained. These sections provide the reader with the basic background knowledge which is necessary to understand the remainder of this thesis. Those readers who are familiar with logic, can safely skip the two sections. Other readers may also consult the textbook [Reeves, 1990], which gives an excellent description of the details of logic formulas and their meaning.

The remainder of the chapter is dedicated to an overview of relations between formulas. In any system which is based on logic, the theory of relations is exploited. In this section, those relations are investigated which are used by the systems that are discussed in this thesis.

2.2 Full first-order logic formulas

The basis of logic theory is the logic formula. The definition of a logic formula is recursive, and starts with the smallest entities.

Definition 2.2.1 *The smallest entities which exist in logic formulas are constants and variables. Variables are denoted by strings beginning with an upper case letter; all other strings are constants.*

Example The following are constants: *chips*, a, 1. These are variables: X, *ProductBought*. □

The existence of variables distinguishes first-order logic from propositional logic and adds the possibility to define links between several kinds of information.

The smallest entities can be combined into a term. Terms are also defined recursively.

Definition 2.2.2 Terms *are defined as follows:*

1. *constants are terms;*
2. *variables are terms;*
3. *if f is a function of arity n, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.*
4. *nothing else is a term.*

Example The following are terms: *chips*, a, 1, X and *ProductBought*. If *price* is a function of arity 1, then $price(chips)$, $price(a)$, $price(1)$, $price(X)$ and $price(ProductBought)$ are also terms. □

Terms come together in logic formulas.

Definition 2.2.3 Well formed formulas *(or just* formulas*) are defined as follows:*

1. *if p is a predicate symbol of arity n and $t_1, \ldots t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula, called an* atom *or* literal. *Predicate symbols will be denoted starting with a lower case character.*
2. *if ϕ is a formula, then $\neg\phi$ is a formula. If ϕ is a literal, then $\neg\phi$ is also called a literal.*
3. *if ϕ and ψ are formulas, then $\phi \wedge \psi$ (a conjunction), $\phi \vee \psi$ (a disjunction), $\phi \rightarrow \psi$ (an implication) and $\phi \leftrightarrow \psi$ (an equivalence) are formulas.*
4. *if ϕ is a formula and X is a variable, then $\exists X \phi$ and $\forall X \phi$ are formulas.*

Example If *buys* is a predicate of arity 2, then the following are some examples of atoms: $buys(chips, a)$, $buys(a, chips)$, $buys(chips, X)$, $buys(X, chips)$, $buys(chips, price(ProductBought))$. Both $buys(X, chips)$ and $\neg buys(X, chips)$ are literals.

Well formed formulas are: $\exists C\ client(C) \wedge buys(C, chips)$ and $\forall P\ buys(1, P) \rightarrow good(P)$. The symbol $\exists$ is called the existential quantifier and should be read as "there exists", the $\forall$ symbol is called the universal quantifier and means "for all"; the formulas could be read as:

> There exists a client who buys chips.

and

> For all products it is true that, when they are bought by client one, they are good.

The latter means the same as:

> All products bought by client one are good.

□

In the remainder of this chapter, it is assumed that formulas are *closed*: there are no variables which are not quantified. For such formulas, it is possible to formally define when they are true. most of the time, it depends upon the interpretation what is the truth value of a formula.

Definition 2.2.4 *An* interpretation I *of a set of formulas* Σ *is a function which performs the following tasks*[1]*:*

- *for the functions occuring in* Σ*, it defines how they map to all possible constants;*
- *for the predicates in* Σ*, it defines how they map from constants to truth values.*

An interpretation function I can be used to compute whether a formula is true or not, following the rules for quantifiers and connectives. Essential in this procedure is the notion of a variable assignment.

Definition 2.2.5 *A function* V *is a variable assignment for a formula* ϕ *when it assigns constants to some of the variables in* ϕ*. When* ϕ *is a formula,* $V(\phi)$ *is the formula* ϕ *in which all occurrences of variables in* V *have been replaced by the corresponding constants.*

Variable assignments are necessary when evaluating quantifiers. For example, to evaluate an existential quantifier, a variable assignment should be found which makes the remainder of the formula true. The notation of a variable assignment will often be used in the remainder of this document, and should not be mixed up with the concept of *substitution*, which will be discussed later on in this chapter.

Definition 2.2.6 *A model of a set of formulas* Σ *is an interpretation which makes all formulas in* Σ *true.*

Models play an important role in logic. A lot of definitions depend upon their meaning. The most important is that of logical implication.

[1] This definition is a simplified version of the definition in [Nienhuys-Cheng, 1997]. There an extra domain D is used; all functions and predicates are mapped to a meaning in D.

Definition 2.2.7 *Let Σ be a set of formulas, and ϕ a formula. Then ϕ is a* logical consequence *of Σ, written as $\Sigma \models \phi$, if every model of Σ is also a model of ϕ.*

It has been shown that to prove logical consequence, it suffices to find a reasoning which deduces ϕ from the set Σ. This corresponds to the procedure which most scientists are using: to prove a new theorem, it suffices to deduce it from existing theorems and axioms. However, for an automated computer, Church has proved that the logical consequence relation is undecidable. Thus, for arbitrary Σ and ϕ, there can not exist a computer program which finds the deduction. Only weaker relations between formulas can be computed. The last section of this chapter summarizes some of these relations. This section concludes with some properties of logic formulas which will often be used.

Definition 2.2.8 *Formulas ϕ and ψ are equivalent, denoted with $\phi \leftrightarrow \psi$, if and only if $\phi \models \psi$ and $\psi \models \phi$.*

Proposition 2.2.1 *Assume that ϕ and ψ are logic formulas (possibly with free variables), and Q is a set of quantifiers which quantifies variables in ϕ and ψ. Then the following equivalences hold:*

- $Q\,\phi \rightarrow \psi \leftrightarrow Q\,\neg\phi \vee \psi$.
- $Q\,\neg(\phi \vee \psi) \leftrightarrow Q\,\neg\phi \wedge \neg\psi$.
- $Q\,\neg(\phi \wedge \psi) \leftrightarrow Q\,\neg\phi \vee \neg\psi$.

Finally, the definition of a substitution is important. A substitution is the main operation which can be applied to formulas.

Definition 2.2.9 *A* substitution *θ is a finite set of the form*

$$\{x_1/t_1, \ldots, x_n/t_n\}, n \geq 0, \tag{2.1}$$

where the x_i are distinct variables and the t_i are terms. x_i/t_i is called a binding *for x_i. A substitution is* ground *if no t_i contains variables. If ϕ is a formula, then $\phi\theta$ is formula ϕ in which all free variables $x_i (x_i/t_i \in \theta)$ have been replaced by the their corresponding term t_i.*

Definition 2.2.10 *A* unifier *for two formulas ϕ and ψ is a substitution θ such that $\phi\theta = \psi\theta$. If such θ exists, ϕ and ψ are unifiable.*

Example The following two formulas are unifiable:

$$\phi = buys(X, a) \vee buys(Y, X) \tag{2.2}$$

$$\psi = buys(a, X) \tag{2.3}$$

using the following unifier:

$$\theta = \{X/a, Y/a\}. \tag{2.4}$$

Also the following two formulas are unifiable:

$$\phi = buys(X, a) \quad (2.5)$$
$$\psi = buys(Y, a) \quad (2.6)$$
$$\quad (2.7)$$

using the following unifiers:

$$\theta_1 = \{X/a, Y/a\} \quad (2.8)$$
$$\theta_2 = \{X/f(a), Y/f(a)\} \quad (2.9)$$
$$\theta_3 = \{X/f(f(a)), Y/f(f(a))\} \quad (2.10)$$
$$\vdots$$

□

As can be seen in this example, the set of unifiers can be infinite. For practical purposes, a more precise unifier is searched: the *most general unifier*.

Definition 2.2.11 *The* most general unifier *of two formulas ϕ and ψ is the unifier θ, such that for every other unifier θ_1 of ϕ and ψ there is a substitution θ_2 for which $\theta\theta_2 = \theta_1$.*

Example For the following two formulas:

$$\phi = buys(X, a) \quad (2.11)$$
$$\psi = buys(Y, a) \quad (2.12)$$

these are some most general unifiers:

$$\theta_1 = \{X/Z, Y/Z\} \quad (2.13)$$
$$\theta_2 = \{Y/X\} \quad (2.14)$$
$$\theta_3 = \{X/Y\} \quad (2.15)$$
$$\vdots$$

□

Most general unifiers thus still are not unique. The only variation, however, is in the names of the variables which are used.

2.3 Restricted first-order logic

Within the world of logic, several restricted versions of first-order logic are popular. *Clausal logic* turns out to have some theoretical advantages which make it a useful tool in mathematical proofs about logic. *Horn clauses* are the basis of PROLOG, the most well-known programming language based on logic. The first restriction which is considered here, is the restriction of valid formulas to *clauses*.

Definition 2.3.1 *A* clause *is a finite disjunction of zero or more literals.*

Example The following is a clause:

$$buys(X, a) \vee buys(X, b) \vee \neg buys(X, c) \vee \neg buys(X, d). \tag{2.16}$$

□

Note that clauses do not have quantifiers. In situations where this is desired, closed formulas are obtained from clauses by adding universal quantifiers for each free variable in the clause. More often, clauses are written using a (reversed) implication:

$$buys(X, a) \vee buys(X, b) \leftarrow buys(X, c) \wedge buys(X, d). \tag{2.17}$$

Using the properties discussed in Theorem 2.2.1, it can be shown that this notation is equivalent.

The importance of clauses is due to the existence of a proof procedure based on *resolution* which uses the notation of clauses. A proof procedure is a definition of valid steps which can be used to prove logical consequence. However, although clauses restrict first-order logic, it has been shown that $\Sigma \models \phi$ is not decidable even if Σ and ϕ only contain clauses.

A further restriction are *Horn clauses.*

Definition 2.3.2 *A* definite program clause *is a clause containing one positive, and zero or more negative literals. A* definite goal *is a clause containing only negative literals. A* Horn clause *is either a definite program clause, or a definite goal.*

When using the notation for clauses, one could equivalently say that a Horn clause has at most one atom before the arrow.

Example The following are Horn clauses:

$$buys(X, a) \leftarrow buys(X, c) \wedge buys(X, d) \tag{2.18}$$
$$buys(X, a) \tag{2.19}$$
$$\leftarrow buys(X, c) \wedge buys(X, d)$$

□

The atoms before the arrow are called the head of the clause; the atoms after the arrow are the body. Horn clauses are used because of the *SLD-resolution* proof procedure which can be used for such clauses. Although logical consequence for Horn clauses still turns out to be undecidable, this procedure allows the usage of logic for computational purposes, as is done in PROLOG. The remainder of this section is a short description of PROLOG.

In PROLOG, a Horn clause is denoted as

```
buys(X,a) :- buys(X,c), buys(X,d).
```

or

```
buys(X,a).
```

or

```
:- buys(X,c), buys(X,d).
```

Instead of an arrow, the `:-` symbol is used; a $\wedge$ connective is replaced with a comma and every sentence must be finished with a dot. As PROLOG is a programming language, an executional meaning has to be given to the clauses. This can be characterized by top-down, left-to-right evaluation and by backtracking.

To prove a definite goal, all the literals in the goal are scanned from left to right. If for some assignment of variables a literal is true, the next literal is evaluated. If some literal can not be satisfied, the system backtracks over a previous variable assignment and looks for another assignment, until all possibilities have been tried. As soon as a variable assignment is found for which all literals are true, the definite goal is proved and the system returns `Yes`, with a vairable assingment if required. Furthermore, PROLOG uses the *closed world assumption*: if no variable assignment could be found, then it is assumed the definite goal can not be satisfied and the system returns `No`.

To check the truth of a literal, a top-down pass is made through the database. Of every clause whose head can be unified with the literal, the body (after applying a most general unifier) is considered as a definite goal which has to be proved. If the definite goal does not succeed, the search continues for another unifying head in the database.

Also in PROLOG functions are allowed. However, in some situations, this complicates systems unnecessarily. Therefore sometimes a simplified version of PROLOG is used: DATALOG, which misses the possibility to use functions[2].

2.4 Relations between formulas

As was stated in the previous section, the $\models$ relation is undecidable. Given a set of formulas, it is thus not possible to remove those formulas which are implied by other formulas and thus contain redundant information. Especially when large sets of formulas are automatically generated, it is still desirable to remove as much redundant formulas as possible. In this section, some approximations of the $\models$ relation are studied which could be used for this purpose.

The most well-known relation is the θ-subsumption relation.

Definition 2.4.1 *Let C en D be sets of literals; C θ-*subsumes *D if there exists a substitution θ such that $C\theta \subseteq D$.*

Example These are some examples of θ-subsumption:

[2] And, consequently, the possibility to use lists!

- $C = \{buys(X, a)\}$ subsumes $D = \{buys(1, a), buys(1, b)\}$.
- $C = \{buys(X, Y), buys(X, a)\}$ subsumes $D = \{buys(1, a)\}$.

□

Note that θ-subsumption is also a relation between sets of atoms. The definition can easily be extended to a conjunction or disjunction of atoms. In that case, one would say that $\phi = buys(X, a)$ subsumes $\psi = buys(1, a) \wedge buys(1, b)$.

It is important to notice the relation between logical implication and θ-subsumption. For example, consider logical formulas of a restricted form, called *queries* [Dehaspe, 1997].

Definition 2.4.2 *A formula ϕ is a* query *if it is a disjunction of atoms and only has existential quantifiers.*

most of the time, formulas are required which do not have free variables. In that case all variables are assumed to be quantified existentially. If it is clear that the formula should be closed, the quantifiers are therefore not displayed. Substitutions are applied to such queries as if there are no quantifiers.

Theorem 2.4.1 (Dehaspe, 1998b) *Let formulas ϕ and ψ be queries. If ϕ subsumes ψ, then $\psi \models \phi$. This case is also denoted as $\psi \succeq \phi$.*

Example Let $\phi = buys(X, a)$ and $\psi = buys(1, a) \wedge buys(1, b)$, then $\phi\{X/1\} \subseteq \psi$. Furthermore, $\psi \models \phi$, as for the variable assignment $V(X) = 1$ can be chosen. □

It is clear that the reverse relation of Theorem 2.4.1 can not hold. Queries will return in more detail in the next chapters.

Although θ-substitution is definitely a more restricted relation than logical consequence, it still turns out to be very hard.

Theorem 2.4.2 (Kietz, 1994) *θ-subsumption is NP-complete.* [3]

Proof Let (V, S) be any 3SAT instance, where:

$$
\begin{aligned}
V &= \{x_1, \ldots, x_n\}, &(2.20)\\
S &= \{\{l_{11}, l_{12}, l_{13}\}, \ldots, \{l_{m1}, l_{m2}, l_{m3}\}\}, &(2.21)\\
\{l_{11}, \ldots, l_{m3}\} &\subseteq \{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}. &(2.22)
\end{aligned}
$$

Let f map Boolean literals $\{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$ to $2n$ first-order-logic variables $X_1, \ldots, X_n, NX_1, \ldots, NX_n$. The 3SAT instance (V, S) is solvable iff D

[3] In [Kietz, 1994] it is suspected that this theorem was already proved by Baxter (1977), but this unpublished paper could not be found, so they give a new proof, of which a modified version is given here.

subsumes C for the following two sets of atoms:

$$\begin{aligned} D = \quad \{ \quad & p(f(x_n)), \ldots, p(f(x_n)), \\ & ne(f(x_1), f(\overline{x_1})), \ldots, ne(f(x_n), f(\overline{x_n})), \\ & c(f(l_{11}), f(l_{12}), f(l_{13})), \ldots, c(f(l_{m1}), f(l_{m2}), f(l_{m3})) \\ \} \quad & . \end{aligned} \tag{2.23}$$

$$\begin{aligned} C = \quad \{ \quad & p(0), p(1), \\ & ne(0,1), ne(1,0), \\ & c(1,1,1), c(0,1,1), c(1,0,1), c(1,1,0), c(1,0,0), c(0,1,0), c(0,0,1) \\ \} \quad & . \end{aligned} \tag{2.24}$$

It is clear that $|D| + |C|$ is linear with respect to $|V| + |S|$. The atoms p ensure that every variable x_i is given a value while the ne atoms ensure that x_i has the opposite value of $\overline{x_i}$. Finally, every element $l \in S$ is mapped to an atom c which in turn must be mapped by substitution θ to an atom which would satisfy l. □

Note that the proof does not use functions in atoms. Thus, the result is even valid for DATALOG clauses.

The subsumption relation induces an equivalence relation.

Definition 2.4.3 *Let C and D be sets of atoms. Then $C \sim D$ iff $C \succeq D$ and $D \succeq C$.*

If the sets of atoms are queries, then two equivalent queries $C \sim D$ are also logically equivalent, $C \leftrightarrow D$. The reverse of course does not hold.

Plotkin defines reduced clauses as clauses containing no redundant literals with respect to θ-subsumption:

Definition 2.4.4 (Plotkin, 1970) *A clause D is reduced iff $C \subseteq D$ and $C \sim D$ imply $C = D$.*

Gottlob and Fermüller [Gottlob, 1993] have shown that deciding whether a clause D is reduced is co-NP-complete, and that finding a reduced clause D requires at least a linear (in $|D|$) number of calls to an NP oracle.

Less restrictive is the following relation, which will be called *atom set isomorphism* in this Thesis[4].

Definition 2.4.5 *Let C and D be sets of atoms. Then D and C are (atom set) isomorphic, denoted with $D \simeq C$, if there exist substitutions θ_1 and θ_2 such that $C\theta_1 = D$ and $D\theta_2 = C$.*

Example The following sets are isomorphic:

$$C = \{buys(X, a), buys(X, Y)\}, \tag{2.25}$$

$$D = \{buys(Y, Z), buys(Y, a)\}. \tag{2.26}$$

Using substitution $\theta_1 = \{X/Y, Y/Z\}$. □

[4] Maybe some other name is used in literature for this relation, but this was unknown to the author.

Note that $C\theta_1\theta_2 = C$, and thus $\theta_2 = \theta_1^{-1}$. As a result, θ_1 and θ_2 can only map from variables to variables. It is clear that isomorphic queries are also logically equivalent.

It is interesting to compare atom set isomorphism with θ-subsumption for reduced sets of atoms.

Theorem 2.4.3 *Let C and D be reduced sets of atoms. Then $C \simeq D$ iff $C \sim D$.*

Proof "$\Rightarrow$": this is clear as $C\theta_1 \subseteq D$ and $D\theta_2 \subseteq C$.
"$\Leftarrow$": as $C \succeq D$, $C\theta_1 \subseteq D$, and as $D \succeq C$, $C\theta_1\theta_2 \subseteq C$. Let $C' = C\theta_1\theta_2$. Because $C' \subseteq C$ and $C\theta_1\theta_2 \subseteq C'$, also $C \sim C'$ holds (by definition), and then $C = C'$ because C is reduced. Thus $C\theta_1\theta_2 = C$. In the same way, $D\theta'_1\theta'_2 = D$. As $|C\theta_1| = |C|$ and $C\theta_1 \subseteq D$, $|D| \geq |C|$. As $|D\theta'_1| = |D|$, $|C| \geq |D|$, and finally $|D| = |C|$. By combining $|C\theta_1| = |D|$ and $C\theta_1 \subseteq D$, $C\theta_1 = D$ is shown. This proves that $D \simeq C$. $\square$

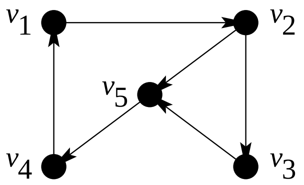
This theorem proves that atom set isomorphism is weaker than subsumption equivalence. As a result, it is hoped that it is easier to compute whether this relation holds. However, atom set isomorphism also turns out to be a complex relation. In the sequel, we will prove that atom set isomorphism has the same complexity as graph isomorphism. Köbler, Schöning and Torán, for example, have studied graph complexity exhaustively [Köbler, 1993], but did not succeed in proving that this problem is in P nor that it is NP-complete. They only showed that it is more probable that graph isomorphism is in P than that it is in NP. No more recent paper was found which shows any fundamental changes since the appearance of that book.

Definition 2.4.6 *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two directed graphs. G_1 and G_2 are isomorphic (denoted with $G_1 \simeq G_2$) if there exists a bijection β between nodes such that for all $(v_i, v'_i) \in E_1$: $(\beta(v_i), \beta(v'_i)) \in E_2$ and for all $(v_i, v'_i) \in E_2$: $(\beta^{-1}(v_i), \beta^{-1}(v'_i)) \in E_1$.*

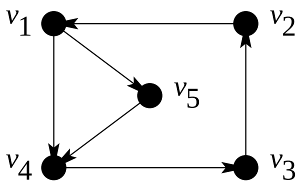
Lemma 2.4.1 *If atom set isomorphism is in complexity class X which is at least polynomial, then graph isomorphism is at most in X.*

Proof This can be proved by showing that an algorithm which decides whether two atom sets are isomorphic, can also be used to decide whether two graphs are isomorphic, and by showing that the transformation used is polynomial.

First, it must be checked that both graphs have the same number of nodes and edges. It is clear that this operation is linear. Next each graph is translated into a set of atoms as follows. Let f be a function which maps every node to a variable. Then for every node $v \in V$ an atom $node(f(v))$ is added and for every edge $(v_i, v'_i) \in E$ an atom $edge(f(v_i), f(v'_i))$. It is clear that this operation is linear too. If the atom set isomorphism algorithm succeeds, it found a mapping between variables of the atom sets, and consequently a mapping between the nodes such that the graphs are isomorphic. $\square$



$node(V_1), node(V_2), node(V_3), node(V_4), node(V_5),$
$edge(V_1, V_2), edge(V_2, V_3), edge(V_3, V_5),$
$edge(V_2, V_5), edge(V_5, V_4), edge(V_4, V_1)$



$node(V_1), node(V_2), node(V_3), node(V_4), node(V_5),$
$edge(V_2, V_1), edge(V_1, V_4), edge(V_1, V_5),$
$edge(V_3, V_2), edge(V_4, V_3), edge(V_5, V_4)$

Figure 2.1: Two graphs and their corresponding atom sets.

Example Figure 2.1 shows two isomorphic graphs and two atom sets which are permutations of each other. A substitution to obtain the second set from the first is $\{V_1/V_2, V_2/V_1, V_3/V_5, V_4/V_3, V_5/V_4\}$. □

Definition 2.4.7 *Let $G_1 = (V_1, E_1, \lambda_1)$ and $G_2 = (V_2, E_2, \lambda_2)$ be two (node) labeled directed graphs. G_1 and G_2 are isomorphic (denoted with $G_1 \simeq G_2$) if there exists a bijection β between nodes such that for all $(v_i, v_i') \in E_1$: $(\beta(v_i), \beta(v_i')) \in E_2$ and for all $(v_i, v_i') \in E_2$: $(\beta^{-1}(v_i), \beta^{-1}(v_i')) \in E_1$. Furthermore for all $v_i \in V_1$: $\lambda_1(v_i) = \lambda_2(f(v_i))$ and for all $v_i \in V_2$: $\lambda_2(v_i) = \lambda_1(f^{-1}(v_i))$.*

Lemma 2.4.2 *If labeled graph isomorphism is in complexity class X which is at least polynomial, then atom set isomorphism is at most in X.*

Proof This can be proved by showing that an algorithm which decides whether two graphs are isomorphic, can also be used to decide whether two atom sets are isomorphic, and by showing that the transformation used is polynomial.

First, it must be checked that the atom sets have the same number of atoms and the same number of variables. It is clear that this check can be performed linearly. Next, each atom set is translated into a graph.

The atom sets are translated in two steps. First the variables are translated. Each variable is mapped by function f into one node which has label v. Next, each atom is translated into a set of interconnected nodes: if $p(\sigma_1, \ldots, \sigma_n)$ is an atom, then this is mapped to an interconnected set of n nodes which has the following n labels: if σ_i is a constant, then the label is $p \cdot i \cdot c \cdot \sigma_i$. If σ_i is a variable, then the label is $p \cdot i \cdot v$, and the node is connected to node $f(\sigma_i)$.

If some algorithm determines that two graphs are isomorphic, then also a substitution can be found by considering how variable nodes are mapped to each other. As isomorphic nodes must have the same label (and the labels contain all

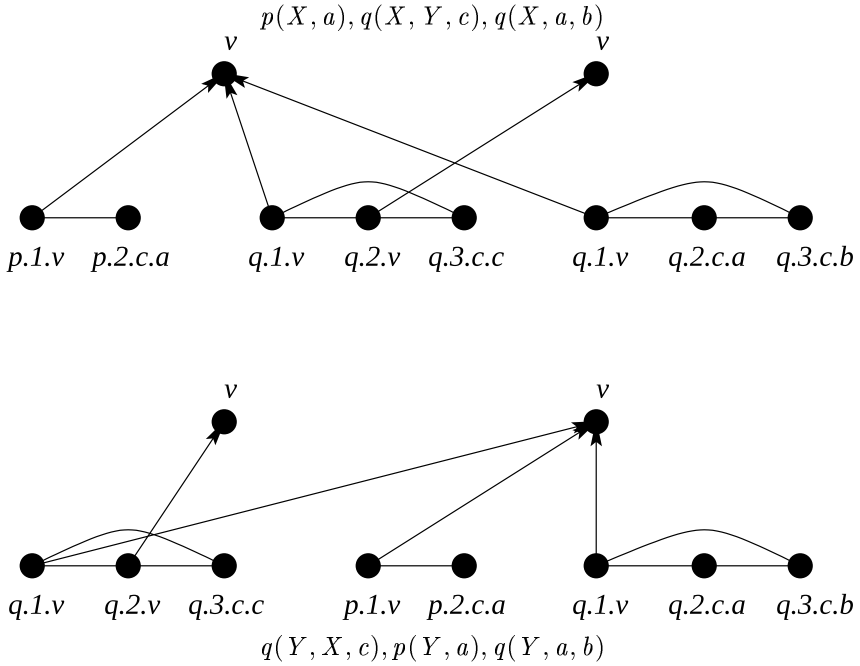


Figure 2.2: Two atom sets and their corresponding graphs.

information about the atom and the position in the atom), and as isomorphic nodes must have connections to the same set of labels, only atoms which are isomorphic are mapped to each other. □

Example Figure 2.2 shows two isomorphic atom sets and their corresponding graphs.

□

Lemma 2.4.3 *Labeled graph isomorphism is of the same complexity as graph isomorphism.*

Proof "⇒" Add the same label to each node.
"⇐" Collect the set of labels, and give each label a number starting with 3. Then each original node of label number k can be represented using a set of k new interconnected nodes. Define one of these nodes to be the connecting point. Furthermore, for each edge add one node, which is connected to both connecting points of the original nodes. Then one of the two edges belonging to an original edge can not be mapped to an edge in an original node, as the two ending nodes are not connected to one identical node. In the same way, an edge in an original node is never mapped to an edge which belongs with an original edge. Nodes within original nodes are only mapped to each other if all

nodes (but the connection point) are connected to the same number of nodes, and thus the labels are equal.

Furthermore, this translation method is linear in the number of edges, nodes and labels. □

Theorem 2.4.4 *Atom set isomorphism and graph isomorphism are of the same complexity.*

Proof By combination of Lemmas 2.4.1, 2.4.2 and 2.4.3. □

In applications, clauses are most of the time restricted to (PROLOG) Horn clauses. In the semantics of PROLOG, an unbound variable is repeatedly bound to all possible values and the remainder of the clause is evaluated using this assignment. Switching two literals can have a major impact on this evaluation mechanism, as a variable can now be introduced by another predicate. It can be interesting to know whether two clauses are not only isomorphic, but also introduce variables in the same way. To this purpose, we define the following isomorphism.

Definition 2.4.8 *Two ordered atom sets C and D are introduction order isomorphic iff:*

- *The corresponding unordered atom sets are isomorphic;*
- *Let θ be such that $C\theta = D$. Let $a = p(\sigma_1, \ldots, \sigma_k)$ be some atom in C, and let n be its position in C. If σ_i is a variable and σ_i does not appear in any atom before n, then the position of $a\theta \in D$ is such that $\sigma_i\theta$ does not appear before that position.*

  The same also holds for the reverse.

Example The following two atom sets are introduction order isomorphic:

$$\phi = p(X,Y), q(X), z(Y) \tag{2.27}$$
$$\psi = p(X,Y), z(Y), q(X) \tag{2.28}$$

The following two atom sets are not introduction order isomorphic:

$$\phi = p(X,Y), q(X), z(Y) \tag{2.29}$$
$$\psi = q(X), z(Y), p(X,Y) \tag{2.30}$$

□

Theorem 2.4.5 *Ordered atom sets isomorphism is of the same complexity as graph isomorphism.*

Proof This can be proved in the same way as atom set isomorphism. Lemma 2.4.1 is such that it also holds for ordered atom sets: all variables are introduced by the node predicate. For the reverse, a modification of the mapping of atom sets to graphs is necessary. Let $p(\sigma_1, \ldots, \sigma_k) \in C$ be an atom. If σ_i is a variable which does not occur earlier in C, and thus the atom in fact introduces this variable, then this parameter is mapped to a node with label $p \ldots i \ldots v*$. The star indicates that this atom is the first atom that uses the corresponding variable. By graph isomorphism, it can only be mapped to a node which also contains the star and thus introduces the variable. □

For ordered atom sets, we can also define another isomorphism relation, which is based on permutations.

Definition 2.4.9 *Two ordered atom sets C and D are introduction order permutations (denoted with $C \simeq_p D$) iff:*

- *The corresponding unordered atom sets are equal;*
- *Let $a = p(\sigma_1, \ldots, \sigma_k)$ be some atom in C, and let n be its position in C. If some σ_i is a variable and σ_i does not appear in any atom before n, then the position of $a \in D$ is such that σ_i does not appear before that position.*

This relation is easily checked. The first requirement can be checked polynomially by performing a linear search for every atom of C in D. The second requirement is performed linearly by building a data structure which saves the variables currently introduced.

A relation which can not be used as an approximation of logical equivalence for queries, is the unification relation.

Example Consider these two queries:

$$\phi = \exists X\, p(a, X) \tag{2.31}$$
$$\psi = \exists X\, p(X, a) \tag{2.32}$$

and assume that we intend to use the unification relation to approximate logical implication between these two formulas. These queries can be unified (after removing the quantifiers, otherwise unification would not be possible) using the unifier $\theta = \{X/a\}$. However, $\phi \not\models \psi$ and $\psi \not\models \phi$. □

Unification can neither be used for universally quantified formulas. This will be illustrated using an example.

Example Consider these two formulas:

$$\phi = \forall X\, p(a, X) \tag{2.33}$$
$$\psi = \forall X\, p(X, a) \tag{2.34}$$

These can be unified (after removing the quantifiers), using the unifier $\theta = \{X/a\}$. However, $\phi \not\models \psi$ and $\psi \not\models \phi$. □

2.5 Summary

In this chapter the formal definitions of logic formulas, (Horn) clauses and queries were given. Some of the theory concerning them was summarized, including models and proof procedures. An overview was given of relations between formulas. As implication is undecidable, approximations were investigated. Apart from well-known approximations such as θ-subsumption, also the complexity of less powerful procedures was analyzed. An isomorphism based on the renaming of variables was found to have the same complexity as graph isomorphism. Only one relation was found to be polynomial: atom sets which are permutations of each other. This is of course not surprising.

The importance of these relations will become clear in Chapter 5. When large sets of formulas are generated, it is important to have some procedure to remove those which are redundant. In this chapter it was shown that this is very difficult from a complexity point of view.

Chapter 3

Inductive Logic Programming

In the previous chapter, the programming language PROLOG was briefly mentioned. PROLOG was designed to give humans a means of expressing an algorithm without going into too much implementation details. As research proceeded further, the idea rose that it might also be possible for computers to generate such programs. Given a set of examples which illustrate the result of the algorithm, the task of the computer could also be to construct the algorithm. Especially when the algorithm is unknown, such a style of programming would have a lot of appeal. The researchers which investigated this idea called it *Inductive Logic Programming* (ILP), as it is in a sense logic programming by automated induction from examples. In this chapter an overview is given of the aspects which have been studied in the field of ILP. These aspects are illustrated on some example systems. This introduction to ILP provides the reader the possibility to determine the place within the world of ILP of the algorithms discussed in this thesis. Readers who are interested, can also consult the textbook [Nienhuys-Cheng, 1997]. This book has been the source of much of the contents in this chapter.

3.1 Overview

This chapter is composed of two main sections. The first section gives an overview of several aspects of ILP. Aspects which are discussed include the target of the algorithms, the knowledge which algorithms may use to obtain this target and the algorithms themselves. In the second section, it is illustrated how these aspects apply to well-known ILP algorithms. Algorithms which will be covered in some detail here are PROGOL, FOIL, LINUS and CLAUDIEN. Two other algorithms, WARMR and TILDE, will only be touched upon, as they will be given in more detail in the following chapters.

3.2 Dimensions of ILP

3.2.1 Interactivity

ILP systems exist in two flavors of 'interactivity'. *Incremental* learners do not have a fixed database of knowledge. At any moment, one might add knowledge to the system which is used afterwards. Most of these systems have to be guided in their knowledge discovery and thus are *interactive*.

At the other side are the *batch learner* systems. These systems need all their information before they start. Typically they do not need interaction with the user to discover knowledge. In the sequel only such non-interactive batch learners are considered.

3.2.2 What is learned?

Most ILP systems are based on standard Prolog. Consequently, the result of the algorithms consists of a Prolog program, which is a set of Horn clauses. What the resulting Prolog programs express, and how complex the programs may be, is generally described by assigning to the algorithm one of these three labels: single predicate learner, multiple-predicate learner or non-monotonic learner.

In the normal problem setting, ILP is used to learn one specific predicate. For example consider the client database of a bank. For each client the bank stores information like income, debts and property values, together with a field indicating whether the client is a good client or not. The bank would like to have a definition of a good client. Converting the problem to logic, one would say that the bank wants to have a definition of the predicate *goodclient*; in this definition the functions *income*, *debt* and *propertyvalue* can be used, together with the predicates *hasproperty* and $\geq$. One of the result found by the system could be:

$$\begin{aligned} goodclient(X) \leftarrow \; & income(X) \geq 10000, \\ & hasproperty(X, Y), \\ & propertyvalue(Y) \geq 1000000. \end{aligned} \tag{3.1}$$

The system thus only discovers clauses which have *goodclient* in the head.

An extension of this setting is the multiple-predicate learner approach. For a set of predicates a system of this type discovers definitions. It performs this task incrementally: it first learns one predicate, which can then be used in the definition of a new predicate, and so on. One of the problems which has to be solved by such a system thus is the choice of the order in which predicates are learned.

In both the normal problem setting and the multiple-predicate learner approach, there is a separation between the background knowledge and the examples of the target predicate. The background knowledge together with the created hypothesis should entail the examples, which means that the examples are not necessary any more as they can be reconstructed from the hypothesis and the background knowledge.

Another approach to ILP is the *non-monotonic* problem setting. In this approach no single predicate has to be learned; rather, one searches for a set of clauses which can be satisfied for many examples. In the banking example, such a system could discover the following rule

$$\begin{aligned} \mathit{debt}(X) \geq 10000 \quad \leftarrow \quad & \mathit{hasproperty}(X, Y), \\ & \mathit{propertyvalue}(Y) \geq 100000, \\ & 10000 \geq \mathit{income}(X), \end{aligned} \tag{3.2}$$

which states the hypothesis that everybody who has an income lower than 10000 and a property which costs more than 100000, has a debt larger than 10000. As can be seen in the example, the rules which are discovered need not define any predicate, but reflect some interesting properties of the examples. There is no distinction between the background knowledge and the examples. As a result, there is no way to reconstruct the examples from the rules which have been found.

3.2.3 Which knowledge is used?

The systems discussed in the previous section use knowledge to discover new knowledge. How is the *apriori* knowledge represented? There exist two kinds of knowledge:

1. *Intensional knowledge* is knowledge which can only be expressed using quantified formulas. For example the following knowledge is intensional:

$$\mathit{grandmother}(X, Y) \leftarrow \mathit{parent}(X, Z), \mathit{parent}(Z, Y), \mathit{female}(Y), \tag{3.3}$$

   which states the well-known fact that the female parent of a parent is a grandmother.

2. *Extensional knowledge* is knowledge which is expressed using facts. No quantifier has to be used in order to express the knowledge in logic. The following knowledge would be extensional:

$$\begin{aligned} & \mathit{mother}(\mathit{beatrix}, \mathit{juliana}) \\ & \mathit{mother}(\mathit{willem}, \mathit{beatrix}) \\ & \neg \mathit{mother}(\mathit{willem}, \mathit{juliana}); \end{aligned} \tag{3.4}$$

   it states the fact that (within the domain of people concerned) Juliana is the mother of Beatrix and Beatrix is the mother of Willem; Juliana however is not the mother of Willem.

For some predicates extensional knowledge thus defines the constants for which the predicate certainly is true or false. What about the truth values for other constants? In most systems, the *closed world assumption* is used, which means that the predicate is considered to be false in all cases which are not mentioned.

Most systems use a mix of intensional and extensional knowledge. Examples are always given extensional. Background knowledge however can be extensional as well as intentional.

3.2.4 How is knowledge found?

The search for logic formulas can be performed into two directions: starting from the examples or starting from the most general formula $\leftarrow$. The first approach is called the bottom-up approach, the second is the top-down approach. Both search techniques will be illustrated below in the normal problem setting using the family domain example. The target predicate is *daughter*; the positive examples of this predicate are:

$$daughter(beatrix, bernard), daughter(juliana, Wilhelmina). \tag{3.5}$$

The background knowledge consists of the facts:

$$\begin{aligned} BK \quad = \quad & parent(Wilhelmina, juliana), parent(juliana, beatrix), \\ & parent(bernard, beatrix), female(Wilhelmina), \\ & female(beatrix), female(juliana). \end{aligned} \tag{3.6}$$

In the sequel, predicate d will be used as an abbreviation of the predicate *daughter*; p stands for *parent* and f for *female*. The constants *beatrix*, *bernard*, *juliana* and *wilhelmina* will be replaced by b, v, j and w, respectively. The background knowledge becomes:

$$BK \quad = \quad p(w,j), p(j,b), p(v,b), f(w), f(b), f(j). \tag{3.7}$$

The bottom-up method starts with one example as hypothesis, for example $H = d(b,v) \leftarrow BK$. This hypothesis is too specific and needs to be generalized. One method to do this is to compute the *relative least general generalization* of H and the next example, $d(j,w) \leftarrow BK$. The *rlgg* is obtained by computing for each pair of predicates from both clauses a generalization which is created by replacing mismatching constants by a variable. For the example this is:

$$\begin{aligned} d(V_{b,j}, V_{v,w}) \quad \leftarrow \quad & p(V_{w,j}, V_{b,j}), p(V_{v,w}, V_{b,j}), p(V_{j,v}, b), p(w,j), p(j,b), p(v,b), \\ & f(V_{b,w}), f(V_{w,j}), f(V_{b,j}), f(w), f(b), f(j). \end{aligned} \tag{3.8}$$

After removing redundant predicates from this formula (including BK), the following formula results:

$$d(X,Y) \leftarrow p(Y,X), f(X), \tag{3.9}$$

which already is a correct definition of *daughter*.

Assuming the normal problem setting, the top-down method starts with $H = d(X,Y) \leftarrow$ as initial hypothesis. It scans all examples to determine whether it is correct. Using the closed world assumption, there are a lot of examples for which this hypothesis is not correct, so it has to be refined. Which refinements can be made? To this purpose, most systems introduce a *language bias*. The bias declares which clauses can be found and which can not. Assume that the bias is such that the following predicates can be used in a refinement:

$$p(X, Y), f(X), \tag{3.10}$$

where X and Y should be variables previously used, then the following refinements will all be tested:

$$\begin{array}{lcl} d(X,Y) & \leftarrow & p(X,Y) \\ d(X,Y) & \leftarrow & p(Y,X) \\ d(X,Y) & \leftarrow & p(X,X) \\ d(X,Y) & \leftarrow & p(Y,Y) \\ d(X,Y) & \leftarrow & f(X) \\ d(X,Y) & \leftarrow & f(Y) \end{array} \tag{3.11}$$

None of these hypotheses is correct, so a refinement of each has to be attempted. For example, these are refinements of clause (3.11):

$$d(X,Y) \leftarrow p(Y,X), f(X)$$
$$d(X,Y) \leftarrow p(Y,X), f(Y) \tag{3.12}$$

Clause (3.12) will also be tested and proves to be correct.

Although top-down and bottom-up search techniques seem rather disjunctive, some systems use a combination of both, such as PROGOL.

3.3 Example systems

In the next sections, some well-known ILP systems are discussed with respect to the dimensions previously given. It shows how these systems fit in the dimensions and should give more insight in the practical applications of the techniques.

3.3.1 PROGOL

PROGOL [Muggleton, 1995] is a non-interactive, multiple predicate learner. Background knowledge can be specified intensional as well as extensional. As search technique a hybrid top-down, bottom-up method is used.

The system starts with one example and uses this example to build a most specific hypothesis, which is a hypothesis that entails the example considered, and is so specific that any other hypothesis which may entail this example is more general than this one. The addition of atoms in this phase is guided by *mode* declarations.

The general syntax of a *mode* declaration is $modeh(n, atom)$ or $modeb(n, atom)$, where n is a natural number or inf. A *modeb* declaration specifies an atom which can be added to the body of a clause; a *modeh* declaration specifies an atom that can be used as head. Variable n indicates how many times an atom can be added to a clause or can be used as head. The *atom* consists of a predicate applied to terms; a term can be a place-marker or a function applied to terms. A place-marker is either $+type$, $-type$ or $\#type$. For example reconsider some of the predicates in Formula 3.10. The bias could consist of the following *mode* declarations:

$$\begin{gathered} modeh(1, g(+Person, +Person)) \\ modeb(2, p(-Person, +Person)) \\ modeb(1, +Person = +Person) \\ modeb(3, f(+Person)) \end{gathered} \tag{3.13}$$

A new predicate g, which stands for *grandmother*, has been added. Only one type is necessary, which is called $Person$.

As head of the clause the mode is used which subsumes the example that is considered. Variables are assigned a value according to this subsumption substitution. After this, atoms are added to the body of the clause according to the mode declarations. A $-type$ marker should introduce a new variable in the clause, a $+type$ marker should use an existing variable, while a $\#type$ place marker is replaced by a constant. All atoms are added which can be satisfied by some variable assignment for the shorter clause.

As an example, this mechanism is applied here to the family case. To illustrate some concepts of PROGOL better, the new predicate g is used. The example of this predicate which PROGOL looks at first, is

$$g(beatrix, wilhelmina). \tag{3.14}$$

The *hmode* declaration which subsumes is

$$mode(g(+Person, +Person)). \tag{3.15}$$

This generates the following clause and variable assignment:

$$g(P_1, P_2) \leftarrow \tag{3.16}$$
$$V(P_1) = beatrix, V(P_2) = wilhelmina \tag{3.17}$$

Valid expansions are, according to the modes[1]:

$$P_1 = P_2, p(P_3, P_1), p(P_4, P_2), f(P_1), f(P_2). \tag{3.18}$$

Using the variable assignment and the closed world assumption on the dataset, $p(P_4, P_1)$, $f(P_1)$ and $f(P_2)$ can be proved. Variable P_4 may be assigned the value *juliana* or *bernard*. The current clause is:

$$g(P_1, P_2) \leftarrow p(P_4, P_1), f(P_1), f(P_2). \tag{3.19}$$

[1] $P_i = P_i$ is not generated.

Valid new expansions are now

$$P_4 = P_1, P_4 = P_2, p(P_5, P_4), f(P_4). \tag{3.20}$$

Using the background knowledge in combination with the closed world assumption, the following clause is found which can still be proved, with variable assignment $V(P_5) = \textit{wilhelmina}$ and $V(P_4) = \textit{juliana}$:

$$g(P_1, P_2) \leftarrow p(P_4, P_1), f(P_1), p(P_5, P_4), f(P_4). \tag{3.21}$$

Eventually one further expansion is found.

$$g(P_1, P_2) \leftarrow p(P_4, P_1), f(P_1), p(P_5, P_4), f(P_4), P_2 = P_5, f(P_5). \tag{3.22}$$

It is clear that this clause is too specific. It specifies for example that the parent must be female.

The specific hypothesis is used as a bias in a top-down search to determine the most general hypothesis. Atoms from the specific hypothesis are added to an empty clause in an A* search which is guided by a heuristic based on the number of examples covered by the clause and the number of examples violated by the clause. Once that the most general hypothesis is found which covers many positive examples and none of the negative examples, the search stops. All positive examples which are covered by the clause are removed. For the remaining examples the process is repeated, and for one example a new specific clause will be built.

In the example, this means that, when also other family relations are given, those occurrences of f which are too specific are removed.

The mode mechanism introduced in Progol has afterwards been used in many other systems, especially to guide the search in top-down systems.

3.3.2 Foil

Foil [Quinlan, 1993] is one of the most popular ILP systems. Because it does not perform an exhaustive search, but a heuristic one, it is more efficient and hence more practical than most other systems. Foil can be characterized as a non-interactive, top-down, single predicate learner. Background knowledge can only be specified extensional and might contain positive and negative examples.

A summary of the Foil algorithm is given in Figure 3.1. Essentially it keeps building clauses as long as there are positive examples which can not be explained. While building clauses, there are two kinds of literals which are added:

- gainful literals, which are literals that cover as much positive examples and as less negative examples as possible, according to a criterion based on information theory;

- determinate literals, which are literals that introduce new variables and are true for all facts. These literals will never satisfy the information criterion, but are necessary because they introduce new variables.

Let $T(X_1, \ldots, X_n)$ be the target predicate
Let F^+ be the set of positive examples of T
Let F^- be the set of negative examples of T
while there are examples in F^+ **do**
    $C := T(X_1, \ldots, X_n) \leftarrow$
    $F_2^- := F^-$
    $F_2^+ := F^+$
    while there are examples in F_2^- and C is not too complex **do**
        find the best new literal L for C to split F_2^- and F_2^+
        $C := C \wedge L$
        remove from F_2^+ and F_2^- the facts which are not satisfied
        add all possible new determinate literals to C
    remove redundant literals from C
    add C to the theory
    $F^+ := F^+ - F_2^+$

Figure 3.1: The FOIL algorithm.

It is clear that this mechanism is greedy and could add redundant literals, such as determinate literals which introduce variables that are never used. To simplify the theory, these redundant literals are removed.

FOIL has additional mechanisms to allow recursive theories and to avoid infinite determinate literal additions.

A lot of variations of FOIL have been developed. Some systems are more expressive than the original one, while others introduce better heuristics; for example the FFOIL system [Quinlan, 1996b] focuses at learning functions.

3.3.3 LINUS

LINUS [Lavrač, 1994] is a non-interactive single predicate learner. Background knowledge can be specified intensionally as well as extensionally. The search technique is however neither top-down nor bottom-up. The idea of LINUS is that a lot of ILP problems can be translated into attribute-value learning problems. To solve such a problem, a well-known technique like decision trees can be used. The resulting classifier can be transformed back into first order logic and is simplified if possible.

Logic formulas thus are used because of their expressive ease and the possibility to simplify some results when applying axioms of first order logic. A lot of first order logic problems however can not be converted to attribute-value learning problems and LINUS is not capable of solving these problems.

3.3.4 CLAUDIEN

CLAUDIEN [De Raedt, 1996] is a non-interactive non-monotonic learner. Background knowledge can be specified intensionally as well as extensionally. The search for interesting clauses is performed top-down, starting with the clause $\leftarrow$. As bias the DLAB formalism is used, which has the following syntax:

$$dlab\_template(atoms \leftarrow atoms), \tag{3.23}$$

where *atoms* is an atom or a DVAR declaration:

$$n_1 - n_2 : [atoms_1, atoms_2, \ldots, atoms_n]. \tag{3.24}$$

All clauses will be generated which have between n_1 and n_2 atoms from the specified list of *atoms*. For example consider the following *dlab_template*:

$$dlab\_template(d(X,Y) \leftarrow 2{-}2 : [1{-}1 : [p(X,Y), p(Y,X)], 0{-}2 : [f(X), f(Y)]]. \tag{3.25}$$

This template will define the following space of clauses:

$$\begin{array}{lcl}
d(X,Y) & \leftarrow & p(X,Y) \\
d(X,Y) & \leftarrow & p(Y,X) \\
d(X,Y) & \leftarrow & p(X,Y), f(X) \\
d(X,Y) & \leftarrow & p(X,Y), f(Y) \\
d(X,Y) & \leftarrow & p(X,Y), f(X), f(Y) \\
d(X,Y) & \leftarrow & p(Y,X), f(X) \\
d(X,Y) & \leftarrow & p(Y,X), f(Y) \\
d(X,Y) & \leftarrow & p(Y,X), f(X), f(Y)
\end{array}$$

This space will be searched in a breadth-first manner: any clause which is not specific enough, will be specialized by adding an atom after all clauses of the same size have been tested.

In the previous example, CLAUDIEN behaves just like a single predicate learner and tests show a comparable performance. Contrary to the predicate learners CLAUDIEN however allows for a disjunction of predicates in the head (thus allowing all clauses, not only Horn clauses).

CLAUDIEN does not only distinguish itself by its bias, but also by the way it defines knowledge, which it splits up into background information and observations. Each observation consists of independent facts. The validity of each clause is tested on all observations separately[2]. Only if a clause is valid in a sufficient number of observations, the clause is considered to be general enough. The usability of this mechanism can be seen in a supermarket example. Let us consider each client as an observation, then the supermarket could have the following observations about its clients:

[2]As each example can be considered as a different interpretation for the clause considered, this mechanism is called *learning from interpretations*.

observation 1	$buys(chips), buys(beer), buys(pizza), livesat(thehague).$
observation 2	$buys(chips), buys(beer), buys(potatoes), livesat(leiden).$
observation 3	$buys(chips), buys(wine), livesat(wassenaar).$
observation 4	$buys(chocolat), buys(wine), livesat(voorschoten).$

Furthermore, the background knowledge is:

$$poor(thehague), poor(leiden), rich(wassenaar), rich(voorschoten).$$

and the *dlab_template* is defined to be:

$$dlab\_template(atoms \leftarrow 0-3 : [atoms, livesat(X), 1-1 : [poor(X), rich(X)]])$$

where *atoms* is defined as

$$\begin{gathered} 1-6 : [buys(chips), buys(beer), buys(pizza), \\ buys(potatoes), buys(wine), buys(chocolate)] \end{gathered}$$

Finally, assume that a clause should succeed on all examples, then the following dependencies are found:

$$\begin{array}{rcl}
buys(chips); buys(wine) & \leftarrow & true \\
buys(chips); buys(chocolate) & \leftarrow & true \\
buys(beer); buys(wine) & \leftarrow & true \\
buys(pizza); buys(potatoes); buys(wine) & \leftarrow & true \\
buys(chips) & \leftarrow & livesat(A), poor(A) \\
buys(beer) & \leftarrow & livesat(A), poor(A) \\
buys(wine) & \leftarrow & livesat(A), rich(A) \\
buys(pizza); buys(potatoes) & \leftarrow & livesat(A), poor(A)
\end{array}$$

Redundant formulas are thus removed. Note furthermore that the bias also includes clauses like $buys(chips) \leftarrow buys(chips)$. These trivial clauses are also not given as result.

As is clear from this example, CLAUDIEN can be useful when performing knowledge discovery. By splitting up all knowledge in partitions called observations, an efficient implementation is possible which only has a small part of a database in main memory. The observation mechanism however also has a major drawback: it is impossible to find relations between the observations, unless all observations are put together in one huge observation. This however would deteriorate performance and would turn the algorithm virtually impossible to execute.

3.3.5 WARMR

WARMR [Dehaspe, 1998] is a non-interactive non-monotonic learner. Background knowledge can be specified intensionally as well as extensionally. The search is exhaustive. As bias an adapted version of the mode formalism of PROGOL is used. WARMR will be covered in detail in the next chapter.

3.3.6 Tilde

Tilde [Blockeel, 1998] is a non-interactive learner which can also be used to learn single predicates. Background knowledge can be specified intensionally as well as extensionally. The search is heuristic. As bias an adapted version of the mode formalism of Progol is used. Tilde will be covered in full detail in Chapter 6.

3.4 Summary

In this chapter some aspects of Inductive Logic Progamming were discussed. By listing these aspects, the advantages of ILP became clear.

The results of an ILP algorithm are represented as logic programs. As logic programs are very close to human language, the results of the algorithms are better to interpret than those of some other learning algorithms.

ILP allows for the incorporation of background knowledge in a general algorithm. Background knowledge can be used to enhance the quality of the learned rules considerably.

Finally, some learning strategies were discussed and illustrated on example systems. It was shown how some algorithms perform an exhaustive search, such as Progol, while others perform merely a heuristic search, such as Foil.

This chapter should have given the reader the insight to determine the place within the world of ILP of the algorithms discussed in the next chapters.

Chapter 4

WARMR and APRIORI

In the past few years, "association rules" have become very popular in the knowledge discovery world. For one part this can be explained by their straightforward meaning. For the other part this is due to the existence of an efficient level-wise algorithm which can be used to find such rules. In their basic form association rules are however somewhat restricted. From the world of ILP came the idea that it may also be possible to combine ILP with association rules, thus giving association rules the expressive wealth of logic. This chapter discusses the level-wise association rule algorithm and some of the problems of this algorithm. The algorithm which was the result of the cooperation between ILP and assocation rule researchers is presented subsequently.

4.1 Overview

The chapter is organized as follows. In the first section association rules are introduced. The popular APRIORI algorithm [Agrawal, 1996] is explained and an overview is given of the problems with which association rule researchers are confronted. The second section explains the WARMR algorithm [Dehaspe, 1997], which is the first-order expansion of APRIORI. The section contains a description of the algorithm, and illustrates how WARMR can also be applied to other association rule problems. The last section summarizes.

4.2 Association Rules

In this section, the notion of association rules is introduced. The first subsection gives an idea about what association rules really are. Also the formal definition is given. The second subsection goes into the details of the algorithm which is at the core of most algorithms which discover association rules: APRIORI. The last subsection contains a selected list of topics.

4.2.1 Definition

Before giving the formal definition, the idea will be explained using an example.

Example Given is a database of supermarket *transactions* which for each client contain the set of products that the client buys. Then an association rule could be the following:

if a client buys coffee, that clients buys also milk and sugar.

The task is to find such rules which are valid for a lot of clients. □

In the process of discovering such rules, two steps are taken. First those sets of products are searched which are often bought together. These sets are then translated into rules. It is in the first task that APRIORI plays its important role.

Given a set of m items and a database of transactions, which are subsets of the items, the task of APRIORI is to find those sets of items which occur more often than is specified by the *minimum support* parameter of the algorithm. Formally this is stated as follows: let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be the set of database items. Each transaction T in the database $\mathcal{D}$, has a unique identifier and contains a set of items, also called an item set, or k-item set if its size is k. The support of an item set is the number of transactions in $\mathcal{D}$ which contains the item set. Frequent item sets are those item sets whose support exceeds the minimum support threshold, and the task of the algorithm is to find all frequent item sets.

After APRIORI has found the frequent item sets, the item sets are converted into rules. In principle all subsets of the frequent item set are generated to this purpose[1]. The subset is used as antecedent, while the complement is used as consequent. For each such obtained rule, the *confidence* is computed: the support of the total set divided by the support of the subset. Those rules for which the confidence exceeds a minimum confidence parameter are part of the result of the association rule algorithm.

4.2.2 APRIORI

An overview of the APRIORI algorithm is given in Figure 4.1. It discovers the frequent item sets in order of increasing size, starting with the frequent item sets of size 1. After a set of candidates of size k has been generated (C_k), the frequencies of all item sets in it are counted. All item sets which are not frequent, are removed; the remaining item sets are put into a temporary set L_k which is used to create a new set of candidates. Figure 4.2 shows the algorithm which is used to create new candidates. First it creates new candidates by combining two smaller item sets. However, of some of these candidates it can already be determined *apriori* that they are not good: these candidates are removed in the *prune*-step, which follows. The idea is that all subsets of a frequent item set must also be frequent: this is the so-called monotonicity.

[1] There are some efficient strategies which can be used to avoid an exhaustive search.

$C_1 := \{$ 1-item sets $\}$
$k := 1$
while $C_k \neq \emptyset$ **do**
  apriori-count(C_k)
  $L_k = \{c \in C_k | c.count \geq minsup\}$
  C_{k+1} := apriori-gen(L_k)
  $k := k + 1$
$Answer = \bigcup_k L_k$

Figure 4.1: Algorithm APRIORI

$C_{k+1} := \{\{c.item_1, \ldots, c.item_{k-2}, c.item_{k-1}, c.item_k\}|$
  $\{c.item_1, \ldots, c.item_{k-1}\}, \{c.item_1, \ldots, c.item_{k-2}, c.item_k\} \in L_k,$
  $c.item_k > c.item_{k-1}\}$

prune step:
for all item sets $c \in C_{k+1}$ **do**
  for all k-subsets s of c **do**
    if $(s \notin L_k)$ **then**
      delete c from C_{k+1}

Figure 4.2: **apriori-gen**: the algorithm for creating candidates in APRIORI.

4.2.3 Issues about Association Rules

Although the general ideas of association rules and APRIORI are quite straightforward, there are a lot of issues which are still topic of further research. This section contains a selected list of some research topics.

Implementation

The previous description of APRIORI is only superficial. There are still some implementation decisions to be made which can affect the performance of the algorithm.

In [Agrawal, 1996], a possible implementation is discussed which uses a *hash-tree*. A hash-tree is a tree which has hash-tables in the internal nodes. Each bucket in the node points to a child of the node. To add a new item set to the tree, the tree is traversed from the root to a leaf. At depth d, the next node is determined by hashing on the dth item of the item set. In this way, item sets which share a prefix also share their path from the root. The count procedure is implemented in a similar way. Given an item set of size k, all its subset of size $k - 1$ are generated and searched in the tree.

Other implementations of the APRIORI algorithm followed that of [Agrawal, 1996]. In [Pijls, 1999] a slightly different procedure is described, in which the hash nodes are replaced by completely filled and ordered arrays. This has some consequences on the counting mechanism previously described. Instead of a mechanism which first generates all subsets of the transaction and next searches for them in the tree, now a mechanism is used which first walks through the items in a node and next checks for their presence in the transaction; of those items which do appear, the child nodes are considered. In this way, also all subsets of the transaction are counted. It is clear that in case the number of items in a node is large and the number of items in a transaction is small, this mechanism could be disadvantageous. However, it is preferable because of its simplicity. No hash function is used and nodes are very small.

A further aspect which is described in more detail in [Pijls, 1999], is the way in which one tree is obtained from the other. A new set of candidates can easily be generated by copying leaves. Let us consider a set of leaves which are brothers. All these brothers share their path from the root and should be combined into new candidates. It is clear that for each leaf those brother items which are frequent and which have a higher number, should be copied as children.

Another well-known variant of APRIORI is APRIORI-TID. In APRIORI-TID with every item set the possibly large list of transaction *id*s is stored that contain the item set. Counting is not necessary, as the size of the set equals the support. All effort is now in the generation of candidates. If a new candidate is generated from smaller item sets, a new list of *tid*s has to be obtained by computing the intersection of the lists of the smaller item sets.

Weighted items

One major problem of APRIORI is that it generates a lot of rules when the support is low. Sometimes the amount of rules even exceeds the size of the data! There is thus a strong need for mechanisms which avoid the usage of a low support or which find those rules that are interesting within the set of rules. In the following some situations are discussed and the solutions which have been proposed.

One situation is when supermarkets are interested in those products which are bought together with rare items. To find such rules, it would be necessary to decrease the minimum support. One solution was offered by [Liu, 1999]. The proposal is to assign every item a weight. The weight of an item set is defined to be the highest weight in the set; the support of the item set is the product of the weight with the count. In this way, sets with rare items can be given a higher ranking.

A disadvantage of this weighting procedure is however that the efficient APRIORI algorithm can no longer be applied. A set can get a lower support when a rare item is removed, which violates the monotonicity on which APRIORI is built. Other solutions therefore use another approach in which still a minimum support is possible.

Taxonomies

In most supermarket situations, a *taxonomy* can be built of the products. For example: Nike and Adidas are shoes, cola and juice are drinks, and so on. Apart from rules with items, in this case also rules can be built with types of products. These rules can be used to prune other rules.

A solution is the one proposed in [De Graaf, 2000]. Consider the following item set:

$$\{Nike, juice\} \tag{4.1}$$

A new item set can be obtained by replacing items with a more general term, for example:

$$\{shoes, juice\} \tag{4.2}$$

An expected value of $support(\{Nike, juice\})$ can be obtained by computing the following expression:

$$support(\{shoes, juice\}) \times \frac{support(Nike)}{support(juice)}, \tag{4.3}$$

If the expected value approximates the real value, then this item set is not interesting. This is however but one approximation method. Another, maybe less ingenius one, could be:

$$support(\{shoes, juice\}) \times (\#\ products\ of\ type\ juice)^{-1} \tag{4.4}$$

This approximation has the advantage that it also computes an expected value in case *all* items are replaced by an ancestor. Care should be taken when several products are replaced by the same ancestor.

Pruning without background knowledge

The methods above assume some background knowledge which is incorporated into the algorithm. Also without any additional knowledge there are possibilities to distinguish interesting rules. We present a possible method here[2].

When is a frequent item set interesting? The answer to this question is: when it has a support that is not expected, or: if its support could not have been determined without counting in the database. To determine whether an item set is interesting, thus reduces to the question whether there exists a method to predict the support.

Which is a method to predict the support, if no background knowledge is given? There is an easy way to compute an expected support, which is based on the observation that the support of an item set is in fact the chance that the set of items occurs together in a transaction. If the support of set I_1 is s_1, and the support of set I_2 is s_2, then the support of $I_1 \cup I_2$ would be $s_1 \cdot s_2$ if the sets of items are disjoint and independent of each other. If indeed $I_1 \cup I_2$ has support

[2] The author is unaware of any articles which have already described this method.

$s_1 \cdot s_2$, this larger set is not very interesting. Following this idea, an item set I can thus be the pruned if:

$$\exists S \subset I : |support(S) \cdot support(I \backslash S) - support(I)| < \epsilon, \tag{4.5}$$

for some parameter $\epsilon > 0$.

To find S an A* search can be performed. The distance measure in Formula 4.5 can be used as heuristic. The speed of the algorithm increases with the parameter ϵ, even as the number of (uninteresting) sets in the result.

Each item set generated by this mechanism is guaranteed to be unexpected with respect to the parameter and the information available. If the parameter ϵ is small, this also means that it is possible to approximate for every item set whether this set is frequent or not, by finding a partition of the set in subsets such that:

1. the subsets are interesting, where single frequent items are also interesting;

2. the multiplication of the supports of these sets is higher than the minimum threshold.

Theoretically, this could mean that the set of frequent item sets can be reconstructed from the smaller interesting set.

4.3 WARMR

In this section a first-order implementation of APRIORI is discussed: WARMR [Dehaspe, 1997]. As it is an extension, it borrows some important ideas from APRIORI, but also incorporates some new ideas. This section is organized analogous to the previous section.

The first subsection defines the first-order equivalent of the frequent item set. It will appear that WARMR has a bias specification mechanism which is missing for ordinary association rules. This bias specification mechanism is discussed in the second subsection. The third subsection then introduces the algorithm itself.

The last subsections highlight some aspects of WARMR: the fourth subsection discusses the link between frequent itemsets and frequent queries in more detail; the fifth section gives an illustration of how WARMR can be applied to one of the extensions of association rules: taxonomies. The sixth subsection provides some additional insight in WARMR by comparing it to CLAUDIEN.

4.3.1 Frequent Queries

The first-order extension of item sets were chosen to be queries; the algorithm searches for frequent queries. Note that queries are existentially quantified and can have free variables. These free variables are bound by a key predicate, as follows:

$$key(X_1, \ldots, X_n), Q(X_1, \ldots, X_n), \tag{4.6}$$

where $X_1, \ldots, X_n$ are the unbound variables which occur in query Q. All queries which are discovered during the search, should have the same key predicate and thus must have the same set of free variables. The key predicate is used to define the frequency of a query in a similar way as the frequencies of item sets. The frequency of a query is defined as follows:

Definition 4.3.1 *Let $key(X_1, \ldots, X_n), Q(X_1, \ldots, X_n)$ be a query, then the frequency of the query is the number of different bindings of $\{X_1, \ldots, X_n\}$ for which $key(X_1, \ldots, X_n)$ and the query Q can be proved by a* PROLOG *engine.*

4.3.2 Bias

As was already stated in the previous chapter, as bias mechanism an adapted version of the mode mechanism of PROGOL is used, which is called the WRMODE mechanism. The bias consists of a set of mode declarations and a key. Mode declarations define the atoms which can be used to incrementally build queries. A mode declaration is an atom which has *mode constraints* as parameters:

$+$ this position is strictly *input*: when adding this atom to a query, at this position a variable should be used which already occurred;

$-$ this position is strictly *output*: when adding this atom to a query, at this position a variable should be used which does not occur earlier;

$+-$ this position can be both input and output;

c at this position the constant c should be filled in.

Furthermore, also in WRMODE it is possible to use a type name after a plus or minus. This will restrict the usage of variables in a straightforward way: a plus position is only allowed to be filled in with a variable which has the correct type as defined by a minus.

Example Reconsider a supermarket case. The bias could be defined as follows:

- a key $client(-C)$;
- a set of mode declarations

$$\{buys(+C, beer), buys(+C, chips), livesat(+C, -P), poor(+P), rich(+P)\}.$$

To illustrate the results of the WARMR algorithm, these mode declarations will be used to discover the frequent queries of a small database of PROLOG facts.

The knowledge could be described as:

$$\begin{aligned}&poor(thehague), poor(leiden), rich(wassenaar), rich(voorschoten),\\&client(1), buys(1, chips), buys(1, beer), buys(1, pizza), livesat(1, thehague),\\&client(2), buys(2, chips), buys(2, beer), buys(2, potatoes), livesat(2, leiden),\\&client(3), buys(3, chips), buys(3, wine), livesat(3, wassenaar),\\&client(4), buys(4, chocolate), buys(5, wine), livesat(5, voorschoten).\end{aligned} \tag{4.7}$$

One of the queries which is found if the minimum support is two, is

$$client(A), buys(A, chips), \tag{4.8}$$

because for three values of A (which means three clients) $buys(A, chips)$ is true. In the example, these are some frequent queries discovered by WARMR:

$$\begin{aligned}Q_1 &= client(A), buys(A, beer).\\Q_2 &= client(A), buys(A, chips).\\Q_3 &= client(A), livesat(A, B).\\Q_4 &= client(A), buys(A, beer), buys(A, chips).\\Q_5 &= client(A), buys(A, chips), buys(A, beer).\\Q_6 &= client(A), buys(A, beer), livesat(A, B).\\Q_7 &= client(A), buys(A, chips), livesat(A, B).\\Q_8 &= client(A), livesat(A, B), poor(B).\\Q_9 &= client(A), buys(A, beer), buys(A, chips), livesat(A, B).\\Q_{10} &= client(A), buys(A, beer), livesat(A, B), poor(B).\\Q_{11} &= client(A), buys(A, chips), livesat(A, B), poor(B).\\Q_{12} &= client(A), buys(A, beer), buys(A, chips), livesat(A, B), poor(B).\end{aligned} \tag{4.9}$$

□

In the example, it can be seen that the mode mechanism creates a lot of superfluous queries. Query Q_4 can be created from query Q_1, while the logically equivalent query Q_5 can be obtained from query Q_2. This a major disadvantage of the WRMODE declaration mechanism. Other formalisms, such as DLAB, do not suffer from this problem. However, these mechanisms are also more restricted. It will be one of the tasks of WARMR to detect such redundancy.

4.3.3 Algorithm

Figure 4.3 shows the main loop of the WARMR algorithm. Note the strong similarity with the algorithm in Figure 4.1. The main differences are in the warmr-gen and warmr-count procedures. Warmr-gen can be found in Figure

$C_1 := \{key\}$
$k := 1$
while $C_k \neq \emptyset$ **do**
  warmr-count(C_k)
  $L_k = \{c \in C_k | c.count \geq minsup\}$
  $I_k = C_k \backslash L_k$
  C_{k+1} := warmr-gen(L, I)
  k:=k+1
$Answer = \bigcup_k L_k$

Figure 4.3: Algorithm WARMR

$C_{k+1} := \emptyset$
for all queries $c \in L_k$ **do**
  for all $c' \in$ WRMODE refinements(c) **do**
    Add c' to C_{k+1} unless:
      (a) there is a $e \in \bigcup_{i \leq k} I_i$: $e \succeq c'$, or
      (b) there is a $e \in \bigcup_{i \leq k} L_i$: $e \sim c'$

Figure 4.4: **warmr-gen**: the algorithm for creating candidates in WARMR.

4.4. It shows the mechanisms which are used to prevent redundancy within the set of candidates. Both are based on θ-subsumption.

Step (a) corresponds to the prune step of the APRIORI algorithm. It checks whether this specific query is not θ-subsumed by (and thus is a logical consequence of) a more general, but infrequent query. In that case, this query can never be frequent and can be be pruned *apriori*.

Step (b) checks if a candidate query is not subsumption equivalent with a previous query, and thus should be discarded.

Example Let us apply the warmr-gen procedure to a situation which has been drawn from the supermarket example:

$$L_3 = \{client(C), livesat(C, P_1), poor(P_1)\} \tag{4.10}$$

The set of WRMODE refinements is $\{R_1, R_2, R_3, R_4, R_5\}$, with:

$$R_1 = \{client(C), livesat(C, P_1), poor(P_1), buys(C, beer)\} \tag{4.11}$$
$$R_2 = \{client(C), livesat(C, P_1), poor(P_1), buys(C, chips)\} \tag{4.12}$$
$$R_3 = \{client(C), livesat(C, P_1), poor(P_1), livesat(C, P_2)\} \tag{4.13}$$
$$R_4 = \{client(C), livesat(C, P_1), poor(P_1), poor(P_1)\} \tag{4.14}$$
$$R_5 = \{client(C), livesat(C, P_1), poor(P_1), rich(P_1)\} \tag{4.15}$$

Furthermore, assume that:

$$I_2 = \{\{client(C), buys(C, beer)\}\} \tag{4.16}$$

Using this information, warmr-gen will discard R_1, as there is an element in I_2 which subsumes R_1. Furthermore, R_3 and R_4 will be discarded, as they are equivalent with L_3. Note that by removing query R_3, the following query will never be generated:

$$client(C), livesat(C, P_1), poor(P_1), livesat(C, P_2), rich(P_2), \tag{4.17}$$

which means something like:

> a client lives at a poor place and at a rich place.

This is remarkable as the given query is not contradictory with the modes, nor is it equivalent with any smaller query. The θ-subsumption mechanism thus not only prunes equivalent queries, but also introduces an additional bias to the queries which may be found. □

4.3.4 Comparison between item sets and queries

In fact, a rule discovered by WARMR could be written as

$$\exists X_1 \ldots \exists X_n \, a_1 \wedge \ldots \wedge a_k, \tag{4.18}$$

where $X_1 \ldots X_n$ are all variables occurring in atoms $a_1 \ldots a_k$ which are different from the key variables. The frequency of the rule is the number of key variables for which this rule is true.

Now consider frequent item sets. APRIORI discovers frequent item sets for a database which, for every transaction, contains a set of items which are in the transaction. The database can be seen as a binary table which defines for each transaction whether an item is in the transaction or not:

	item 1	item 2	item 3
transaction 1	1	0	1
transaction 2	0	1	1
transaction 3	1	0	1

Frequent item sets can easily be translated into frequent queries using a predicate $b(T, I)$ which is true if transaction T contains item I. The frequent item set $\{1, 3\}$ becomes

$$b(T, 1) \wedge b(T, 3) \tag{4.19}$$

where T is the key variable. A general way to translate the frequent item set into a frequent query is shown by the following table, which only has other column labels:

T	$b(T,1)$	$b(T,2)$	$b(T,3)$
1	1	0	1
2	0	1	1
3	1	0	1

A frequent query is a disjunction of column labels; the items in the frequent item set determine which columns.

What about the other direction? Is it possible to translate a frequent query discovery problem into an ordinary item set frequency problem? To demonstrate this issue, another supermarket example will be used. Two predicates can occur in a query: $b(C,P)$ and $d(P_1,P_2)$; $b(C,P)$ is true if client C buys product P; $d(P_1,P_2)$ is true if product P_1 is a *different kind of product* than product P_2 (thus note that the d predicate is not equal to the $=$ predicate, and consequently different products can be of the same kind). An example of a query which can be found by WARMR is:

$$b(C,a) \wedge \exists X b(C,X) \wedge d(X,a), \tag{4.20}$$

which reads like:

> a client buys product a and buys at least one product which is of a different kind than product a.

To translate this problem into an association rule problem, at least a binary table has to be built in which columns correspond to the truth values of predicate b. However, additional columns are necessary. The query $\exists X b(C,X) \wedge d(X,a)$ can not be written as the conjunction of columns of predicate b, thus a new column which contains the truth values for this query is necessary. As queries can also exist for other constants than a, also columns are needed for other constants. If n is the number of constants, than at least $\binom{n}{1}$ additional columns are necessary.

However, these columns are not enough. Consider the following query which might be frequent:

$$\exists X b(C,X) \wedge d(X,a) \wedge \exists Y b(C,Y) \wedge d(Y,b) \wedge d(X,Y). \tag{4.21}$$

This query can not be written as the conjunction of previously discussed columns because of the last atom which defines a link between columns. New columns are necessary for this kind of queries. As a and b might be any constants, the number of columns which has to be added is $\binom{n}{2}$.

In the same way, one can consider queries like

$$\exists X b(C,X) \wedge d(X,a) \wedge \exists Y b(C,Y) \wedge d(Y,b) \wedge \exists Z b(C,Z) \wedge d(Z,c) \wedge d(X,Y) \wedge d(Y,Z) \wedge d(X,Z), \tag{4.22}$$

for which $\binom{n}{3}$ new columns will be necessary. Reasoning that also queries exist with the same number of $\exists$s as constants, one can compute that $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ additional columns are necessary at least[3]. For a reasonable number of items

[3]More columns will be necessary as queries like $\exists X b(C,X) \wedge d(X,a) \wedge d(X,b)$ have not been considered.

in the supermarket (e.g. 100), the number of columns should already be larger than 2^{100}, which is more than any computer on earth could possibly store, not to mention the fact that it would take too long to build such a table.

There are a lot of other situations which can not be transformed into an ordinary association rule problem. One other problem setting will be discussed. Consider a situation where products have properties which are expressed using a predicate $p(X, Y)$. An example of an interesting rule is:

$$\exists X b(C, X) \wedge p(X, f) \wedge p(X, v) \wedge b(C, Y) \wedge X \neq Y \wedge p(Y, d) \wedge p(Y, j), \tag{4.23}$$

which could read like:

> a client buys a product having properties $f$ood and $v$egeterian, and buys another product having properties $d$rink and pure $j$uice.

The frequent queries thus are a conjunction of such atoms:

$$\exists X b(C, X) \wedge p(X, p_1) \wedge \ldots \wedge p(X, p_k), \tag{4.24}$$

where X is different from previous products and $\{p_1, \ldots, p_k\}$ are some properties. Assume there is a large number of properties, for example 100, then there are 2^{100} combinations of properties. For all of these combinations, a new column is necessary, which contains the truth value of the corresponding conjunction for each client.

Thus these example are intractable using association rules. Could WARMR succeed on these example? In the worst case it cannot, as the amount of possibly frequent rules is too large. However, in practical cases WARMR will do a better job. Most queries are far too specific to succeed and will not need specialization, which cuts down the number of queries which have to be computed drastically.

4.3.5 An application of WARMR: taxonomies

In [Dehaspe, 1997] several applications are given of of WARMR. One of these applications is taxonomies. In this section the proposed mechanism is summarized.

It is assumed that the database contains a set of $buys(C, P)$ facts. Every fact corresponds to a client buying an item. The taxonomy is encoded using a set of is_a facts. Examples of facts for both predicates are:

$$buys(1, Nike). \tag{4.25}$$

$$is\_a(shoe, Nike). \tag{4.26}$$

The bias which is then used to utilize the taxonomy, is:

$$\begin{aligned} \{ \quad & buys(+C, -P), is(+P, a_1), \ldots, is(+P, a_n), \\ & is\_a(+P, t_1), \ldots, is\_a(+P, t_m)\}. \end{aligned} \tag{4.27}$$

This bias allows to check the products which are bought by a client, but also to check the types of the products. All rules which are generated by a specialized taxonomy algorithm will thus be generated by WARMR too (and a lot more).

4.3.6 Comparison between CLAUDIEN and WARMR

The search for clauses is performed top-down just like in CLAUDIEN, but the criterion for stopping the extension of a clause is very different. CLAUDIEN specializes a clause if it is contradictory with too many examples, hoping it will find a clause which is valid for enough examples. Adding atoms to the body of the clause here means that the body succeeds in less observations, which in turn increases the number of observations for which the clause is true. WARMR performs the contrary: only those clauses which are frequent already are specialized, hoping that a more specified body of a clause still holds for a lot of key variables. Extension stops when the body does not succeed for enough key variables. The main difference between CLAUDIEN and WARMR therefore is that the first system searches for entire clauses which are valid in a lot of observations, while the second searches for all bodies of clauses which are valid for a lot of key variables.

4.4 Summary

In this chapter the notion of association rules was introduced, as well as a first-order extension of that notion, WARMR. Some of the problems with which association rule researchers are confronted, are:

- the size of the item set space is very large, which makes an efficient implementation of association rule algorithms necessary;
- the number of rules can be very large, which makes an algorithm necessary to determine interesting rules.

It was shown that WARMR can find rules which can not be found by an ordinary association rule algorithm, and thus is a useful extension. It was illustrated furthermore how WARMR can be used to emulate other association rule algorithms. However, it was also found that:

- WARMR is not an efficient algorithm:
  - WARMR uses PROLOG to prove the truth of an atom, and thus needs a PROLOG interpreter or compiler;
  - WARMR uses θ-subsumption, which is NP-complete, to find redundancies between queries;
  - WARMR does not currently exploit any of the efficient implementation methods that were introduced for APRIORI.
- The θ-subsumption used by WARMR introduces a bias on the search space which may be undesirable.
- WARMR does not in any means provide a default system for discovering interesting rules, which make it a very weak emulator of special purpose algorithms, such as those for taxonomies.

In the next chapter an algorithm is introduced which tackles the first two disadvantages. For the third problem a solution is proposed.

Chapter 5

FARMER

The theoretical integration of logic into association rules is step one. In order for this formalism to become applicable to much more practical cases, it is necessary to proceed with a next step: the enhancement of the performance of the algorithm and the simplification of its results. In this chapter, the idea is explored that a more efficient algorithm may be obtained by removing the need for a PROLOG compiler or interpreter, and by using an efficient tree shaped data structure, which is based on an implementation of APRIORI by Pijls and Bioch (cf Section 4.2.3). To express this target, the algorithm was called FARMER, Faster Association Rules on MultiplE Relations. Furthermore, some ideas will be given about how the performance can even be enhanced further and how the results can be pruned.

5.1 Overview

This chapter is organized as follows. In the first section, a thorough description is given of the current FARMER algorithm. All elements, from bias to algorithm to data structures, are discussed. To establish whether a speedup was indeed obtained, and at which costs, the results of some experiments are presented in the second section. The third section contains some ideas about future work. Apart from further speed enhancements, also a notion of interesting queries is proposed there. The last section summarizes.

5.2 Algorithm description

This section describes all elements of the FARMER algorithm. The first subsection starts with the specification of the knowledge which can be provided to FARMER as input. The second subsection describes the bias declaration mechanism of FARMER. Here it is formally defined which queries may be generated according to a set of declarations, and which do not. The third subsection introduces the basic FARMER algorithm. This algorithm is a subset of the complete algorithm. Missing features include (more) elaborate equivalence mechanisms and search space reduction strategies. These additions are described in the fourth subsection.

5.2.1 Knowledge

Like WARMR, also FARMER is based on notation borrowed from logic. Tables which contain information are referenced to by using predicates. In contrast to WARMR, which also allows the ordinary PROLOG predicates with untyped parameters, FARMER requires predicates to have predefined typed parameters. The definition of predicates and modes, will rely on the types.

Definition 5.2.1 *$type(name, [c_1, \ldots, c_n])$ is a type definition and defines a named ordered set of constants $c_1, \ldots, c_n$. $\mathcal{T}$ is the collection of all types.*

Example With the type definitions $type(Product, [chips, wine, chocolate])$ and $type(Client, [1, 2, 3])$ two types are introduced: *Product* and *Client*. A *Product* can have the following values: *chips*, *wine* and *chocolate*, while a *Client* can be 1, 2 or 3. □

Another definition tells FARMER which constants are allowed as parameters of predicates.

Definition 5.2.2 *$predicate(p(T_1, \ldots, T_n))$ is a predicate definition and consists of the name of a predicate followed by a list of parameter types. $\mathcal{P}$ is the collection of all predicates.*

Example $predicate(buys(Client, Product))$ is a predicate definition. □

One special kind of logic formula is the fact.

Definition 5.2.3 *$p(c_1, \ldots, c_n)$ is a fact if $c_1 \in T_1, \ldots, c_n \in T_n$ for some predicate definition $predicate(p(T_1, \ldots, T_n))$.*

Example $buys(1, chips)$ and $buys(2, wine)$ are facts. □

Facts are the main knowledge type which can be provided to FARMER. Together with external procedures, they constitute the knowledge base of FARMER, which will be abbreviated to *KB*. More information about the incorporation of facts and external procedures will be given in the remainder of this chapter.

5.2.2 Language bias

As language bias an adapted version of the mode mechanism described in Sections 3.3.1 and 4.3.2 is used. The bias defines the space of queries which is searched.

Definition 5.2.4 *$mode(k, p(D_1, \ldots, D_n))$ is a mode declaration for a predicate p; k is a natural number. D_i (where $1 \leq i \leq n$) is either a constant $c \in T_i$, $+$ (both input), $-$ or $\setminus$ (both output). $\mathcal{M}$ is an ordered set of all mode declarations.*

Example $mode(1, buys(+, chips))$ and $mode(2, buys(+, -))$ are mode declarations. □

Definition 5.2.5 *$key(p)$ defines predicate $p(T_1, \ldots, T_n)$ to be the key. p should not occur in the knowledge base and is true for every variable in $T_1 \times \ldots \times T_n$.*

The meaning of the key is similar to the meaning of the key in WARMR and will become clear in the definition of the language bias.

Definition 5.2.6 *The language bias is an ordered set of mode declarations $\mathcal{M}$ together with one key predicate. Furthermore, it contains a function $ID : \mathcal{M} \cup key \rightarrow Integer$ which assigns to every mode declaration and to the key a unique identifier, such that for all $m_1, m_2 \in \mathcal{M}$:*

- *$ID(m_1) < ID(m_2)$ iff m_1 occurs before m_2 in the order;*
- *$ID(m_1) = ID(m_2)$ iff $m_1 = m_2$.*

The ID^{-1} function will be abbreviated by m_{id}.

Example Given the following types and predicate definitions:

> $type(Client, [1, 2, 3])$.
> $type(Product, [chips, chocolate])$.
> $predicate(buys(Client, Product))$.
> $predicate(client(Client))$.

Then this is a bias:

> key: $key(client)$
> mode declarations: $(mode(1, buys(+, chips)), mode(1, buys(+, chocolate)))$
> $ID(key(client)) = 1$
> $ID(mode(1, buys(+, chips))) = 2$
> $ID(mode(1, buys(+, chocolate))) = 3$

This bias will also be used in the following examples. □

The final result of FARMER will consist of similar queries as those delivered by WARMR. At an intermediate level, however, another representation is used, which is based on so-called pseudo atoms.

Definition 5.2.7 *A pseudo atom is an atom which has a mode identifier instead of a predicate identifier.*

As every mode identifier corresponds to a mode declaration, a pseudo atom can easily be translated into an atom. In which way this is performed exactly, will be discussed in more detail below.

The 'pseudo' prefix can also be used for other formalisms built on atoms. For example, a pseudo query is a query in which pseudo atoms are used.

Definition 5.2.8 *The language bias defines a set of pseudo queries $\mathcal{L}$ as follows:*

1. *formula $Q = id(X_1, \ldots, X_n)$ is a pseudo query, where $id = ID(p)$ and $p(T_1, \ldots, T_n)$ is the key predicate. $X_1, \ldots X_n$ are new variables of types $T_1, \ldots, T_n$;*
2. *if Q' is a pseudo query, $m = mode(k, p(D_1, \ldots, D_n))$ is a mode declaration of a predicate $p(T_1, \ldots, T_n)$ and $id = ID(m)$, then $Q' \wedge id(V_1, \ldots, V_n)$ is a query if for each V_i (where $1 \leq i \leq n$):*
   - *(a) if D_i is a constant, then $V_i = D_i$;*
   - *(b) if D_i is $-$ or $\backslash$, then $V_i = X$, where X is a variable that does not occur in Q'. X is a new variable of query Q and has type T_i;*
   - *(c) if D_i is $+$, then $V_i = X$ where X is a variable that does occur in Q' and has type T_i.*

   Furthermore, id may not occur more than $k-1$ times in the query Q'.

Example The set of pseudo queries is constructed as follows:

$$\begin{aligned}
Q_1 &= \{1(C)\} \\
Q_2 &= \{1(C) \wedge 2(C, chips), 1(C) \wedge 3(C, chocolate)\} \\
Q_3 &= \{1(C) \wedge 2(C, chips) \wedge 3(C, chocolate), \\
&\quad\ 1(C) \wedge 3(C, chocolate) \wedge 2(C, chips)\}
\end{aligned}$$

Note that these are all queries which may be found for the bias of the previous example. □

Note that Q_3 of the example contains two formulas which are introduction order permutations of each other (see Definition 2.4.9). It will be the task of FARMER to avoid the creation of one of these queries.

Every pseudo atom refers to a mode declaration which specifies input and output parameters. Both information about the mode declaration and the atom can be combined into one notation by prefixing the parameters of the atom with the mode constraints and by using the predicate identifier in stead of the mode identifier.

Example The following are some pseudo queries in prefix notation:

$$\begin{aligned}
Q_1 &= \{client(-C)\} \\
Q_2 &= \{client(-C) \wedge buys(+C, chips) \wedge buys(+C, chocolate) \\
Q_3 &= \{client(-C) \wedge buys(+C, chips) \wedge buys(+C, chocolate), \\
&\quad\ client(-C) \wedge buys(+C, chocolate) \wedge buys(+C, chips)\}
\end{aligned}$$

□

This prefix notation will frequently be used in the sequel.

Every pseudo query can be translated into an ordinary query. What is meant by the translation of a query, is explained formally by the next definition.

Definition 5.2.9 *The translation of a pseudo query, denoted with $tr(Q)$ is determined using the function trv, as follows: $tr(Q) = trv(Q, \emptyset)$, where trv is defined as follows:*

- *if $Q = \emptyset$, $trv(Q, S) = \emptyset$;*
- *if $Q = id(V_1, \ldots, V_n) \wedge Q'$, then $trv(Q, S) = p(V_1, \ldots, V_n) \wedge N(V_1, S_1) \wedge \ldots \wedge N(V_n, S_n) \wedge trv(Q', S_{n+1})$. S_1 equals S. Depending on the mode constraint of V_i, $N(V_i, S_i)$ is:*
  - *if $D_i = \backslash$, then $N(V_i, S_i) = \wedge_{S \in S_t} S \neq V_i$, where $S_t = \{S \in S_i | S$ is of the same type as $V_i\}$. S_{i+1} is set to $S_i \cup V_i$;*
  - *$N(V_i, S_i) = \emptyset$ and $S_{i+1} = S_i$, otherwise.*

This definition states that a query can be obtained from a pseudo query by replacing mode identifiers with predicate identifiers, except when the mode constraint is a backslash. In that case a disjunction is added which forces a variable assignment to a different constant than is previously used. The queries which are obtained using the *tr* function, correspond to the queries which are found by WARMR.

Another way of looking at pseudo queries is that the pseudo atoms define some sort of stream relation. The mode is used as a unit which translates input values (denoted by +) into output values (denoted by a − or a \). The *mode* declarations guide the addition of units to the system.

Example Add the following predicates and language bias:

$$predicate(likes(Client, Product))$$
$$mode(1, buys(+, -))$$
$$mode(1, likes(+, +))$$

Then the following is one of the queries defined by this bias:

$$client(C) \wedge buys(C, chips) \wedge buys(C, P) \wedge likes(C, P).$$

The stream relation which is expressed in this query is depicted in Figure 5.1.

□

5.2.3 Basic FARMER

In this section a simplified algorithm will be described to find a set of frequent queries. It is called the basic FARMER algorithm. Its results contain many logically equivalent queries. The goal of the next section will be to improve

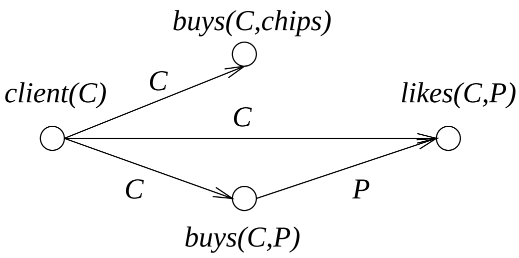


Figure 5.1: A graph which depicts the stream relation of the query given in the last example of Section 5.2.2.

the basic FARMER algorithm in such a way that some equivalent queries will be removed by the algorithm.

FARMER uses the same generate and test mechanism which is also used by APRIORI and WARMR. Whereas for itemsets it is clear when they are a subset of a transaction, this is not obvious for pseudo queries. Before giving the algorithm, it is therefore necessary to define properly what is meant with the support of a (pseudo) query.

The evaluation mechanism for pseudo queries is based on the notion that a pseudo atom can be regarded as a unit which translates a set of inputs into sets of outputs. The mode declaration exactly defines which parameters can be used as inputs and which only serve as outputs. For every mode declaration, a procedure in any programming language could therefore be used, although, of course, a more traditional PROLOG approach is also possible. FARMER has built-in procedures to compute a set of outputs using a set of facts, and offers an interface to link modes to external procedures. For both, there are two kinds of procedures: one which can be used for modes which do not have outputs and another for modes which have output variables.

Definition 5.2.10 *A boolean procedure is a function $\beta : T_{c_1} \times \ldots \times T_{c_n} \rightarrow \{true, false\}$, where $T_{c_i} \in \mathcal{T}$; the number n is called the input dimension.*
An output procedure is a function $\beta : T_{c_1} \times \ldots \times T_{c_n} \rightarrow 2^{T_{k_1} \times \ldots \times T_{k_m}}$, where $T_{c_i}, T_{k_i} \in \mathcal{T}$. The number n is called the input dimension and m is the output dimension.

Every mode declaration must be linked to a procedure. If no procedure is specified, a default procedure is used, which consults the built-in knowledge base of facts.

Definition 5.2.11 *Given is a mode $mode(k, p(D_1, \ldots, D_n)) \in \mathcal{M}$. Let id be its identifier and let $predicate(p(T_1, \ldots, T_n)) \in \mathcal{P}$. If the mode has no outputs, then the default boolean procedure is $\beta_{id} : T_1 \times \ldots \times T_n \rightarrow \{true, false\}$, which is defined as:*

$$\beta_{id}(V_1, \ldots, V_n) = \begin{cases} true & \text{if } p(V_1, \ldots, V_n) \in KB \\ false & \text{otherwise} \end{cases} \tag{5.1}$$

If the mode has outputs, then the default output procedure is $\beta_{id} : T_{c_1} \times \ldots \times T_{c_m} \rightarrow 2^{T_{k_1} \times \ldots \times T_{k_{n-m}}}$. *Here the* T_{c_i} *are in order the input parameters, and the* T_{k_i} *are in order the output parameters.*

$$\beta_{id}(V_{c_1}, \ldots, V_{c_m}) = \{(V_{k_1}, \ldots, V_{k_{n-m}}) \mid p(V_1, \ldots, V_n) \in KB\} \tag{5.2}$$

Procedures are used during the evaluation of queries. The link between pseudo atoms and procedures is defined by two functions, the atom interpretation function and the atom variable assignment function.

Definition 5.2.12 *Given is a pseudo atom* A *and a set of variable assignments* $B = \{X_1/c_1, \ldots, X_k/c_k\}$. *Let* X *be an ordered set, such that the* i*th element is constant* c *if the* i*th input of* A *is:*

- *constant* c*;*
- *a variable* X_i *which is bound to* c *by* B.

The atom interpretation function I *is defined as follows:*

$$I(A, B) = \begin{cases} \beta_A(X) & \text{if } F = \emptyset \\ \text{true} & \text{if } F \neq \emptyset \wedge \beta_A(X) \neq \emptyset \\ \text{false} & \text{otherwise} \end{cases} \tag{5.3}$$

The atom variable assignment function $\mathcal{B}$ *is defined as follows:*

$$\mathcal{B}(A, B) = \begin{cases} \{\{X_{k_1}/c_1, \ldots, X_{k_n}/c_n\} | (c_1, \ldots, c_n) \in \beta_A(X)\} & \text{if } F \neq \emptyset \\ \{\emptyset\} & \text{otherwise} \end{cases} \tag{5.4}$$

Here is X_{k_i} *the* i*th output variable of pseudo atom* A.

These functions are used by FARMER to evaluate the truth of queries. The next definitions state which output can be expected from the basic FARMER algorithm. Thereafter, the algorithm which creates the results is given.

Definition 5.2.13 *The interpretation function* I *assigns a truth value to a pseudo query* Q *given a set of variable assignments* $B = \{X_1/c_1, \ldots, X_k/c_k\}$, *and is defined as follows:*

1. *if* Q *is empty* $I(Q, B) = $ *true;*
2. *otherwise,* $Q = A \wedge Q'$. *Then* $I(Q, B)$ *is true iff:*
   (a) $I(A, B)$ *is true;*
   (b) *there is a* $B' \in \mathcal{B}(A, B)$ *such that:*
      i. $I(Q', B \cup B')$ *is true;*
      ii. *no variable which belongs to a* $\backslash$*-output is assigned a constant which is already used in* B.

The procedure-based interpretation function has a strong restriction: every input variable of a pseudo atom should be assigned a value, otherwise it can not be evaluated. The purpose of the key is to assign all free variables a value.

Definition 5.2.14 *Let $Q = key(X_1, \ldots, X_n) \wedge Q'$ be a query, then the support of Q, denoted by $support(Q)$, is defined as:*
$support(Q) =$
$|\{B \mid B = \{X_1/c_1, \ldots, X_n/c_n\}, c_i \in T_i (1 \leq i \leq n), I(Q', B) = true\}|$

Definition 5.2.15 *The output of the basic* FARMER *algorithm consists of a set of pseudo queries $\mathcal{Q}$, such that:*

1. *no pseudo queries in $\mathcal{Q}$ are introduction order permutations (see Definition 2.4.9);*
2. *for every pseudo query $Q \in \mathcal{L}$ for which $support(Q) \geq minsup$, there is a query $Q' \in \mathcal{Q}$ such that $Q' \simeq Q$.*

Queries which have a support that exceeds the threshold value *minsup* are also called *frequent (pseudo) queries.*

To find the set of pseudo queries, the basic FARMER algorithm creates a tree which contains all frequent pseudo queries as a path from the root to a leaf.

Example Add the following declarations:

$predicate(diff\_kind(Product, Product))$
$mode(2, buys(+, -))$
$mode(1, diff\_kind(+, +))$

Then Figure 5.2 shows such a tree. It can easily be seen that every path corresponds to a query which is defined by the bias. Note that none of the queries are permutations of each other. □

The search for frequent queries is started with a query containing only the key predicate. After that, a process of creating and checking queries is repeated. A new query is created by adding a child atom to an existing query. After all possible queries are expanded with one atom, a pass through the knowledge base is made to count the support of the new queries. The process thus consists of two phases which are similar to those of APRIORI and WARMR: a *candidate generation* phase and *support counting* phase. Each which will be discussed separately in more detail below.

Candidate generation

Consider a leaf in the tree of queries. The path from the root to this leaf represents some query which can possibly be specialized into a more specific query. A specialization is obtained by adding an atom. There are three types of specializations:

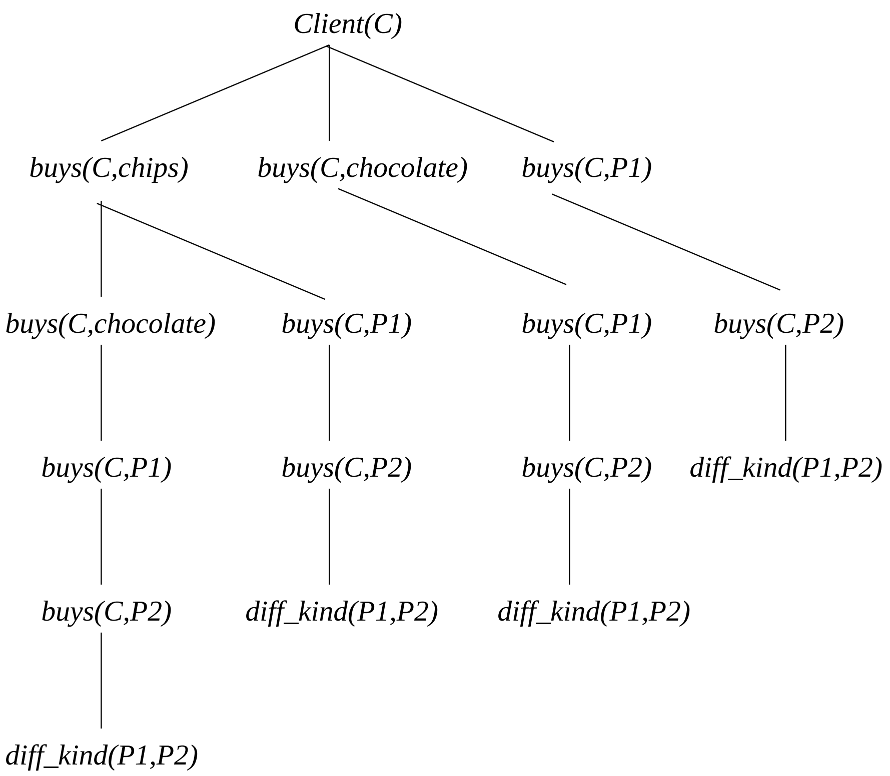


Figure 5.2: A tree which contains queries for the language bias of Section 5.2.3.

- Add a right brother of a leaf as a child of that leaf. Repetitive application of this rule results in the generation of all subsets of a set of brothers. No new atoms are introduced in fact, only existing atoms are copied. Making such copies does not violate the mode definition restrictions, as the copied atoms depend on atoms which are also parents of the leaf node to which they are copied. It is this kind of copying nodes that is also performed by Pijls' and Bioch's implementation of APRIORI, see [Pijls, 1999].

- If the leaf has some output variables, all atoms should be added which use at least one of these variables. The order of the child atoms is defined by the order of the mode declarations. This specialization allows for the introduction of new predicates to the tree.

- A special case is a leaf node which has some output variables and a corresponding mode declaration which allows the addition of the same atom with new output variables. For such a leaf, a copy of the leaf itself with new output variables has to be added as a child.

After the addition of atoms to a query, new queries result which might not be good candidates. If some smaller and more general subset query is not frequent, this larger and more specific query can not be frequent too. To incorporate this thought, WARMR and APRIORI perform a *prune*-step. In APRIORI this operation is not complex as it only requires some subset tests. In WARMR, on the other hand, this operation is very time consuming, as it requires a test for θ-subsumption with every infrequent query. Although with some basic atom counting strategies a lot of tests can be avoided, there still remain a lot of tests to perform. Basic FARMER uses another strategy and does not perform any pruning step. The assumption is that the overhead in counting the extra candidates is less than that of a pruning algorithm[1].

Support counting

To count the support of the queries in the tree, for every combination of key variables a traversal of the tree has to be performed. Figure 5.4 shows the counting algorithm.

For the root node, which is the key, all assignments to the output variables are performed in turn. The recursive count procedure is called for all the children of the root. In this way, every 'example' is considered in turn and the set of true queries for that example is determined.

[1] Instead, also a less complex algorithm than θ-subsumption could be used to prune. For example, from every new candidate query, each atom which does not introduce new variables can be removed in turn, resulting in a set of queries which are one atom smaller. In the tree it is verified whether these queries are frequent; only if all these subqueries are frequent, the new query is maintained. The removal of atoms introducing new variables is not allowed as other atoms in the query might depend on this atom. The remaining, smaller queries still have the same number of variables. It would also be possible to allow the removal of atoms which have unused output variables.

```
Expand candidates(node,variables)
Let M be the mode declaration corresponding to node
M.counter--
if node is internal then
  add output variables of node to variables with correct type
  for all children do
     expand candidates(child,variables)
else if node.support ≥ minsup then
  for all dependent mode declarations M' of M which
        have a counter > 0
  do
     for all bindings of inputs to existing variables
           of the correct type, such that at least one
           input is bound to an output variable of node
     do
        add child for M' with new output variables.
  add as child all brothers at the right of node which
  have support ≥ minsup and corresponding mode
  declaration M'.counter > 0.
  if M.counter > 0 and M has outputs then
     add child to node of mode declaration M, with exactly
     the same input variables but new output variables
else
  remove node
M.counter++.
```

Figure 5.3: The algorithm for expanding the tree.

The recursive count procedure is called for a node and a set of variable assignments. It first checks whether the pseudo atom in the node can be satisfied. If this is not the case, nothing else can be done. If the node can be satisfied, the support is increased if the node is a leaf and thus is the last atom of the query. Otherwise, the children will be evaluated. The target is to find for each child a variable assignment that satisfies the child. This search is performed by trying all variable assignments for the outputs of the node in turn. Note that the set of variable assignments might be empty when there are no outputs. Furthermore, the order in which the assignments are tried will influence the speed of the algorithm. A special strategy might therefore be useful. A boolean variable is used to remember whether any child is left which has not already been satisfied by some assignment. When all children have been satisfied, the passing of assignments and children can stop and the procedure returns.

Within this procedure, a further similary with Pijls' and Bioch's implementation of APRIORI can be found: the candidate tree is passed for each example, and the the validaty of each candidate is tested for the candidate. For atoms the reverse (checking for the existence of candidates using the example) would be much more difficult.

5.2.4 Data structures

To make the internal representation simpler and consequently more efficient, the algorithm does not use any names. Every predicate, type, constant or mode identifier is translated into an integer number, where constant numberings restart for each type.

Example In the remainder of this section, an extended type declaration for products is used:

$$type(Product, [chips, chocolate, cola, 7up]).$$

The numbers which are given, are:

name	number
buys	1
diff_kind	2
chips	1
chocolate	2
cola	3
7up	4
1–3	1–3

□

The type information about predicates and constants is translated into two tables: one table containing for each type the number of constants, and another table containing for each predicate the type numbers of the parameters.

```
Count candidate(node,B)
if I(node, B) = true then
   if node is a leaf then
      disable node
      node.supportcount++
      return true
   else
      for all B′ ∈ 𝓑(node, B) do
         d:=true
         for all children do
            if child not disabled then
               d := Count candidate(child,B ∪ B′) ∧d
         if d = true then
            disable node
            return true
return false

Count tree
bc := 1
for all bindings B of key variables to constants do
   for all children of the root do
      Count candidate(child,B)
   enable all nodes
   bc++
```

Figure 5.4: The procedure for updating the support values of a tree of queries.

Example The tables for the predicate and type declarations are:

type number	number of constants	comment
1	3	*Client*
2	4	*Product*

predicate number	parameters	comment
1	1,2	*buys*
2	2,2	*diff_kind*

□

The mode declarations are transformed into another table, which contains for each mode declaration the number of times that it may be included in a query, the number of the predicate and the parameters. Each parameter consists of a $+$, $-$, $\backslash$ or c, followed by a number. The $+$, $-$ and $\backslash$ symbols are followed by a type number, which is obtained by a lookup in the predicate type table. Symbol c is followed by the number of a constant. With every mode declaration, some additional data structures are created which allow fast traversals through output positions, inputs, and so on.

Example The table of mode declarations is:

mode number	id	predicate	parameters	comment
1	1	1	+1,c1	$mode(1, buys(+, chips))$
2	1	1	+1,c2	$mode(1, buys(+, chocolate))$
3	1	1	+1,c3	$mode(1, buys(+, cola))$
4	1	1	+1,c4	$mode(1, buys(+, 7up))$
5	2	1	+1,\2	$mode(2, buys(+, \backslash))$
6	1	2	+2,+2	$mode(1, diff\_kind(+, +))$

□

As no procedure is specified for any mode, default procedures will be used. Before starting the search, for every mode procedure a datastructure is built which allows quick execution of the procedure. For example, the mode declaration $mode(1, buys(+, \backslash))$ allows for the introduction of a node which, given some client, should find a product which is bought. For this mode declaration a table is built which contains for each client the products he buys.

Example The following knowledge base is used:

$buys(1, chips).$
$buys(1, chocolate).$
$buys(2, chips).$
$buys(2, cola).$
$buys(3, chocolate).$
$buys(3, cola).$

$buys(3, 7up)$.
$diff\_kind(chips, cola)$.
$diff\_kind(chips, 7up)$.
$diff\_kind(chocolate, cola)$.
$diff\_kind(chocolate, 7up)$.
$diff\_kind(cola, chips)$.
$diff\_kind(7up, chips)$.
$diff\_kind(cola, chocolate)$.
$diff\_kind(7up, chocolate)$.

Then the table which is created, becomes:

client	products
1	1, 2
2	1, 3
3	2, 3, 4

□

The products are stored in a list, thus allowing a fast discovery of the products bought.

For a boolean procedure, such as the procedure for the mode declaration $mode(1, buys(+, chips))$, this would not be an efficient storage structure. A traversal of the list in search for *chips* would be necessary. Therefore, a different datastructure is used for mode declarations which have no outputs: a multidimensional binary matrix.

Example The matrix for predicate *buys* becomes:

1	1	2	3	4
1	1	1	0	0
2	1	0	1	0
3	0	1	1	1

For *diff_kind* the matrix is:

2	1	2	3	4
1	0	0	1	1
2	0	0	1	1
3	1	1	0	0
4	1	1	0	0

□

The tree of queries is represented using mode declaration IDs and numbers of constants and variables. Using the mode table, all additional information about a node can be obtained, including the predicate number, the types of the parameters, whether they are inputs or outputs, and so on. New in the tree are variables. A variable is represented by a combination of a type number and a

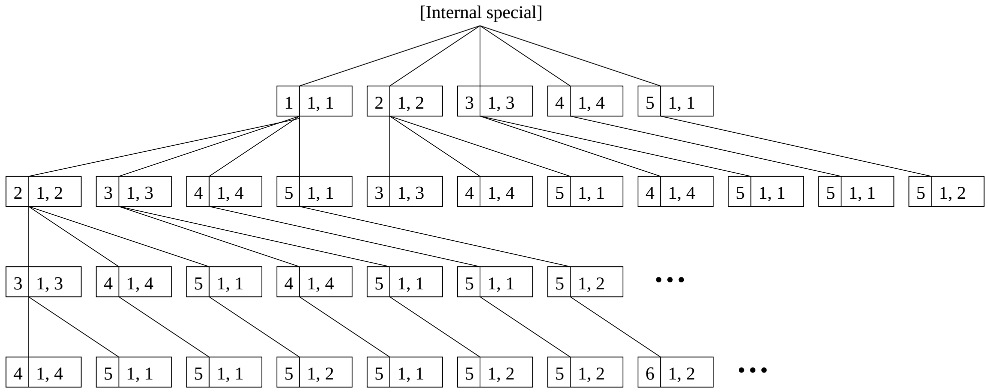


Figure 5.5: A part of the tree which is constructed in Section 5.2.4 . To understand the meaning of a node, a lookup in the mode table is necessary. Support and enable fields are not shown.

variable number. Only the variable number is stored in the tree, as the type number can be found in the mode table. Every path from the root to a node is a query. For all these queries, the support is computed and stored in the node. Additional to the support field, a counter field is used to store whether a node is disabled or not. If this counter field has a lower value than the bc counter used in Figure 5.4, then the node is enabled. A node is disabled by setting the counter to bc. All nodes are enabled by incrementing bc, thus avoiding a traversal of the tree to reset counters. The counters should be reinitialized when expanding the tree.

Example A part of such a tree is shown in Figure 5.5. □

During the expansion of the tree some special data structures are used. First of all, a set of *variables* is used which contains all variables in the query until this node, together with the correct type. As a variable is identified by its number, it is sufficient to represent this set as an array which contains for each type the number of variables. A new variable of a certain type is created by incrementing the counter of the corresponding type.

Another data structure maintains for each mode declaration a set of dependent mode declarations. A dependent mode declaration has at least one input variable which can be bound to an output variable of the current mode. This table can easily be constructed from the mode declaration table. It is advantageous to construct this table before computing the expansions.

After a tree is created, the supports of the queries in it are counted. In this counting phase, a set is used which contains assignments of variables to constants. These bindings are stored in a double array. The jth cell of the ith row contains the value of the jth variable of type i. An additional binary matrix is used for the \-output feature, as it should be possible to check quickly whether a constant has already been used or not.

5.2.5 Farmer

The output of basic Farmer contains a lot of logically equivalent queries. This has two major drawbacks. First of all, it means that the same query might be counted and expanded various times. Probably redundant computations are performed in that case. Secondly, it is difficult to discover the interesting queries in data if some query is repeated exhaustively. Therefore as much equivalent queries as possible should be removed during the search. Warmr uses θ-subsumption for this purpose, which is a time consuming operation. Farmer filters in a more ad-hoc fashion. In the next sections, some of the equivalences which are most likely to occur are given and their incorporation in Farmer is discussed.

After the equivalence mechanisms, some additional features of Farmer are discussed. The first is the mechanism of mode numbers. This is an additional bias specification mechanism which can be used to speed up the search or to avoid some undesirable queries. The second feature is a limited mechanism for finding interesting queries, which has as sole purpose the reduction of the number of output queries.

Exact repetition

The basic Farmer algorithm allows that a pseudo query contains several equal atoms after application of the *tr* function.

Example Consider the following mode declarations:

$$mode(1, buys(+, \backslash))$$
$$mode(1, buys(+, +))$$

Then the following is a valid pseudo query:

$$client(-C_1), buys(+C_1, \backslash P_1), buys(+C_1, +P_1)$$

The first *buys* atom is introduced using the first mode declaration. The second atom is introduced using the second mode declaration and uses the P_1 which was introduced before. □

This kind of repetition is easy to prevent. The repetitive atom will always be introduced as a child of the original one, as it uses variables introduced by that atom. A simple comparison between the child and its parent suffices to prevent this query.

Query isomorphism

The basic Farmer algorithm has no means to determine that one query is introduction order isomorphic with another. To test whether two queries are isomorphic is already a consuming operation; comparing some query with every other query is even more complex. Farmer only checks some cases of isomorphisms. Also, the θ-subsumption based operation of Warmr is not used as this operation prunes too many queries.

Before discussing the cases which are pruned by Farmer, some extra attention should be given to the mechanism which is used to assign names to variables. Every variable is represented by two numbers: a type number and a variable number. The name of a variable is uniquely defined by its position in the query. Two queries which have the same pseudo atoms can never have different variable numbers. The question whether two queries are isomorphic, thus reduces to the question if there is some permutation of atoms, followed by an according deterministic variable renumbering, which makes some query equal to another. This also means that if two different queries are isomorphic, there must at least be one atom which can be moved more to the front of the query (while remaining variable introduction order) such that after deterministic renumbering, that atom equals an atom which originally occurred at that position.

To tackle the problem of isomorphisms, it is necessary to characterize the elements of Farmer which cause the creation of such atoms which can be moved more to the front. Reconsider the three mechanisms which are used by Farmer to expand queries:

1. generate dependent nodes;
2. copy the right brothers of the parent;
3. copy the parent itself.

The first mechanism generates some nodes which could not have been added earlier; the problem can not be caused by such nodes, as these can not be moved. The second mechanism copies atoms which have already been introduced at a higher level. Such an atom may be moved to an earlier position, but if it then also equals an earlier atom, this movement could also be performed in the source query of the copied atom (as the copied atom does not use any variables of the target query), which means that the problem already occurred in the source query. The only remaining mechanism is the copying of a parent; it is also intuitively clear that the node which is introduced by doing this, can easily be switched with its parent.

We have thus characterized the queries for which there can exist introduction order isomorphic queries elsewhere in the tree. These are queries in which a copy of a parent occurs.

Furthermore, some extra attention should be given to the order of the children. There are three types of children. It can easily be seen that the order of these children does not influence the queries which are found by basic Farmer, although it does influence the order of the atoms in a query and the speed of the search process. Another possible order for the children (which is not used by FARMER) is:

1. a copy of the parent;

2. copies of the right brothers of the parent;

3. dependent nodes.

This order results in pseudo queries which have a series of outputting atoms first, followed by a list of atoms using the outputs.

Example In Figure 5.6, the trees for the two child orders are illustrated. In the bottom tree, the *buys* atom which is a copy of its parent, is copied left and down into the tree. In this way, every query left in the tree is expanded with this atom. Queries first contain the atoms which use variables introduced by the original *buys* atom. □

The order which is used by FARMER has as important consequence that each query which is order introduction isomorphic to another query more to the left in the tree, can safely be removed without cutting some queries from the search space. In order to prove this statement, consider two isomorphic queries q_1 and q_2 which have the following configurations:

$$
\begin{aligned}
q_1 &= Q \wedge p(\backslash V_1) \wedge Q_a(+V_1) \wedge Q_b(+V_1) \wedge p(\backslash V_2) \wedge Q_a(+V_2) \wedge Q_c(+V_2) \\
q_2 &= Q \wedge p(\backslash V_1) \wedge Q_a(+V_1) \wedge Q_c(+V_1) \wedge p(\backslash V_2) \wedge Q_a(+V_2) \wedge Q_b(+V_2)
\end{aligned}
$$

The capitals Q denote conjunctions of atoms, $Q(+V)$ is a conjunction in which the input positions are bound to variables in V. An atom $p(-V)$ denotes the application of a predicate p which introduces a set of variables V. When determining the isomorphism, V_1 and V_2 should clearly be mapped to each other. Assume that the first atom in Q_c (if any) is higher than Q_b in the order of modes. As a result, q_2 appears more to the right in the tree than q_1. The question is whether any query which is an expansion of q_2 also exists more to the left in the tree. The expansions can be divided into two kinds:

- new atoms using only variables from V_2;

- new atoms using variables from both V_1 and V_2.

Each one has to be considered in turn. First, q_2 is expanded with an atom $a(+V_2)$. An equivalent query is a query which applies a to V_1:

$$
\begin{aligned}
q_1' &= Q & \wedge p(\backslash V_1) \wedge Q_a(+V_1) \wedge Q_b(+V_1) \wedge a(+V_1) \\
&& \wedge p(\backslash V_2) \wedge Q_a(+V_2) \wedge Q_c(+V_2) \\
q_2' &= Q & \wedge p(\backslash V_1) \wedge Q_a(+V_1) \wedge Q_c(+V_1) \\
&& \wedge p(\backslash V_2) \wedge Q_a(+V_2) \wedge Q_b(+V_2) \wedge a(+V_2)
\end{aligned}
$$

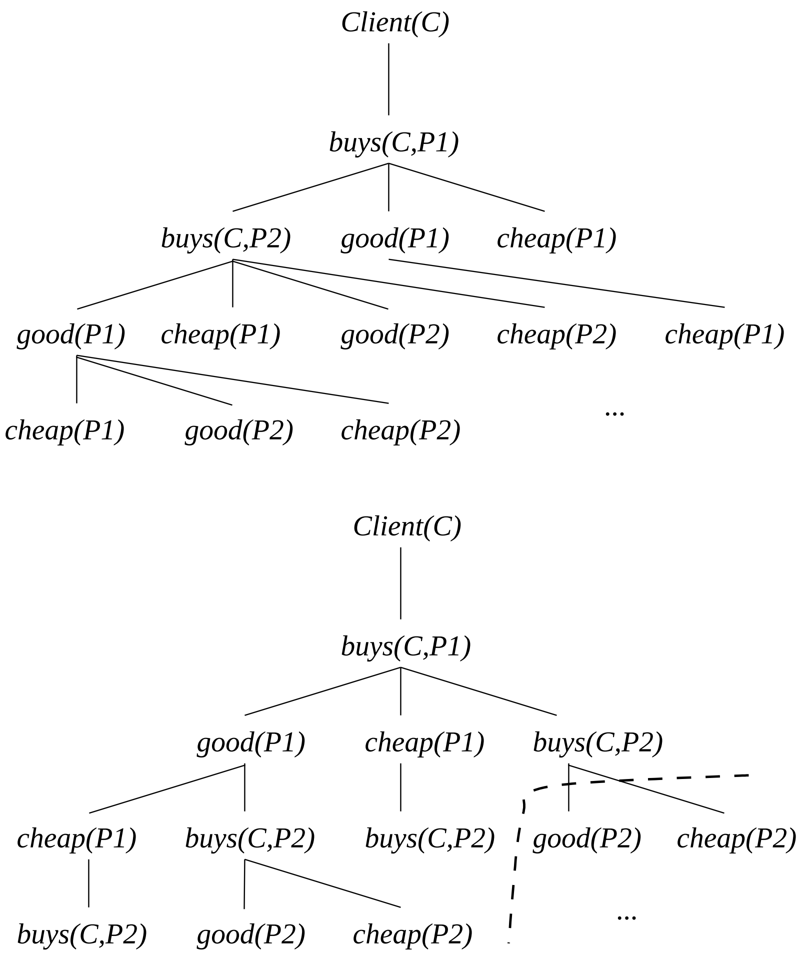


Figure 5.6: The same set of queries generated using different orders of child nodes. In the top tree, the copy of the parent is the first child, in the bottom tree it is the last child. In that tree, the nodes at the right of the dotted line do not need to be created as they contain queries which are isomorphic with queries more to the left in the tree.

This first query also exists in the tree, as $a(+V_1)$ is an atom which does not depend on V_2 and is also generated after query $Q \wedge p(\backslash V_1)$.

In the second case, an atom $a(+V_1, +V_2)$ is added which uses variables from V_1 and V_2:

$$\begin{array}{lcllr} q_1'' & = & Q & \wedge p(\backslash V_1) \wedge Q_a(+V_1) \wedge Q_b(+V_1) & \\ & & & \wedge p(\backslash V_2) \wedge Q_a(+V_2) \wedge Q_c(+V_2) & \wedge a(+V_1, +V_2) \\ q_2'' & = & Q & \wedge p(\backslash V_1) \wedge Q_a(+V_1) \wedge Q_c(+V_1) & \\ & & & \wedge p(\backslash V_2) \wedge Q_a(+V_2) \wedge Q_b(+V_2) & \wedge a(+V_1, +V_2) \end{array}$$

As a uses variables from V_2 too, it can not be introduced after the first application of p. However, it is clear that the equivalent query q_1'' also exists in the tree and appears more to the left in the tree.

We have thus shown that all queries may be removed for which there exist isomorphic queries in the tree, unless it is the leftmost query in the tree. It is however still hard to determine this for a query. In the sequel of this section, a mechanism is discussed which in some cases succeeds in determining whether the new query could occur more to the left in the tree and is thus redundant.

The mechanism works as follows. Assume that we are expanding a node $node_2$, and that it is known that, after moving one or more literals, this node is equal to some other node $node_1$ higher in the tree and thus more the left in the query. Then for each expansion of node $node_2$, it must be checked whether this expansion node can also be moved to an earlier position. As this expansion node occurs directly after its parent, an isomorphic node of this node should also occur directly after $node_1$. Because we want to know whether an isomorphic query exists more to the left in the tree, it suffices to determine whether the node $node_1$ has an isomorphic child at the left of the path which leads to $node_2$.

To check the isomorphism, it would be necessary to know the exact permutation of all nodes which led $node_2$ to be isomorphic to $node_1$. Because it is considered to be too space consuming in most cases, FARMER does not save this information. Instead, it tries to reconstruct the resulting variable renumbering by comparing the variables which occur in $node_2$ and $node_1$. As not all variables occur in these nodes (remember that $node_2$ might have a parent which had to be moved *too*), this is certainly not a complete reconstruction, but at least some children of $node_1$ can appear to be equal to children of $node_2$ after this substitution.

To allow this mechanism, two fields are added to the nodes:

- a field *introdepth* which remembers the depth at which the node was introduced;

- a field *equivalent* which remembers the depth at which an equivalent node occurs.

The first field is always initialized to the depth at which the node was created for the first time. It is used to restrict the search only to the dependent nodes of $node_1$.

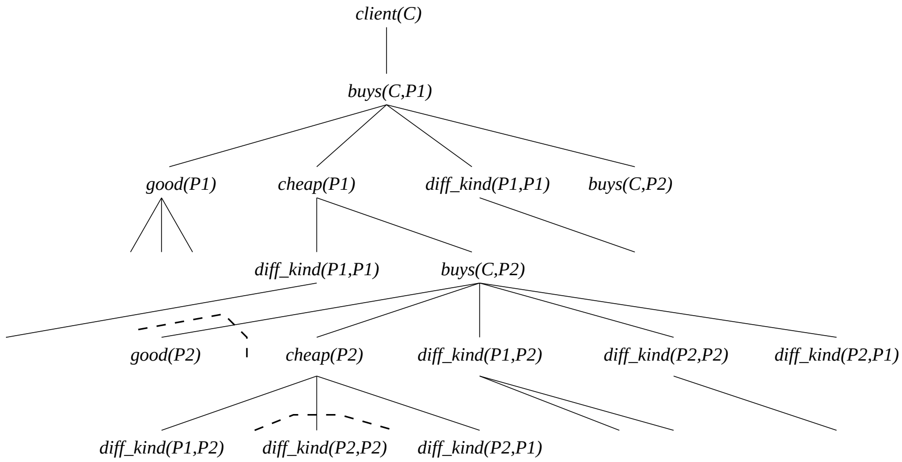


Figure 5.7: Part of the search tree for the example in Section 5.2.5. Those parts of the tree which appear behind a dotted line, need not be generated.

The second field is only set in those situations where an equivalent node exists. Initially, it is only set when a node is a copy of its parent, in which case it is initialized to the depth of its parent. It is here that the conclusion comes back that the isomorphism problem is caused by the duplication mechanism.

In Figure 5.8, the algorithm is shown which reflects the ideas discussed and removes some of the redundant children.

Example Figure 5.7 shows a part of a tree which contains nodes that can be discarded. The node $good(+P2)$ is not generated, as the equivalent node $good(P1)$ is more to the left than the node $cheap(+P1)$, which is part of the query which ends in the leaf $good(+P2)$. The same holds for the node $diff\_kind(+P2, +P2)$. □

A major shortcoming (among others) of this mechanism is that it only removes redundant dependent nodes. Copied nodes can very well be redundant too. There may be some possibilities to remove these too, for example by considering the variables at the introduction depth level and by considering the equality of *introductiondepth* fields with the children of $node_1$, but these ideas have not been explored further.

Minus output

There are two output modes, a backslash output and a minus output. The minus output (which corresponds to the default PROLOG way of assigning) causes many queries to be equivalent:

Example These queries are equivalent:

CheckEquivalence $(node_2, S_2, history)$
if $node_2.equivalent$ is defined **then**
  Let d be $node_2.equivalent$
  Let $node_1$ be $history[d]$ and let S be the set of its children.
  Let $node_1'$ be $history[d+1] \in S$
  $S_1 := \{s \in S, s \leq node_1' \mid s.introdepth = d+1\}$
  $\theta := \emptyset$
  for all positions i in $node_1$ and $node_2$
      which contain variable number
  do
    $\theta := \theta \cup \{$ variable at i in $node_1$/variable at i in $node_2\}$
  $i := 1$
  for all $s_1 \in S_1$ **do**
    while $S_2[i].mode \neq s_1.mode$ **do**
      $i := i+1$
    $j := i$
    while $S_2[j].mode = s_1.mode$ **and** $S_2[j]\theta \neq s_1$ **do**
      $j := j+1$
    if $s_1 = node_1$ **then**
      $S_2[j].equivalent = d+1$
    else
      Discard node $S_2[j]$

Figure 5.8: An algorithm which for a given node and its ordered set of created dependent nodes, removes some nodes of that set which are equivalent to queries more to the left in the tree. The parameter *history* should be an array which, in order, contains the nodes which led to the leaf $node_2$. This leaf should just have obtained new children of which the equivalency has to be checked.

$$\begin{array}{lcll}
q_1 & = & client(-C) & \wedge buys(+C,-P1) \wedge good(+P1) \wedge cheap(+P1) \\
q_2 & = & client(-C) & \wedge buys(+C,-P1) \wedge good(+P1) \wedge cheap(+P1) \\
 & & & \wedge buys(+C,-P2) \wedge good(+P2) \\
q_3 & = & client(-C) & \wedge buys(+C,-P1) \wedge good(+P1) \wedge cheap(+P1) \\
 & & & \wedge buys(+C,-P2) \wedge good(+P2) \wedge buys(C,P3) \\
q_4 & = & client(-C) & \wedge buys(+C,-P1) \wedge good(+P1) \wedge cheap(+P1) \\
 & & & \wedge buys(+C,-P2) \wedge good(+P2) \wedge cheap(P2)
\end{array}$$

□

Without using knowledge about the expansions still possible to come, none of these queries can be removed. For example consider query q_4 which is clearly too long now. If there is a mode which takes two products as inputs, this mode can expand q_4 into a new query which is not equivalent with any smaller query.

The removal of such too long queries however still is desirable when presenting the results of a search to the user. An algorithm which analyses rules to this purpose can be used as a postprocessing step.

Mode numbers: an additional restriction mechanism

Although the mode mechanism already restricts the range of clauses which is explored, it is still unavoidable that some queries are tested which at beforehand are known to be useless.

Example Consider this pseudo-query:

$client(-C), buys(+C,-P)$

and assume that there are these two modes:

$mode(1, good(+)).$
$mode(1, bad(+)).$

According to this bias, also the following query is created and tested:

$client(-C) \wedge buys(+C,-P) \wedge good(+P) \wedge bad(+P).$

This would be a waste of time if we know at beforehand that *good* and *bad* can not be true for the same product. □

The mechanism which can be used in FARMER to avoid such queries from even being generated, is based on *mode numbers*. Every mode can be given a number (certainly not unique); FARMER will never add a child which has the same mode number as its parent.

Example The following modes can be used:

$mode(1, good(+), 1).$
$mode(1, bad(+), 1).$

as a result, this query can not be generated:

$$client(-C) \wedge buys(+C, -P) \wedge good(+P) \wedge bad(+P)$$

because the mode number of $good(+P)$ is 1, which is the same as the mode number of $bad(+P)$. □

The numbering mechanism can also be used to avoid other undesirable queries.

Example Consider the following query:

$$\begin{aligned} client(-C) \quad &\wedge \quad buys(+C, \backslash P_1) \wedge price(+P_1, -R_1) \wedge low(+R_1) \\ &\wedge \quad buys(+C, \backslash P_2) \wedge price(+P_2, -R_2) \wedge high(+R_2). \end{aligned}$$

There can be situations where this is considered to be valuable information. To generate this query, the following modes are necessary at least:

$mode(2, buys(+, \backslash)).$
$mode(2, price(+, -)).$
$mode(1, low(+)).$
$mode(1, high(+)).$

However, these modes could also be used to generate this query:

$$\begin{aligned} client(-C) \wedge buys(+C, \backslash P_1) \quad &\wedge \quad price(+P_1, -R_1) \wedge low(+R_1) \\ &\wedge \quad price(+P_1, -R_2) \wedge high(+R_2). \end{aligned}$$

If a product has only one price, this query contains redundant atoms and it is better when it is not generated at all. This could be achieved by giving the *price* mode a number. In that case, the second *price* atom will never be generated as a child of the first, and as a result of the tree construction mechanism, it will neither occur in this query. □

As has been shown in the examples in this section, the mode numbering mechanism can be a useful tool to restrict the space of queries even further.

Interesting Queries (1)

The FARMER algorithm can generate huge sets of frequent queries. If the set is too large, it is impossible to discover the really interesting information; in fact, a new algorithm is needed to find those queries which really contain information. In this section a limited mechanism is discussed which is currently embedded into FARMER. Before proceeding to the algorithm which is used by FARMER, the mechanism will be illustrated for a simple example.

Example Consider the following frequent pseudo query:

$$q = client(-C), buys(+C, chips), buys(+C, chocolate) \tag{5.5}$$

Assume that the support of this query is 100. The following queries must also be frequent:

$$q_1 = client(-C), buys(+C, chips) \tag{5.6}$$
$$q_2 = client(-C), buys(+C, chocolate) \tag{5.7}$$

Suppose that query q_1 also has support 100, then that query does not add much information to query q. Query q_1 could safely be reconstructed from query q. However, if query q_2 has support 500, that could be an interesting query. Its support is remarkably higher than that of a larger query. It could be desirable to still show that query. □

Theoretically, for every frequent query one would like to find the set of queries which imply that query. Only if the support is much higher than the maximum support in that set, the query is interesting. Formally a query q is interesting iff:

$$\forall q' \in \mathcal{Q} : \; q' \models q \Rightarrow \mathit{support}(q) > f(\mathit{support}(q')) \tag{5.8}$$

where $f(x)$ is an increasing threshold function which depends on the support of another node, for example:

$$f(x) = c + x \qquad\qquad (c \geq 0) \tag{5.9}$$

or

$$f(x) = c \cdot x \qquad\qquad (c \geq 1). \tag{5.10}$$

As the $\models$ relation can not be computed, an approximation is used which is based on the subset relation.

Definition 5.2.16 *Let q_1 and q_2 be two pseudo queries. Pseudo query q_2 is a subset of $q_1 = A_1 \wedge \cdots \wedge A_n$ (denoted by $q_2 \subseteq q_1$) if there is a query $q_2' = A_{i_1} \wedge \cdots \wedge A_{i_k}$ $(1 \leq i_1 < i_2 < \cdots < i_k \leq n)$ for which there is a variable renaming substitution θ such that $q_2'\theta = q_2$.*

Note that q_2 is also a pseudo query. Not every subset of a pseudo query still is a pseudo query. If a pseudo atom is removed which introduces a variable that is used used later on, that atom can not be removed without removing also the atoms which depend on it.

As FARMER uses a deterministic variable naming mechanism, the variable renaming substitution is easily found if there exists one.

Interesting queries in FARMER are found by replacing the $\models$ relation with the $\subseteq$ relation.

Definition 5.2.17 *The set of interesting queries in* FARMER *is defined as:*

$$\mathcal{Q}_f = \{q \in \mathcal{Q} \,|\, \forall q' \in \mathcal{Q} : \; q \subseteq q' \Rightarrow \mathit{support}(q) > f(\mathit{support}(q'))\}. \tag{5.11}$$

For the computation of this set, the observation is used that the following definition is equivalent:

$$\mathcal{Q}_f = \{q \in \mathcal{Q} \,|\, \forall q'' \in \mathcal{Q} : \; q \subseteq q'', |q''| = |q| + 1 \Rightarrow \mathit{support}(q) > f(\mathit{support}(q''))\}. \tag{5.12}$$

This can easily be seen by considering the $q' \in \mathcal{Q}$ for which $q \subseteq q'$. At least one of the pseudo atoms in q' but not in q can be added to q, resulting in a query

```
Mark all nodes as interesting
for all nodes in the tree do
   Let q be the query ending in this node
   for all atoms in q do
      if the atom can be removed without violating modes then
         search the remaining query q' after renaming
         if q' is found and support(q') ≤ f(support(q)) then
            mark the node in which q' ends as uninteresting
```

Figure 5.9: An algorithm for marking those queries which are interesting.

q''. As this query is not more specific than q', $support(q'') \geq support(q')$ and consequently $support(q) > f(support(q'')) \geq f(support(q'))$. For every query q' such a query q'' of size $|q| + 1$ can be constructed. If the support of query q is higher than the thresholded support of queries of size $|q| + 1$, then the support must also be higher than that of all larger queries.

The algorithm which computes the set of interesting queries is shown in Figure 5.9. Note that q' might not be in the tree, although it is a subset of q and q is in the tree. This is caused by the isomorphism mechanism.

Example Consider this pseudo query, which is taken from Figure 5.6:

$$client(-C), buys(+C, \backslash P_1), good(+P_1), cheap(+P_1), buys(+C, \backslash P_2), good(+P_2). \tag{5.13}$$

The $good(+P_1)$ pseudo atom can be removed. The resulting query is:

$$client(-C), buys(+C, \backslash P_1), cheap(+P_1), buys(+C, \backslash P_2), good(+P_2). \tag{5.14}$$

This query does not exist in the tree, but there should always be an isomorphic query. □

This mechanism will thus be able to prune some rules, but certainly not all in a clear fashion. Furthermore, it is not based on any of the notions of interestingness which were introduced in Section 4.2.3. Later in this section an idea is given for a mechanism which uses the same notion of interestingness.

5.3 Experimental Results

In order to get an impression of the efficiency of Farmer, both Farmer and Warmr have to be executed on the same datasets. In this section, two comparisons are carried out. In the first experiment, Farmer and Warmr are compared on a typical ILP dataset. In the second experiment, Farmer and Warmr are also compared to an efficient implementation of Apriori using a basket database. This should give an impression of the usability of first-order algorithms for propositional cases.

It should be noted that the implementation of WARMR which has been used is not currently the most efficient, and that a comparison of both algorithms is therefore not completely fair. In [Blockeel, 2000] an execution mechanism is discussed for WARMR which also exploits the tree structure of the search space by sending a *pack* of queries to a PROLOG engine. In the experiments with this query mechanism, speedup factors of 20 are obtained for WARMR in some situations.

In the FARMER algorithm which was used, the basic mechanism for finding interesting queries was disabled. It was conceived that a comparison with WARMR would be more fair in that case, as a similar straightforward mechanism could also easily be implemented for WARMR as a post procedure.

5.3.1 Bongard

Description

The Bongard dataset [Bongard, 1970] contains descriptions of artificial images. The task is to discover patterns in the images. Every image consists only of circles, triangles and rectangles. For the triangles, there are two directions: up and down. The orientation of rectangles is not given. Figures can be included into each other, and the dataset contains for each figure its inclusion relation to other figures.

Results

In Figure 5.10 the results of the experiments are depicted. Figure 5.10(a) shows the number of queries that each algorithm finds. The number for FARMER is higher in all cases, which is also expected as the bias is such that isomorphic queries can occur that FARMER does not detect. For supports lower than 0.2 the difference is certainly relevant, with even 1000 queries more at a minimum support of 0.05. It should however be remarked that the experiment is carried out without a measure of interestingness; if those queries are depicted which are interesting, 'only' 583 of the 5000 queries remain. The output of FARMER is complete, in the sense that it also finds all queries that WARMR discovers.

In Figure 5.10(b) the execution times of the algorithms are compared[2]. FARMER certainly is faster, perhaps by the lack of a complete isomorphism algorithm.

5.3.2 Frequent itemsets

Description

Usually association rules are used in basket analysis. Although frequent queries are more general, FARMER can still be used to perform basket analysis. In

[2] Experiments were carried out on a Sun Enterprise 450 4x400MHz UltraSPARC2 CPU w/4MB E-cache 4GB RAM.

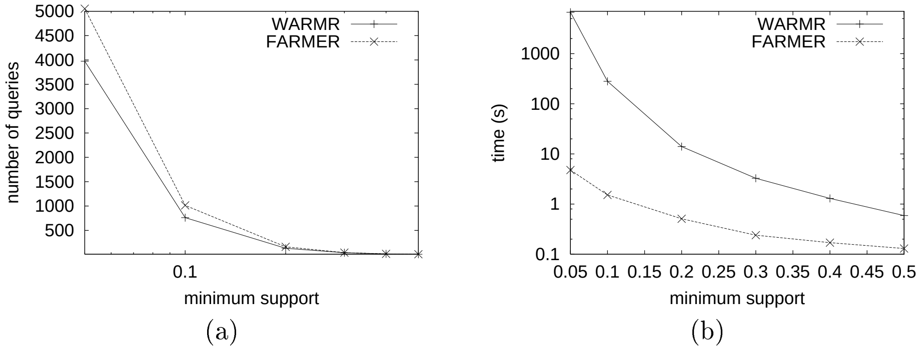


Figure 5.10: A comparison of FARMER and WARMR on the Bongard dataset. (a) The number of queries in the output. (b) The execution time consumed to compute the output. Note that the scales are different.

this experiment, the performance of FARMER is compared to a special purpose algorithm.

As test case a binary digit dataset is used. There are 7 items each of which corresponds to a LED (light emitting diode) of a calculator display. For every number between zero and nine, a set of corresponding items can be obtained by listing the LEDs which are activated when displaying the number. The dataset contains 1000 numbers which are coded in this way.

To the set of 7 items, 93 additional items are added; these are randomly added to the transactions using a uniform distribution.

The special purpose algorithm which is used as comparison is the breadth-first implementation of Apriori by [Pijls, 1999]. FARMER uses many of the mechanisms introduced in that algorithm and should perform comparable to that algorithm.

Results

In Figure 5.11 the results of the experiments are depicted. A characteristic of the dataset is given in Figure 5.11(a). The number of frequent itemsets appears to increase rapidly when the minimum support is beneath 0.1. It is remarkable that there is a 'mountain' between 0.01 and 0.1.

In Figure 5.11(b) the execution times of the algorithms are given. The 'mountain' recurs in the times for FARMER, which are comparable to those of APRIORI, although still exponentially larger. WARMR has a completely different behaviour than both other algorithms and had such high execution times that no experiments were carried out for low supports.

In Figure 5.11(c) both previous graphs are combined and the execution time for each itemset is shown. It makes clear how the algorithms react when the amount of solutions they have to find increases. While the execution times for WARMR decrease (with one exception), the times for the other algorithms in-

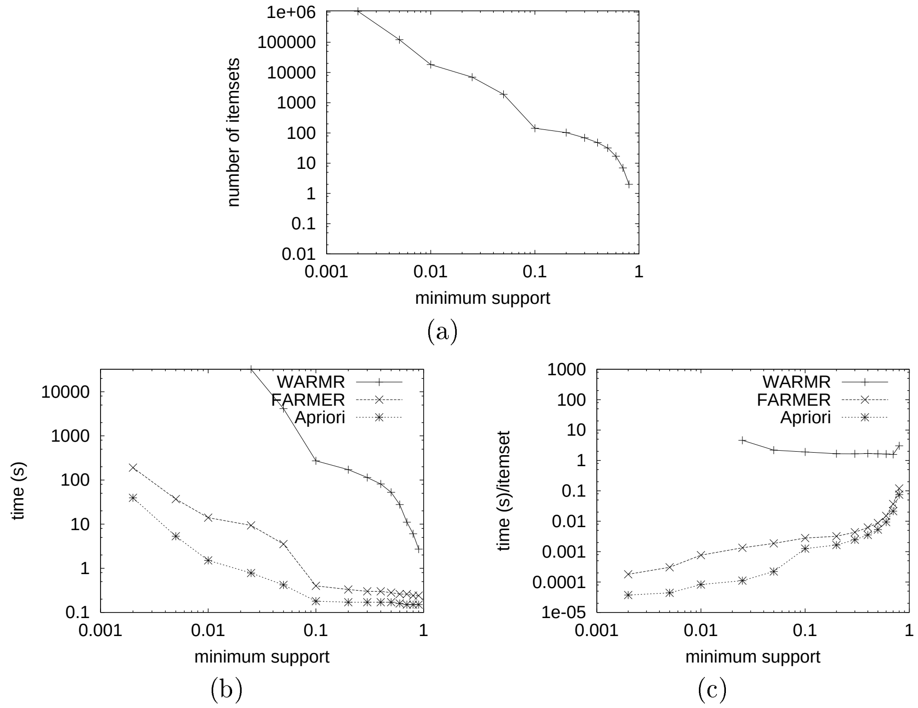


Figure 5.11: Comparison of results for the LED dataset. (a) The number of frequent itemsets in the dataset for each minimum support. (b) The execution times of several algorithms needed to find the frequent itemsets. (c) The execution times consumed for each itemset. Note that the scale is logarithmic on both axes.

crease. Although the overhead for each itemset is larger in FARMER, which could be explained by the additional mechanisms that are hooked in, the difference is acceptable. The increasing trend is caused by the tree data structure, which results in more overlapping evaluations when the number of itemsets increases.

5.3.3 Conclusion

In this section, several experiments have been carried out to investigate the performance of FARMER. It has been shown that FARMER is far more efficient than WARMR from a complexity point of view. Its execution times are lower, and also the time per itemset decreases as the number of frequent queries increases. It has been shown that the performance of FARMER is not as good as that of a specialized algorithm such as APRIORI, but it is still very well comparable.

However, the lack of a complete isomorphism mechanism has also been shown to have redundant queries as a result, thus undesirably making the analysis of the results more complex. The loss in the number of queries is not as large as the gain in efficiency, though.

5.4 Future work

This section contains an overview of some ideas on mechanisms which could be added as future extensions to FARMER. The first extension could provide a speed-up to FARMER by applying a first-order equivalent of APRIORI-TID. The second and third extensions could speed-up as well as prune the set of results by removing some redundant atoms. The fourth extension is a measure of interestingness which could prune the set of results.

5.4.1 FARMER-TID: a variation of FARMER

By discriminating three types of children, also another evaluation strategy could be implemented. The idea is based on algorithms which have previously been discussed in [Agrawal, 1996] for association rules. To every leaf in the tree sets of key variables belong which satisfy that leaf. This set could be saved together with the leaf. In every round, only those sets of key variables which belong with leafs have to be evaluated; other sets are not needed in the round and will not be needed in the next rounds. Furthermore, it would also be possible to determine for some queries beforehand which sets of key variables satisfy the query. Every query which is created by combining two shorter queries is satisfied by the set of key variables in the intersection of the previous key variable sets, except when the added atom has a backslash output.

The disadvantage of this algorithm is that it uses even more memory, especially in the first rounds. Furthermore, although some sets of children can be computed at beforehand, it is not obvious that much gain can be obtained by doing this; still a lot of queries have to be evaluated by passing parts of the database. A reasonable intermediate algorithm therefore could be an algorithm which remembers only one general set of key variables that still are used. This would still allow to consider only those examples which are needed during the evaluation, while on the other hand less memory is needed to save all leafs.

5.4.2 Further pruning

In some biases, there are atoms which can always be proved to be correct for a set of inputs. These atoms actually have as only purpose the introduction of variables which can be used later. Although these atoms can be very useful, they often spoil the results, especially in the mechanism which was discussed previously for finding interesting queries. This mechanism only displays queries which are as long as possible. A longer query may easily be obtained by adding one or more atoms which are always true. As long as the variables introduced by these atoms are not used, they are useless and should rather not be displayed in the results. The simplest solution is to determine beforehand which atoms can always be proved, and to remove such atoms from the queries in the result when the output variables are not used. More elaborate strategies may also be used, one of which is discussed in the next section.

5.4.3 A real equivalence mechanism

In FARMER some attempts have been undertaken to introduce a faster equivalence mechanism. These attemps have not yielded satisfactory results with a sound theoretical basis. Rather, ad-hoc solutions were provided for some equivalences. As it became clear that a good equivalence mechanism is in some cases a necessity to prune the number of results, it is necessary that further research is performed to find an efficient mechanism which has better predictable results and a sound theoretical basis.

5.4.4 Interesting Queries (2)

The mechanism of Section 5.2.5 for discovering interesting queries, is rather basic. Many queries are still part of the output which probably are not considered to be interesting. In this section an idea is discussed for a more elaborate mechanism.

For reasons of simplicity, the mechanism is only valid for biases which have *exactly one* input variable for each mode. In this case a query never contains loops, i.e., variables that are checked together. If for such a bias a mode conform subset is taken of a query, the complement can be partitioned into sets of which each has exactly one input variable.

Example The following query can be partitioned:

$$\begin{aligned}&client(-C), buys(+C, a), hascard(+C, -P), cardinfo(+P, -Y, -T),\\ &type(T, extracard), type(T, childrenscard).\end{aligned} \tag{5.15}$$

The partitions are:

$$\begin{aligned}&client(-C), buys(+C, a), hascard(+C, -P)\\ &cardinfo(+P, -Y, -T), type(T, extracard), type(T, childrenscard)\end{aligned}$$

The first partition denotes the clients which buy a and have a card. The second partition denotes the cards which are of types *childrenscard* and *extracard*. □

A further advantage of such a restricted bias is that also the tree building procedure can be optimized slightly when there are atoms that only introduce variables (cf Section 5.4.2). Those children of such an atom which do not use any of its output variables, need not be expanded further, as there is no other possible way that one of its variables will ever be used later on[3].

[3] There may be some possibilities to build a better isomorphism mechanism for such biases too.

The specialized form of the bias has a consequence that queries fall apart into several pieces of which each has one input variable. In the following examples it is illustrated how for each of these pieces a support can be computed, and how these supports can be combined into an expected value for the total query. If for some subset the expected value equals the real value, the query is considered not to be interesting.

Example Consider this query:

$$client(C), buys(C,P), chips(P) \tag{5.16}$$

and assume the following knowledge:

$$\begin{aligned} p(\{client(C), buys(C,P), chip(P)\}) &= support_C(\{client(C), \\ &\quad buys(C,P), chips(P)\}) \\ &= 0.4, \qquad (5.17) \\ p(\{client(C), buys(C,P)\}) &= support_C(\{client(C), buys(C,P)\}) \\ &= 1.0, \qquad (5.18) \\ p(\{chips(P)\}) &= support_P(\{product(P), chips(P)\}) \\ &= 0.4, \qquad (5.19) \end{aligned}$$

where p is the well-known probability operator and the latter term denotes the fraction of products of type *chips* in the total set of products. In this case the initial query might not be very interesting, as it reflects the distribution of products in the supermarket. The subscript notation is used to make explicit in which table the support is counted. □

Thus, the idea is to introduce for each partition of the query a new key which binds the free variable. The original bias with this new key can be used in another run of FARMER, which results in a support for this subquery (and also for other subqueries)[4]. In the easiest case, the supports of the subsets can be multiplied to compute the probability that they occur together, which results in an expected value. There are however some situations where the computation is more complicated.

Example Another valid partition of query (5.15) is:

$$\begin{aligned} &client(-C), buys(+C,a), hascard(+C,-P), cardinfo(+P,-Y,-T), \\ &type(T, extracard), type(T, childrenscard). \end{aligned} \tag{5.20}$$

Multiplying with the support of the query $t(-T), type(+T, childrenscard)$ would not be reasonable, as it is already known that $type(+T, extracard)$. The solution is to use the following fraction for this subquery instead:

$$\frac{support(\{t(-t), type(+T, childrenscard), type(+T, extracard)\})}{support(\{t(-T), type(+T, extracard)\})} \tag{5.21}$$

This term reflects the well-known formula for conditional probability. □

[4] It could also be a problem to determine the minimum support for this experiment.

In the general case all atoms have to be collected which restrict the value of the input variable even more. These additional atoms are then used to compute a probability for the subset as illustrated in the example. The computed probabilities are finally multiplied to obtain an expected value.

In the next example, a sketch is given how this mechanism can be applied to taxonomies in order to obtain comparable results as for some specialized algorithms.

Example For a taxonomy, the following bias can be used, which is similar to the bias discussed in Section 4.3.5:

$$\begin{aligned}
&predicate(buys(Client, Product)).\\
&predicate(is_type(Product, Type)).\\
&predicate(is_p(Product, Product)).\\
&mode(3, buys(+, \backslash)).\\
&mode(3, is_type(+, shoes)).\\
&mode(3, is_p(+, Nike), n_1).
\end{aligned} \tag{5.22}$$

This could result in such a query:

$$client(-C), buys(+C, \backslash P), is\_type(+P, shoes), is\_p(+P, Nike). \tag{5.23}$$

A partition is:

$$\begin{aligned}
&client(-C), buys(+C, \backslash P), is_type(+P, shoes)\\
&is_p(+P, Nike)
\end{aligned} \tag{5.24}$$

The following fraction will be used as prediction:

$$\frac{support_C(\{client(-C), buys(+C, \backslash P), is\_type(+P, shoes)\}) \times support_P(\{product(-P), is\_type(+P, shoes), is\_p(+P, Nike)\})}{support_P(\{product(-P), is\_type(+P, shoes)\})} \tag{5.25}$$

The latter term exactly denotes the fraction of Nikes within the range of shoes. □

Some further adaptations will be necessary before FARMER can really emulate taxonomies, though. The current bias still allows the creation of a query that does not contain an *is_type* atom. This query will not be filtered out by a subset computation mechanism. One possible solution could be to enhance the mode numbering mechanism such that *is_p* atoms are not allowed as children of $buys(+C, \backslash P)$ atoms, but are still included in larger queries.

The computations which are necessary for this mechanism could be performed reasonably efficient in FARMER, taking advantage of the trees which are also usable as efficient data structure for finding supports.

The mechanism is currently however somewhat restricted, and a sound proof based on probability theory was not given. There may also be some possibilities to port the more elaborate mechanism for taxonomies to FARMER as well.

This mechanism has to take into account the problem of backslashes, which was neglected here, and the problem of "duplicated" atoms. Eventually, it should lead to a procedure which could deduce the following estimation automatically:

$$\underbrace{client(-C), buys(+C, \backslash P_1)}_{\text{A}},\quad is\_p(P_1, a), is\_type(P_1, t_1),\quad \underbrace{buys(+C, \backslash P_2), is\_p(P_2, b)}_{\text{B}}.$$

$$\Rightarrow$$

$$support(\{A, is\_type(P_1, t_1), B\}) \times \frac{support(\{A, is\_p(P_1, a)\})}{support(\{A, is\_type(P_1, t_1)\})}. \qquad (5.26)$$

This formula can be compared to formula 4.3, which was obtained for taxonomies. In this comparison a would be Nike and b would be juice; t_1 would stand for "shoes".

A study of probability networks may also give clues about other measures of interestingness that are possible here.

5.5 Summary

In this chapter, a description was given of the Farmer algorithm. The idea was to improve the performance of Warmr by dropping the need for a Prolog compiler or interpreter, and by using an efficient tree data structure that was described in [Pijls, 1999]. In experiments it was shown that a considerable speed-up was obtained in comparison with the original algorithm. However, also the number of resulting queries increased in some cases. To cope with this undesirable situation, some mechanisms were introduced to reduce the number of rules, but they were shown not to be sufficient. Therefore some ideas were discussed about future extensions which could reduce the results even further. One of these ideas is the notion of interesting rules. It was illustrated how this notion could look like in Farmer and how this could result in an algorithm which also emulates some specialized algorithms better.

Chapter 6

Decision Trees and TILDE

Early in the eighties, a classification algorithm was introduced which became very popular afterwards: the ID-3 algorithm for building decision trees. Especially one of its descendants, C4.5, has been applied extensively ever since. Also for this algorithm the main reasons for its popularity are the simplicity of its results and its efficiency.

In the mean time, ILP researchers were investigating the problem of learning definitions of predicates, which is in fact a binary classification problem. Bearing in mind that both algorithms are based on binary decisions, decision trees and ILP were destined to come together one day, and indeed this happened in the TILDE algorithm [Blockeel, 1998]. This chapter discusses the fundamentals of decision tree construction algorithms, and describes how these fundamentals were applied in TILDE.

6.1 Overview

The chapter starts with an introduction to the notion of decision trees. An example is given to illustrate their meaning. In the other sections, the problem of learning such trees is considered in detail. The second section discusses a general tree construction algorithm, of which some elements still have to be specified. In the third section this is done for the C4.5 algorithm. The fourth section describes the shape of decision trees in TILDE and explains how the details of the general construction algorithm are specified in that case. The last section summarizes.

6.2 Decision trees

A decision tree is essentially a classifier. What is a classifier? A classifier is a method which assigns classes to objects. For example, a method which determines whether a supermarket product is good or bad is a classification method. Of couse, many classification methods are conceivable. One could also perform a mathematical computation to perform a classification. What makes a classifier a decision tree? Essential of decision trees is that they determine the classification of an object by repeatedly asking questions. Depending on the answer to the question, a new question is asked or a classification is returned. Furthermore, questions do not have side-effects and, finally, the graph which displays the order in which questions will be asked has the shape of a tree.

These ideas are illustrated in the following example.

Example Consider a situation in which objects have 4 attributes, stored in attribute vectors X. Each attribute can take two values. The classifier should determine whether an object belongs to class A or class B. Then Figure 6.1 shows an example of a decision tree. For an object $X = (1, 2, 2, 1)$ the classification is B: as X_1 is 1, the next attribute tested is X_2, followed by X_4. □

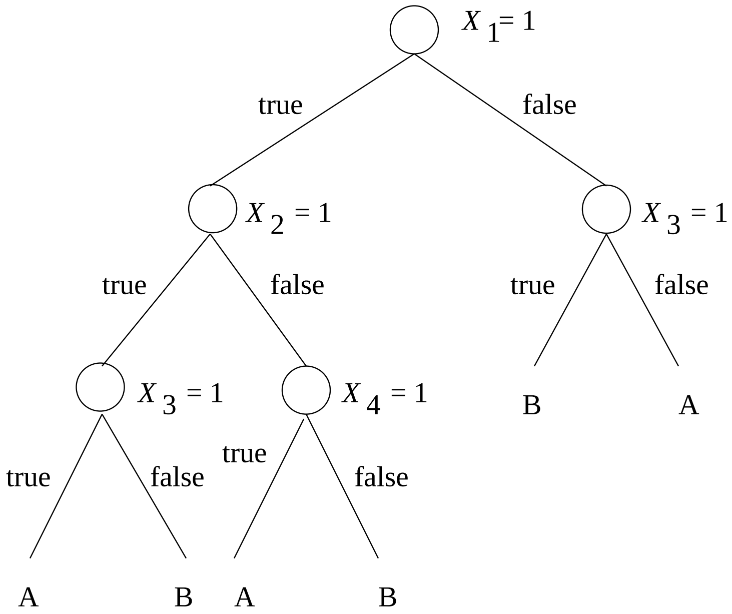


Figure 6.1: A decision tree example.

For every example which is presented to the decision tree, there is thus a unique path from the root to a leaf of the tree.

The main variation which will be discussed in this thesis is that of the tests in the nodes. What kind of tests can occur? In the example, the easiest case is presented: checking the value of one attribute. Another, more complex possibility would be to use logic atoms. That option is investigated in the last section of this chapter.

6.3 A general decision tree algorithm

The creation of decision trees is based on examples. Given some observed examples, a tree should be constructed which not only classifies this examples as desired, but also classifies unseen cases correctly. To obtain good predictions for unknown cases, most algorithms for creating decision trees use *Occam's razor*, the principle that the shortest hypothesis still consistent with most observed examples is probably also the best predictive one.

A general tree construction algorithm is given in Figure 6.2. Several parts need further specification. These are:

- the function which determines a (default) label for a set of examples;
- the first stopping criterion, which decides whether it is considered to be useful to further continue building the tree for a set of examples;
- a second stopping criterion, which decides whether the best test available is useful. Often the stopping criteria are merged into one, but it can be useful to split them up, as the first criterion does not depend upon a test and could be used to avoid searching such a test;
- the best test selection mechanism. This is the most difficult problem to solve in any decision tree algorithm as it influences the shape of the decision tree most.

In the algorithm, the set of possible 'tests' is intentionally left unspecified. The specification of the possible tests will be treated later in this chapter.

In the next sections these subjects will be discussed in more detail.

6.3.1 The labeling function

For the remainder of this chapter we introduce some notation. Given a set of examples E and a label c_i, $p(c_i, E)$ will denote the fraction of examples which are classified with label i. Using this notation, the labeling function $\lambda(E)$ for a set of examples E, is defined as:

$$\lambda(E) = \begin{cases} \lambda(E') & \text{if } E = \emptyset; \\ c & \text{otherwise} \end{cases} \tag{6.1}$$

In this formula, E' is the set of all available examples and c is a label such that $p(c, E) = \max_{c'} p(c', E)$. The main argumentation for choosing labels this way is the following: if the distribution of examples is representative for future cases, then $p(c_i, E)$ approximates the classification accuracy for an algorithm which classifies every example by default as c_i. As this accuracy should be as high as possible, the label should be chosen which maximizes $p(c_i, E)$.

```
create_tree ( E )
if stopping criterion holds for E then
  return label for E
else
  t := select best test ( E )
  if t does not split E enough then
     return label for E
  else
     n := new node for test t
     E_l = examples in E for which t is true
     E_r = E\E_r
     n.left = create_tree ( E_l )
     n.right = create_tree ( E_r )
     return n
```

Figure 6.2: An algorithm for constructing decision trees. The set E is a set of examples for which the classification is known.

6.3.2 The stopping criterion

Essentially, a decision tree algorithm is a splitting algorithm. Each newly created test should split up the initial set of examples into two subsets. However, in some cases it is not useful to find a split-up once more. These are the three situations which can occur and the most frequently used tests to flag these situations:

1. the set contains many examples which all have the same classification. The following test can be used: $p(c_i, E) \geq \theta$ for some class c_i and some previously defined threshold parameter θ[1];

2. the set is considered to be too small, such that a split-up based on this set probably is not representative and does not provide good predictive qualities. To test this, the size of the set is compared to some threshold value;

3. the best split divides the set into two subsets which both have the same distribution of classes as the original one. This topic is closely related to the heuristics which are used to find a best split, and therefore will be treated in more detail when discussing the heuristics in the next section.

6.3.3 Finding the best split

To find the best test, the notion of *impurity* is used. Impurity is a measure of the chaos of classifications. If all examples are classified in the same way, the

[1] Parameter θ is often chosen to be 1.0.

impurity is zero; if each classification occurs in the same amount, the impurity measure should have its maximum value. Commonly used measures which confirm to these requirements, are:

$$\text{class entropy:}\quad I(E) = -\Sigma_i p(c_i, E) \log p(c_i, E); \tag{6.2}$$

$$\text{Gini index:}\quad I(E) = 1 - \Sigma_i p^2(c_i, E). \tag{6.3}$$

As the tree should be as simple as possible, the number of tests should be minimal. Impurity can be used as a heuristic for the number of tests still to come: if the classifications of examples in a node differ a lot, probably a lot of tests will be necessary to separate them from each other. Given a set of examples, that test should be selected which maximizes the impurity gain:

$$\Delta I = I(E) - \frac{|E(t.left)|}{|E|} I(E(t.left)) - \frac{|E(t.right)|}{|E|} I(E(t.right)), \tag{6.4}$$

where $E(t.left)$ denotes the set of examples in E for which the test t succeeds, and $E(t.right)$ denotes the set of examples in E for which the test t does not succeed. If class entropy is used as impurity measure, this is also called *information gain*. A slightly more complex gain measure is the *information gain ratio*, which is obtained by dividing ΔI by the so-called *split* term:

$$-\frac{|E(t.left)|}{|E|} \log \frac{|E(t.left)|}{|E|} - \frac{|E(t.right)|}{|E|} \log \frac{|E(t.right)|}{|E|}. \tag{6.5}$$

This additional term was introduced in $C4.5$ for nodes which have more than two branches, but in some experiments it has also been succesfully applied to the two-branch case. As this term has the highest value when the split divides all examples equally between its children, this particular situation will be punished by this term.

The additional stopping criterion is based on the impurity gain criterion. If the best test available still has a very small impurity gain (smaller than some threshold value), then it is better not to refine. The following may happen if the test is applied: in the next refinement step, a new test has to be found. This test can not possibly provide a good split, as otherwise it would have been preferred to the earlier test. Thus, the subsequent tests will also split a little, which might eventually result in a good description of the examples, but which is certainly not a good generalization procedure.

6.4 Decision trees in C4.5

The most well-known decision tree algorithm is C4.5 [Quinlan, 1993a]. The algorithm C4.5 is a so-called attribute-value learner, which means that every example can be described entirely by giving the values for a fixed set of attributes. A classifier should perform its task using tests on these attributes.

C4.5 allows two kinds of attributes: discrete and continuous. For the first type the '=?' test is available, which can be used to split upon the value of an attribute. In case an attribute has more than two values, this means that a corresponding testing node also has more than two children. The impurity formulas which were previously discussed, are adapted accordingly.

The other type of test is the lower than/higher than test for continuous attributes. Nodes with this test always have two branches. To find the best threshold, a set of possible thresholds is collected from the examples as follows:

1. list all values for the attribute;

2. sort the values;

3. compute the means between all two adjacent values;

4. return the means as threshold candidates.

The threshold which obtains the highest information gain (*not* gain ratio), is considered to be the best.

However, the entropy measure previously discussed has a major disadvantage for continuous attributes. A split for an attribute which has a lot of different values, probably can be optimized much better. The attribute which has the most values, will therefore be preferred. This could cause unimportant attributes to be chosen.

To correct for this preference, Quinlan proposes to use the *Minimum Description Length* principle [Quinlan, 1996a, Rissanen, 1983]. The minimum description length principle states that the theory should be chosen which minimizes the sum of two factors:

- a measurement for the number of examples which violate the theory;

- a measurement for the size of the theory.

The MDL principle thus formalizes the trade-off between size and accuracy of a theory. In the algorithm C4.5 the following measurements are used:

- the exception costs are approximated with the class entropy of the best threshold;

- the size of the theory is estimated by $\log N/|E|$, where N is the number of thresholds and E is the set which must be split by a node.

Effectively the information gain is thus decreased with the latter term, and splits for attributes with many values are punished.

The new information gain term can be divided by the split term afterwards to obtain a new gain ratio measure[2].

[2] However, C4.5 does not use the gain ratio to find the best split, as this was experimentally shown not to be beneficial.

6.5 Logic decision trees and TILDE

6.5.1 Using sets of atoms as tests

A logic decision tree is obtained by using atoms or sets of atoms as nodes in the decision tree. An example is given in Figure 6.3. Contrary to an ordinary decision tree, however, a test does not correspond directly to a node of the tree. The splitting test is obtained by collecting all the atoms which have not been answered with *false* on the path from the root. So, for example, for the node marked with a star in the example decision tree, the test is

$$buys(P), sale(P). \tag{6.6}$$

Also TILDE is based on the learning-by-interpretations principle which was discussed in Section 3.3.4. To determine the validity of query (6.6) for an observation, three steps are executed:

1. add all facts belonging to the observation to a PROLOG knowledge base;
2. ask the query to the PROLOG engine;
3. remove the facts of the observation from the knowledge base, in order to prepare for a next observation.

In the example, an observation could consist of a set of *buys* facts and one *donor* fact. The usage of a PROLOG engine implies that the variables in the query are existentially quantified.

6.5.2 Finding nodes

How are the contents of a node determined? The *mode* mechanism which was introduced by PROGOL and is used by WARMR, is also used by TILDE[3] to generate candidate child nodes. For the example decision tree, the set of modes could be the following:

$$\begin{aligned} mode(1, buys(-P)) \\ mode(1, cheap(+P)) \\ mode(1, donor) \\ mode(1, sale(+P)) \end{aligned}$$

At the node which is marked with the star, the only possible new node is *donor*, as all other modes have already been used. The evaluation mechanism which was discussed in the previous section is used to determine how the observations belonging to a candidate node split up. The information gain (ratio) criterion then is used to find the node which provides the best split of the observations.

Although this mechanism could suffice in a propositional case, it is not enough for a logic decision tree. For example consider the $buys(-P)$ mode.

[3] There is also a version which uses DLAB templates, see Section 3.3.4.

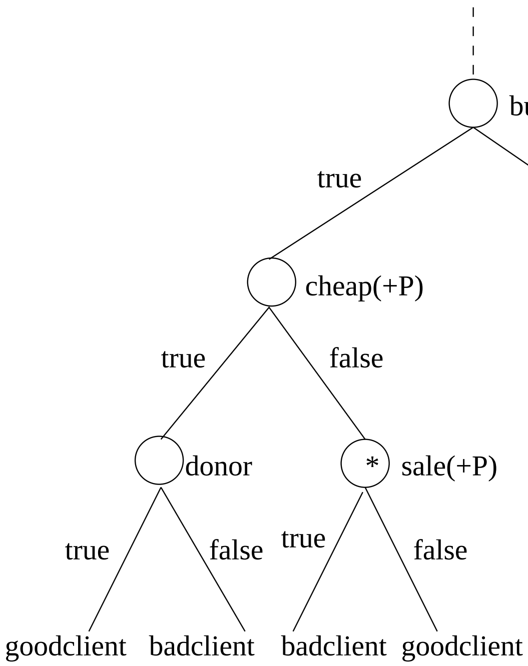


Figure 6.3: An example of a logic decision tree in TILDE.

As it is reasonable that every client buys a product, a candidate node belonging to this mode will not give a good split: for all observations $buys(-P)$ can be proved and the observations will all go to the left branch. However, this does not mean that this node is useless: other nodes may be introduced later in the tree which do add information. In the example, it is probable that the combined test $buys(P), cheap(P)$ does provide a good split.

To cope with this problem, TILDE provides two mechanisms:

- allow more than one atom in a mode, for example:

$$mode(1, (buys(-P), cheap(P))) \tag{6.7}$$

  The names of the parameters indicate that in both places the same variable should be used. The atoms will always occur together in a node in this order, and will be added to an evaluation test together;

- a lookahead mechanism, which operates using a set of lookahead definitions, for example:

$$lookahead(buys(P), cheap(P))$$
$$lookahead(buys(P), sale(P))$$

  Such a lookahead definition essentially means the following: if $buys(P)$ is part of a valid candidate node, then also nodes to which $cheap(P)$, etc., are added, are valid candidates. In this way, also nodes are created which contain several atoms. The lookahead mechanism thus can be considered as a method for generating a set of modes containing several atoms.

A consequence of this mechanism is that the length of a path does not necessarily correspond to the number of atoms in a test.

6.5.3 Determining thresholds

In the previous section, a fixed set of modes was provided to TILDE. This does not suffice when also continuous values in facts are allowed. It would not be practical to add a mode like

$$mode(1, lowerthan(X, b)) \tag{6.8}$$

for each possible value of b. A *mode* generation mechanism is also introduced here to solve this problem. Basically, it looks like this:

$$mode(n, generator(atom(m, \_, X, \_, \ldots)) : +P \leq X) \tag{6.9}$$

The generator is an algorithm which creates a set of thresholds X. It has as argument a predicate with one marked parameter and a number m which indicates the maximum number of thresholds. The generator performs two steps to generate the thresholds:

1. collect all values occurring in facts for the marked parameter, remembering with each value the observation to which it belongs;

2. apply a modified version of the Fayyad and Irani discretization algorithm [Fayyad, 1992] to generate thresholds.

Without going too much into details, the discretization algorithm works by recursively splitting the set of values into two subsets such that the impurity of the subsets is minimal. The major modification is in the entropy formula which is used. Usually, the expression $p(c_i, E(t.left))$ is equal to the fraction $|c_i(E(t.left))|/|E(t.left)|$, where $c_i(E)$ denotes the set of observations in E classified as i. In this case, instead of computing a straightforward size of a set, every observation in a set is given a weight, and the size of a set is the sum of weights. When computing for a threshold its entropy, the weight of an observation in the left branch is the fraction of facts in the observation for which the value is lower than that threshold. This is necessary as the same observation can occur in both branches[4]. Thus, at this moment, the $\exists$ quantifier is not taken into account.

[4] Remember that with every observation, there can be several facts, each supplying a different value!

6.6 Summary

In this chapter, a general algorithm for constructing decision trees was introduced. It was shown that there is still some freedom in this algorithm. The popular C4.5 algorithm uses attribute checking tests and exploits the MDL principle to determine thresholds for continuous attributes. Tilde, at the other hand, allows entire logic formulas as tests and uses a preprocessing algorithm to discretize continuous values. These choices in the Tilde algorithm may have some disadvantages:

- experiments in [Quinlan, 1996a] have shown that the MDL principle yields better results more often;
- the tests become much more expensive than they are in C4.5, as a whole path has to be evaluated for each leaf, and a Prolog interpreter/compiler is necessary to do this.

In the next chapter an algorithm is introduced which aims at solving these disadvantages. This will result in a first-order algorithm which looks more like C4.5 than Tilde.

Chapter 7

GILDER

The introduction of logic into decision trees has yielded an algorithm which is already practically executable, but still has some disadvantages. It is not as close to the popular C4.5 algorithm as may be desired, both from the performance point of view as from the algorithmic point of view. In this chapter, an algorithm is introduced which is closer to C4.5 than TILDE. The introduction of logic requires however some additions. In this chapter some different solutions are proposed for these additions, and it will be argued that these additions have a better accuracy as result.

7.1 Overview

This chapter is organized in the same way as the chapter in which FARMER was introduced. The first section gives a thorough description of the GILDER logical decision tree construction algorithm. The second section lists the results of experiments that have been performed with GILDER. The third section gives some ideas about future extensions. The fourth and last section summarizes.

7.2 Algorithm description

In this section, the GILDER algorithm is introduced. Following the scheme of FARMER, first the definition of knowledge and bias is given. The second section describes the basic GILDER algorithm. Those features which deserve extra attention in logic decision trees, lookaheads and continuous attributes, are treated in the fourth and fifth section, respectively.

7.2.1 Knowledge and bias

GILDER inherits most of the declaration mechanisms for knowledge and bias which were introduced by FARMER. These include type definitions, predicate definitions, mode declarations, facts, pseudo-atoms, pseudo-queries and translations. The meaning of keys differs however slightly from that in FARMER, and also the modes differ from those in TILDE. The differences are highlighted in the following.

Keys

An example of a GILDER tree is given in Figure 7.1. One striking difference with a decision tree in TILDE is the usage of keys. Keys are used throughout the whole knowledge base of FARMER, and to maintain consistency GILDER uses them too. The declaration and the meaning of keys in GILDER is slightly different, however.

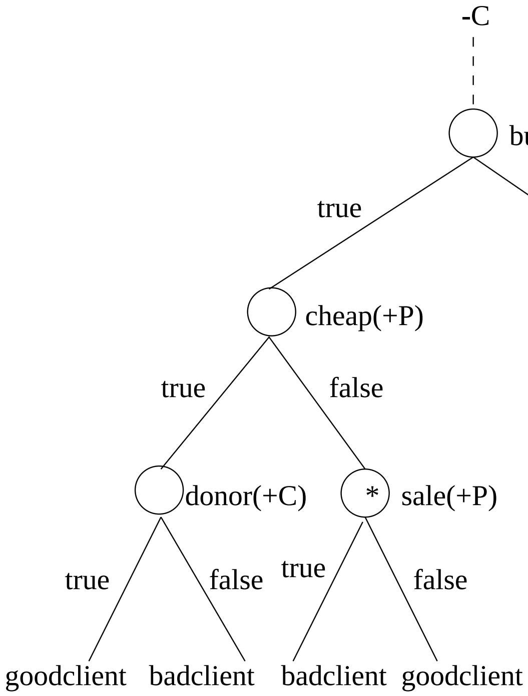


Figure 7.1: An example of a logical decision tree.

One of the parameters of the key predicate is assigned a special function: it indicates the target variable. When the key predicate is appointed, also the parameter must be specified which is the target of the classification. The knowledge base must contain facts for the key predicate; each fact for a key predicate corresponds with an example. The class of the example is the value of the specified parameter. This is thus contrary to FARMER, where facts for key predicates are not allowed.

A further difference is that a key predicate does not appear in the tree.

The predicate is implicitly there to provide the first variables and the class of the example. This usage of key predicates makes sense when one considers the meaning of a decision tree. A path from the root to a leaf corresponds with the following part of a PROLOG program:

$$\begin{aligned} &client(C, good) \leftarrow \neg q(C), buys(C, P), sale(P). \\ &q(C) \leftarrow buys(C, P), cheap(P) \end{aligned} \tag{7.1}$$

A decision tree could be translated into a set of such rules; a PROLOG engine which knows the $\neg$ relation, could afterwards be used to determine for a client whether it is a good client or not.

GILDER clearly does not exploit the learning-by-interpretations mechanism either, although there is a straight-forward correspondence.

Modes

TILDE provides a comprehensive bias mechanism which includes multiple atom modes and lookaheads. GILDER uses the same bias mechanism as FARMER and does not provide any of these features. Only single atoms are allowed as test in a node. The problems which lookahead declarations intend to solve, are tackled by GILDER in a different fashion, which will be described in the next section.

The way the mode mechanism of FARMER is applied to GILDER, needs some further explanation. Definition 5.2.8 formalizes how a set of pseudo-queries $\mathcal{L}$ can be obtained from mode definitions. To determine the candidate nodes which could be added as a leaf to a GILDER decision tree, the following mechanism is applied:

- after collecting in a pseudo-query all atoms which have been answered with 'yes' on the path from the root, a pseudo-query is obtained, for which all valid expansions in $\mathcal{L}$ can be determined in a similar fashion as in FARMER;
- the number of times that a mode is used on the path from the root might not exceed the constant which is given in the mode declaration. This removes some of the candidates obtained by the above mechanism, as also negatively answered nodes are now taken into account. In this way, an easy mechanism is obtained to prevent nodes from being added which have already been answered with 'no', thus avoiding redundancy when computing the best node.

7.2.2 Basic GILDER

This section gives an overview of the basic GILDER algorithm, comparable to the general decision tree algorithm which was discussed in the previous chapter. Special features of GILDER for lookahead modes and continuous attributes are discussed in the next sections.

During the building phase, a decision tree consists of three types of nodes:

- leaf nodes. These nodes contain a classification label;

- open nodes. For each classification label, these nodes only contain a set of key variable values. This node type will not occur in the final tree;

- atom nodes. These nodes contain an atom which is used as test.

Figure 7.2 gives an overview of the GILDER algorithm. It is a combination of the decision tree algorithm in Figure 6.2 and the FARMER counting procedure in Figure 5.4. The *create tree* procedure is the main loop of the algorithm, and leaves freedom for the implementation to expand breadth-first or depth-first[1]. The labeling function and the stopping criterion which are used in this procedure are the same as the ones discussed in the previous chapter and also have the same parameters. For nodes which seem worth a refinement, the *fill in* procedure is called to find an atom for the node.

The *fill in* procedure starts by calling a routine which generates all valid expansion nodes for the current query. This is a generating procedure, no copying is performed as in FARMER. The mode declaration mechanism which was commented in the previous section is implemented in this routine to obtain atoms. For each such atom, an atom node with two open children is generated, and the routine returns a set of such nodes with two children.

The *evaluate history* and *evaluate candidates* functions are used to split the examples in the open node for each of the candidates. This procedure has a strong resemblance with the *count* procedure of the FARMER algorithm. The common path for all of the candidates is evaluated together for each of the examples. By maintaining two sets of nodes, the algorithm only keeps searching for a proof of those candidates which have not been proved already. Furthermore, it also provides the possibility to stop evaluation if all candidates can be evaluated to true for a certain example.

After all candidates have been evaluated, all information is available to find the best new atom for the open node. This selection mechanism is performed by the *best node* procedure, see Figure 7.3. Basically, this function computes the impurity for each of the candidate nodes using the open nodes which are its children. To allow for lookahead, there are however some major additions, which are covered in the next section.

Finally, after the best node is determined, its usefulness is checked. To this purpose, GILDER uses the same thresholding mechanism for information gain which was discussed before.

7.2.3 Lookahead

The problem of using lookahead declarations, is tackled in GILDER by two co-operating mechanisms: automatic detection when lookahead is necessary and a different impurity function. Each of these subjects is discussed in the next subsections.

[1] The current in-core memory implementation is breadth-first, but for databases on disk depth-first probably is more efficient, as it provides better caching possibilities.

```
evaluate candidates ( &proved nodes, &unproved nodes, B1, B2 )
for all node ∈ unproved nodes do
  if I(node, B1) = true then
     add B2 to node.leftchild.set of examples
     remove node from unproved nodes
     add node to proved nodes

evaluate history(depth, history,
                 &proved nodes, &unproved nodes, B1, B2)
if depth = history size then
  evaluate candidates ( proved nodes, unproved nodes, B1, B2 )
else
  if I(history[depth], B1) = true then
     for all B' ∈ ℬ(history[depth], B1) do
        evaluate history ( depth + 1, history,
                           proved nodes, unproved nodes, B1 ∪ B', B2)
        if unproved nodes = ∅ then
           return

fill in(history, node, expansions )
C1 := ∅
C2 := expansions
for all B ∈ node.set of examples do
  evaluate history(0,history, C1, C2, B, B)
  for all node ∈ C2 do
     add B to node.rightchild.set of examples
  C2 := C1 ∪ C2
  C1 := ∅
new node := best node(node, C2, history)
if new node does not split node.set of examples enough then
  node.label :=label node.set of examples
else
  replace node by newnode

create tree
while there is an open node do
  if stopping criterion holds for node.set of examples then
     node.label := label for set of examples
  else
     history := all atoms on the path from the root answered with 'yes'
     candidates := valid expansion nodes of query history
     fill in ( history, node, candidates )
```

Figure 7.2: The basic GILDER algorithm used to create logic decision trees. The parameters prefixed by a & indicate so-called 'reference parameters', which are parameters of which the changes are propagated to the calling function.

best node (*open node*, *candidate nodes*, *history* **)**
init (*best impurity*)
for all *node* $\in$ *candidate nodes* **do**
  if nearly all examples of *open node* go to the left brach of *node* **then**
    add *candidate node* to *history*
    candidates :=valid expansion nodes of query *history*,
      dependent on *candidate node*
    fill in (*history*, *node.leftchild*, *candidates*)
    remove *candidate node* from *history*
  impurity := compute impurity (*node*)
  if *impurity* better than *best impurity* **then**
    best impurity := *impurity*
    best node := *node*
return *best node*

Figure 7.3: The best node function, which determines the best node of a set of candidates.

Automatic lookahead

Figure 7.3 shows the *best node* function, which implements the automatic detection mechanism.

The main characteristic of an atom which does not provide information, but introduces some variables, is that all examples can be satisfied for such an atom. In a decision tree, this means that all examples go to the left branch. Thus, to determine whether lookahead is necessary, GILDER checks for this situation. It uses a minimum threshold on the fraction of examples which go for each classification label to the left branch.

To determine whether the current node can become useful, a subtree is built for the current node. However, not all mode conform expansions are tried. The current node introduces variables and the usefulness of the node should come from the usage of these variables. Therefore, only dependent nodes are generated as candidate children. Dependent nodes are here exactly the same kind of nodes as were also discussed for FARMER in the expansion procedure of Figure 5.3. By recursively calling the *fill in* procedure, the best subtree is created for the current candidate[2]. When finally the candidate with its subtree is determined to be the best, the node and its subtree are returned and pasted into the tree.

[2]It should be noted that more than one candidate might need expansion, and that it would be beneficial to exploit the common path which the several lookaheads have. The current implementation of GILDER does not exploit this.

Impurity function

To determine the best node, an impurity measure can be used. In this section the measures which are used by GILDER are introduced one by one. Every section adds a feature to the previous section; GILDER can be executed using all features, although this is not compulsory.

An entropy function for a subtree Instead of computing the impurity of a node, the impurity function which is used by GILDER should also be capable of computing the impurity of a subtree. The following solution is the most straightforward:

$$I(n) = \begin{cases} \frac{|E(n.left)|}{|E(n)|} I(n.left) + \frac{|E(n.right)|}{|E(n)|} I(n.right) & \text{if } n \text{ is an atom node} \\ -\Sigma_i p(c_i, E(n)) \log p(c_i, E(n)) & \text{otherwise} \end{cases} \tag{7.2}$$

It computes the impurity as if one test splits the set of root node examples into several subsets, for each leaf of the subtree one. This mechanism almost corresponds with the information function which is used in TILDE: as the lookahead mechanism of GILDER only refines left branches, the subtree will always be a kind of list, of which all but one right branches do not contain any examples. The impurity of the subtree (which would be a test in one node in TILDE) corresponds to the impurity of the last node in the list.

To determine which node is better, the impurities of the candidates nodes are compared. The node which has the lowest impurity is the best.

Punishing lookahead (version 1) Function (7.2) has some undesirable properties. In fact, it neglects the results which were obtained in [Quinlan, 1996a] for continuous attributes and attributes with many values. The lookahead mechanism allows for a lot of fine-tuning and it is very reasonable that the preference will go to a completely optimized query instead of one test. However, the longer the query, the more distant from the example the information which is exploited. It is likely that the information which is closest to the example is also the most important and should be preferred. Distant information can be very useful[3], but should therefore be used with care.

The solution which is chosen in GILDER is to introduce punishments for performing lookahead. The general formula is:

$$I(n) = \begin{cases} p_1(n)\frac{|E(n.left)|}{|E(n)|} I(n.left) + p_2(n) + \frac{|E(n.right)|}{|E(n)|} I(n.right) & \text{if } n \text{ is an atom node;} \\ -\Sigma_i p(c_i, E(n)) \log p(c_i, E(n)) & \text{otherwise.} \end{cases} \tag{7.3}$$

Functions $p_1(n) \geq 1$ and $p_2(n) \geq 0$ are used to punish for lookaheads. There are two functions to offer maximal flexibility. Function p_1 can be used to enlarge the impurity in the left-hand subtree. The larger the impurity, the heavier the

[3] In fact, that is the main reason for introducing predicates in decision trees

punishment will become. Thus if an optimal split is obtained ($I(n) = 0$) there will be no punishment for the lookahead. Function p_1 can typically be used to prevent subtrees from being chosen which offer only a slightly better split.

Function p_2 adds a punishment for the lookahead itself, independent of the result which might be obtained from doing this.

The most basic function which can be used for the punishment functions is

$$p_i(n) = c_i \tag{7.4}$$

for some constants c_i. However, this might not always be desirable. In some databases, there is for each set of input values of a mode only one set of output values. In this case a predicate could be considered as a procedure which returns an attribute for a given object. Such a lookahead should not be punished. A general solution is to use the following functions:

$$p_1(n) = max(1.0, c_1 \times \text{average size of the output set for the mode of } n) \tag{7.5}$$

and

$$p_2(n) = c_2 \times \text{average size of the output set for the mode of } n) \tag{7.6}$$

where c_1 and c_2 are some constants. Other possibilities are:

$$p_1(n) = max(1.0, c_1 \times \log(\text{average size of the output set for the mode of } n)) \tag{7.7}$$

and

$$p_2(n) = c_2 \times \log(\text{average size of the output set for the mode of } n); \tag{7.8}$$

in this case the MDL principle is applied in the same way as in C4.5.

The following example illustrates how the average size of the output set for the mode of n can be obtained.

Example The following database of facts is assumed:

$p(1, 2, a, b).$
$p(1, 3, a, c).$
$p(2, 3, a, b).$
$p(2, 3, b, c).$

Furthermore, the mode of the node considered is

$mode(1, p(+, +, -, -)).$

The set of inputs for this mode is $\{(1, 2), (1, 3), (2, 3)\}$ and consequently the number of inputs is 3.

The multiset of outputs for this mode is $\{(a, b), (a, c), (a, b), (b, c)\}$ and consequently the number of outputs is 4.

If $c_1 = 1$, the punishment becomes

$$p_1(n) = max(1.0, 1 \times \frac{4}{3}) = \frac{4}{3}. \tag{7.9}$$

□

The punishment values for each mode can be computed beforehand, so this procedure will not slow down the lookahead mechanism.

Punishing lookahead (version 2) Another possibility is to apply a similar punishment term as was used in C4.5 for continuous attributes. A straightforward solution is to use the initial entropy function and add a punishment term $c \log d/|E(n)|$, where d is the depth of the subtree. Nodes with a larger subtree then receive a larger punishment.

Gain ratio The impurity functions discussed in the previous section can also be used to obtain a gain ratio measure. The mechanism which is used by GILDER is straightforward:

$$\mathit{gain\_ratio}(n) = \frac{-\Sigma_i p(c_i, E(n)) \log p(c_i, E(n)) - I(n)}{\mathit{split}(n)} \tag{7.10}$$

where

$$\mathit{split}(n) = -\Sigma_{n' \in \mathit{leafs}(n)} \frac{E(n')}{E(n)} \log \frac{E(n')}{E(n)} \tag{7.11}$$

and

$$\mathit{leafs}(n) = \{n' \mid n' \text{ is a leaf in the subtree of } n\}. \tag{7.12}$$

7.2.4 Continuous attributes

TILDE uses a discretisation algorithm to compute tests for continuous variables beforehand. This is a rather different approach than the mechanism used by C4.5, which computes the threshold on-the-fly. In this section, the method used by GILDER is discussed. It is based on an idea which is mentioned for example in [Knobbe, 1999].

The idea

The idea will be explained using an example.

Example The following formulas are equivalent, for a given P:

$$\mathit{goodproduct}(P) \wedge \exists N\, \mathit{price}(P, N) \wedge N < 10 \tag{7.13}$$

$$\mathit{min}\{N | \mathit{goodproduct}(P) \wedge \mathit{price}(P, N)\} < 10 \tag{7.14}$$

If there exists a number lower than a certain threshold, than the minimum of all values must be lower than that threshold, and vice-versa. □

This property can be used to learn thresholds in a similar way as in C4.5. Assume that for some query Q the best split on some variable X has to be determined. For each example, the minimum value of X can be computed by generating all the values for X. This minimum value can be considered as a temporary attribute of the example and therefore the algorithm of C4.5 (see section 6.4), including the MDL punishing mechanism, can be applied to determine the best threshold.

The same procedure can also be used for the $>$ test. In that case all the maximum values have to be collected and the C4.5 method is applied to that set.

Incorporation in Gilder

To incorporate this mechanism in Gilder a second kind of procedure is introduced. Just like the procedures discussed in Definition 5.2.10, these procedures can also be linked to modes.

Definition 7.2.1 *A boolean node procedure is a function*

$$\beta : Node \times T_{c_1} \times \cdots \times T_{c_n} \rightarrow \{true, false\}, \tag{7.15}$$

where $T_{c_i} \in \mathcal{T}$*.*

The interpretation function I and the variable assignment function $\mathcal{B}$ are adapted accordingly, and pass their first argument (which is a pseudo atom, see Definition 5.2.12) also to the boolean procedure.

Because boolean node procedures have a node as parameter, they can use the information which is stored in the node and also change information in the node. This capability is used by the algorithm for continuous attributes. For reasons of convenience, this algorithm will be abbreviated to "ca-algorithm", and the special purpose boolean node procedure will be called the "ca-procedure".

Another special feature which is added for continuous attributes, is a mechanism which translates reals into integers. As the values of the attributes do not matter, but only the order of the values, this can safely be done as long as the order remains the same.

Each node which is associated with a ca-procedure can contain one of these two values:

- a threshold, which stores the threshold value that occurs in the final tree;
- a minimum/maximum value, which is used during the learning procedure.

In the remainder of this section, the mechanism for finding the best '$X <?$' test is discussed. The mechanism for finding the higher-than test is similar.

For each example, a node starts with a minimum value inf. When an example is completely considered, and a new example is begun in the *fill in* procedure, the value in the node is temporarily stored with the example and the value in the node is reinitialized to this value.

If the ca-procedure is called during the learning phase with a value, it compares the value with the minimum value stored in the node and adjusts the mimimum in the node if necessary. It always returns false, in order to keep this node in the *unproved nodes* set of Gilder; this makes Gilder also generate the next value.

When the best node is determined in the *best node* function, the ca-node will have all the examples in its right branch open node, and with each example in that node the associated minimum value is stored. The C4.5 procedure is applied to this set to obtain the best threshold. This value is stored as threshold in the node and the examples in the right branch are redistributed over the left and right branches according to the threshold.

When the ca-procedure is called afterwards for a node and a value, it will compare the value to the threshold in the node and return the correct result accordingly.

7.3 Experimental results

In order to check the efficiency and the accuracy of the mechanisms discussed in this chapter, the different settings of the GILDER algorithm are compared to those of the TILDE algorithm. Accuracy is measured using ten-fold cross validation, unless stated otherwise. The time is expressed in seconds[4] and is the mean of the learning times for each cross validation set.

7.3.1 Mutagenesis

Description

The Mutagenesis database [Srinivasan, 1996] contains descriptions of molecules for which the Mutagenic activity has to be predicted. For the purpose of ILP algorithms, this activity has been converted into two values (Mutagenic or non-Mutagenic), which turns it into a classification task. The dataset consists of 230 molecules, which can be divided into two subsets: regression-friendly (188 molecules) and regression-unfriendly (42 molecules). The term regression here stems from experiments with a linear regression algorithm, which yielded bad results for 42 molecules. Most experiments in literature are only carried out on the regression-friendly molecules. This will also be the case here.

Various algorithms, including TILDE, LINUS and PROGOL, have been tested on this dataset. For testing purposes the dataset is divided into four backgrounds:

- B_1: the atoms in the molecule are given, as well as the bonds between them; the type of each bond is given as well as the element and type of each atom;
- B_2: as B_1, but continuous values about the charge of atoms are added;
- B_3: as B_2, but two continuous values describing each molecule are added;
- B_4: as B_3, but knowledge about the complex structures of the molecule (benzene rings, etc.) are added.

Here also a fifth background will be used, which has not been used before:

- B_5: as B_2, but the attributes of atoms which are reached through bonds might not be considered; any number of bonds is allowed, however.

[4]Experiments were carried out on a Sun Enterprise 450 4x400MHz UltraSPARC2 CPU w/4MB E-cache 4GB RAM.

	PROGOL	FOIL	TILDE	GILDER	RIBL
B_1	76%	61%	75%	81.5%	
B_2	81%	61%	79%	88.5%	
B_3	83%	83%	85%	88.5%	
B_4	88%	82%	86%		95%
B_5			75%	90.0%	

Table 7.1: Results of the application of several algorithms to the Mutagenesis dataset. The results for backgrounds B_1, B_2, B_3 and B_4 in the columns of PROGOL, FOIL and TILDE have been taken from [De Raedt, 1997], the results of RIBL from [Sebag, 1997].

It turned out to be difficult to translate the latter background into a GILDER knowledge base. The background knowledge includes predicate definitions and PROLOG lists, which are not supported by GILDER. Therefore, GILDER has not been tested on B_4. However, the results of other algorithms will be given in order to provide some additional comparison possibilities.

Results

Table 7.1 shows a comparison of the results of several algorithms. The parameters for the GILDER algorithm were the following for B_1, B_2 and B_3:

- only use punishment function $p_1(n)$;
- maximum lookahead depth of 3;
- make a leaf for 2 examples;
- no gain ratio;
- MDL principle for continuous variables (if applicable).

For background B_5 the parameters were the same, but a leaf was already made for 6 examples.

The results in the table show that GILDER outperforms the other algorithms. It is remarkable that even for B_2 a very high accuracy is obtained. Background B_5, which is in fact a subset of B_2, performs even better. A further study of the results, including the shape of tree, would be necessary in order to conclude whether the results of GILDER are also useful for experts, but these first results are promising.

The influence of the lookahead punishment mechanism is investigated in Figure 7.4. It shows that the accuracy improves for all backgrounds when the c in $p_1(n) = c$ is increased. Table 7.2 illustrates the execution times of the algorithm. It appears that the lookahead mechanism also has a strong influence on the induction times. This is understandable as the lookahead mechanism avoids the introduction of new variables, and this in turn restricts the number

c	B_2	B_3
1.0	729	1272
2.0	40.9	40.8
3.0	38.7	38.2
TILDE	60.5	

Table 7.2: A rough comparison of execution times in seconds with lookahead punishment and without it. The punishment function was $p_1(n) = c$ ($p_2(n)$ was not used). These times are rather an indication than reliable execution times. TILDE was tested using a maximum lookahead depth of 3; no PROLOG compiler was available for TILDE, therefore higher speeds are also possible.

of modes which can be used as expansion of the tree. Remarkable is also the comparison with TILDE. Although the implementation of TILDE was certainly not the best, TILDE only needed 60.5 seconds to finish. Considering that this algorithm performs as much lookahead as GILDER with $c = 1.0$, this time is certainly acceptable. A further investigation would be necessary in order to determine whether this difference is only due to the online-discretisation algorithm of GILDER. Another possibility could be the extensive mode mechanism of TILDE. Apart from the mechanism already discussed, it is also possible in TILDE to define beforehand how much evaluations for an atom should be performed at run-time. By giving small values, the algorithm speeds up considerably.

From these results, it is not clear what the influence of the MDL mechanism for continuous variables.

7.3.2 Musk

Description

The Musk database [Dietterich, 1996] contains descriptions of molecules which should be classified as musk or non-musk. For each molecule, there are several feature vectors, each describing a conformation of the molecule. Only one of the conformations is known to be responsible for the classification, but it is not known which. All 166 features have continuous values. The problem is to learn a set of properties which should hold for at least one of the feature vectors in order for a molecule to be classified as positive. In [Dietterich, 1996] this problem was called the multiple instance problem, and it was almost solved using axis parallel rectangles.

Several algorithms, including TILDE, C4.5 and back-propagation, have been tested on this dataset. For testing purposes there are two datasets:

- a small dataset containing 476 feature vectors describing 92 molecules. Each of the vectors only describes a conformation which is known to be important. This is the easy learning task;
- a large dataset containing 6598 feature vectors about 102 molecules.

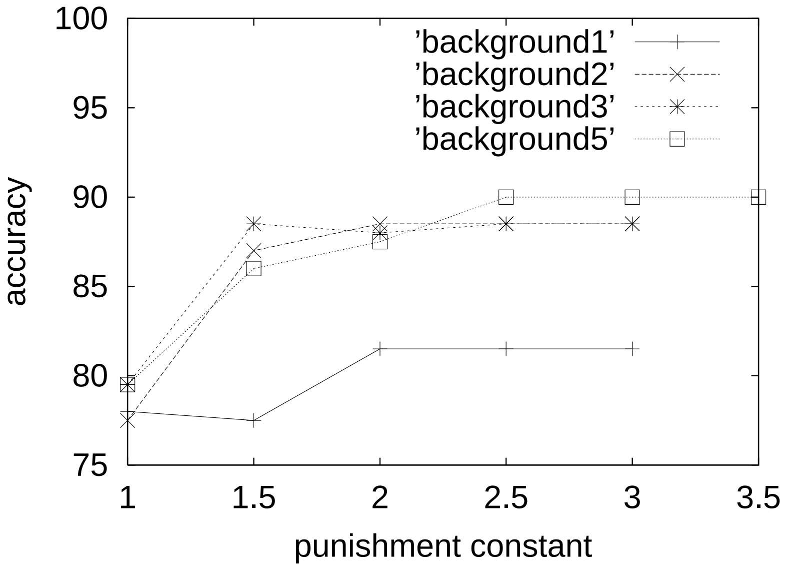


Figure 7.4: The Mutagenesis dataset: the influence of constant c in the punishment formula $p_1(n) = c$.

All algorithms have been tested on both sets.

Results

Table 7.3 and 7.4 show the results of a comparison between several algorithms. The parameters of GILDER which are used in both experiments, are the following:

- no lookahead punishments; this is clear, as the problem only needs a lookahead of depth one in the beginning of the tree building procedure;
- use gain ratio;
- create a leaf when there is only one example left.

Surprisingly, there appears not to be a fixed best setting for treating continuous attributes. In Table 7.5 the results are listed: for the small dataset the application of the MDL principle provides an improvement in accuracy, while the contrary is the case for the larger dataset. It seems reasonable to consider for each dataset separately whether application is necessary or not.

Furthermore, it is remarkable that GILDER performs rather weak on the small dataset (which should be the easiest case), but is slightly better than most other algorithms in the hard case.

Algorithm	Accuracy
iterated-discrim APR*	92.4%
GFS elim-kde APR*	91.3%
GFS elim-count APR	90.2%
Tilde	87.0%
GFS all-positive APR	83.7%
Gilder	81.0%
all-positive APR	80.4%
back-propagation	75.0%
C4.5	68.5%

Table 7.3: Results of the application of several algorithms to the small Musk dataset. The numbers for the other algorithms have been taken from [De Raedt, 1997]. The algorithms with an asterisk are special-purpose APR algorithms.

Algorithm	Accuracy
iterated-discrim APR*	89.2%
Gilder	81.8%
GFS elim-kde APR*	80.4%
Tilde	79.4%
back-propagation	67.7%
C4.5	58.8%

Table 7.4: Results of the application of several algorithms to the large Musk dataset. The numbers for the other algorithms have been taken from [De Raedt, 1997]. The algorithms with an asterisk are special-purpose APR algorithms.

Algorithm	Small dataset	Large dataset
no-MDL Gilder	76.0%	81.8%
MDL Gilder	81.0%	78.2%

Table 7.5: A comparison of the accuracies of Gilder with C4.5 MDL-style punishments for continuous attributes and without them.

It should be noted, however, that the differences are small and that it is questionable whether they are statistically relevant.

7.3.3 Mesh

Description

The mesh database [Dolšak, 1992] contains descriptions of edges in 5 structures. To perform an — at this moment irrelevant — computation on a structure, it is necessary to know what is the optimal[5] number of 'meshes' in the structure. Experts obtain this number by estimating for each edge in the structure the best number of meshes. In order to obtain the latter number, they consider the shape of an edge and its adjacent edges.

The dataset contains for each edge the number of meshes as estimated by an expert, together with a description of the edge and the links to other edges. The learning task is to determine for a given edge the best number of meshes, which is a value between 1 and 17.

Several algorithms have been tested on this database, including TILDE, FOIL and Indigo. Each of these algorithms has been validated using five fold cross validation, where one of the five structures is used for testing and the others for training. This scheme will also be followed here.

Results

Table 7.6 shows a comparison of the results of several algorithms. Of the parameters that were tested, the following were found to be the best:

- punishment functions: $p_1(n) = 2$ and $p_2(n) = 2.5$;
- no gain ratio, but only the basic entropy function;
- create a leaf when the number of examples in a node is less than 7.

As can be seen in the table, the GILDER algorithm performs comparable to the TILDE algorithm. It is remarkable that GILDER is better on all testing sets, except for the last one. Other parameters which were used even obtained a higher accuracy on the first sets, but GILDER ran into difficulties when learning the last one. As the last set is the most important one, GILDER does not improve the TILDE algorithm very much.

As the only relevant difference between GILDER and TILDE is the treatment of continuous variables, the results of this experiment give a good indication of the strength of both mechanisms. The experiments yield the same conclusions as were already obtained for C4.5: run-time discretisation appears to work slightly better, but the difference is not very convincing and it is not clear whether a better result will always be obtained.

[5] Again, for this application it is irrelevant what 'optimal' means.

structure	GILDER	TILDE	FOIL	Indigo 1	Indigo 2	FFOIL	FORS
A	24	22	16	21	21	21	22
B	10	9	9	9	14	15‘	12
C	9	7	8	9	9	11	8
D	17	16	10	41	18	22	16
E	40	45	16	27	33	54	29
Total	100	99	59	107	95	123	87
%	36	36	21	38	34	44	31

Table 7.6: Results of the application of several algorithms to the Mesh dataset. Each number denotes the absolute number of correctly classified edges of the structure; the last row is the percentage of correctly classified edges in the total. The numbers for the other algorithms have been taken from [De Raedt, 1997].

7.3.4 Conclusion

In this section, several experiments have been carried out to investigate the performance and accuracy of GILDER. For both the accuracy and the performance, the results are somewhat ambiguous. For several datasets GILDER succeeded in obtaining a higher accuracy than TILDE, where it is remarkable that GILDER performed better on datasets which are known to be more difficult. The proposed lookahead punishment mechanism was shown to have a lot of positive influence on the accuracy rate. The application of the C4.5 discretisation procedure did however not yield convincingly better results, and for some datasets GILDER could not be tuned to perform better than TILDE at all.

Also from the performance point of view, no convincing results were obtained. Also here, the lookahead punishment function was shown to have a positive influence, but without this mechanism GILDER appears not to execute much faster than TILDE. Possible causes could be that the C4.5 discretisation mechanism is more time consuming, and that the evaluation of the tree is not as time critical in TILDE as it is in WARMR.

7.4 Future work

In this section, some ideas are given for future extensions of GILDER. The purpose of these mechanism is to improve accuracy, understandability and performance.

7.4.1 Post-pruning

The algorithm which has been discussed in this section, misses one important mechanism which is exploited by most decision tree algorithms: post-pruning. The idea is that most decision trees are too specific after that they are constructed, and that some simplification is necessary. After estimating the usefulness of nodes using a sample of the training set some nodes can be removed. Pruning algorithms have been successfully applied to many decision tree algorithms, including TILDE, and it is expected that the same results can also be obtained in GILDER.

7.4.2 More efficient evaluation

Although GILDER uses the tree structure of the search space, it currently has some weaknesses in its evaluation structure, which do not make the algorithm as efficient as it could be. Two important modifications could be made to the algorithm to obtain a higher speed:

- the tree structure is not fully exploited in the lookahead mechanism. The evaluation mechanism of FARMER should be copied more strictly;
- the backtracking mechanism causes sometimes redundant overhead. The following example illustrates this.

  Example The knowledge base consists of the following facts:

$$buys(a, c, 1).$$
$$buys(a, d, 1).$$

  Furthermore, there are the following types and two modes:

$$predicate(buys(Client, Product, Type)).$$
$$predicate(is(Type, Type)).$$
$$mode(1, buys(+, -, -)).$$
$$mode(1, is(+, 1)).$$

  Whether the following formula holds is independent of the product bought:

$$buys(A, B, C), is(C, 1).$$

  However, GILDER will consider *Product* and *Type* together as an output pair. If one output pair can not be proved, it will try the next set. It should be possible to develop a more efficient data structure which proves this query without going through all the values B. □

  As can easily be seen from this example, GILDER (and FARMER) suffer from this problem when there are modes with several outputs, while the database contains facts with the same outputs for the same inputs.

7.5 Summary

In this chapter, a description was given of the GILDER algorithm. GILDER is a first-order logic decision tree algorithm to which some new ideas have been applied in comparison with TILDE. One idea was to obtain a speed-up by removing the need for a PROLOG compiler or interpreter, and by executing prefixes of candidates simultaneously. This target was however not convincingly reached. Another feature which was introduced was that of lookahead punishment. In experiments it was shown that a speed-up as well as an accuracy increase are obtained by using this feature. In the concluding section some ideas were given for further performance and accuracy enhancements.

Chapter 8

An experiment using FARMER and GILDER

In this section a data mining experiment is described which was performed on a dataset of the PKDD'99 challenge. The main purpose is to show the capabilities and incapabilities of FARMER and GILDER in an experiment on a large scale dataset. The consumed computer resources as well as the results are evaluated.

8.1 Overview

The contents of this section is organized as follows. In the first section, the database is introduced, after which previous work on this dataset is discussed in the second section. The third section describes the preprocessing stages which have been performed for all the experiments. The fourth section discusses the results of experiments with FARMER, while the fifth section lists the results of experiments with GILDER. The sixth section summarizes and concludes.

8.2 The bank database

The data set which is used was made available through the "Discovery Challenge" competition, organized as part of the 3rd European Conference on Principles and Practice of Knowledge Discovery in Databases in Prague (September 1999). The data set is based on the data of a real world bank in the Czech Republic. The data has been processed, which means that the current dataset is only a subset of all data, and that no attributes are missing which refer to other tables.

An overview of the database is given in Figure 8.1 and Table 8.1. Contrary to most datasets, it is composed of many tables which have many-to-many relations with each other. This should make this dataset perfectly suitable for the GILDER and FARMER algorithms.

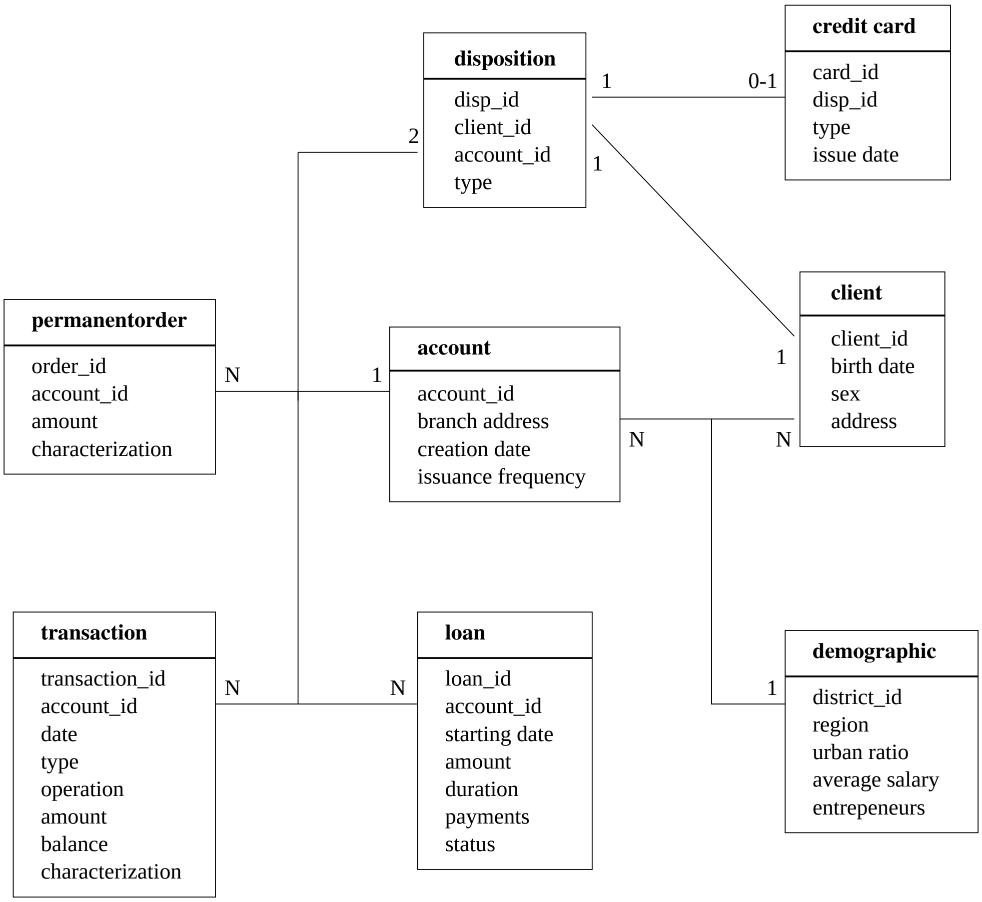


Figure 8.1: A diagram of the bank database.

There was no further background knowledge provided for the database than is given in the database tables. It is not known in which topics the bank is really interested or which concerns they have. Consequently only investigating experiments can be performed.

8.3 Previous work

As part of the Discovery Challenge, several authors have examined the database. The papers which discuss the results of this research have been published in [Berka, 1999]. In this section, a short overview is given of the methods that have been applied there. Results which are relevant for the experiments with Gilder and Farmer will be referenced when discussing those algorithms.

Name	Size	description
account	4500	each record describes static characteristics of an account, such as the location of the branch and the frequency of issuance of statements
client	5369	each record describes characteristics of a client, such as his/her address, birthday and sex
disposition	5369	each record gives the account of a client, and describes whether the client is the owner or only a user
permanent order	6471	each record describes characteristics of a payment order, such as the amount and the reason of the payment (loan, insurance,...), if known
transaction	1056320	each record describes a transaction on an account, and includes information such as the date, the amount, the mode of transaction (credit card, cash, to other bank) and the reason of the payment (household, pension, ...), if specified
loan	682	each record describes a loan granted for an account, and includes information such as the amount and the status of the loan (finished without problems, finished with problems, running with problems, ...)
credit card	892	each record describes a credit card issued to an account, and includes information such as the type of the card (junior, classic, gold)
demographic	77	each record describes demographic characteristics of a district, such as the number of inhabitants, average salary, etc.

Table 8.1: Details about the tables in the bank database.

The authors of the papers have researched three topics:

- loan quality; the target is to determine which clients pay off their loans and which do not. Algorithms which have been applied are GUHA (General Unary Hypothesis Automation), association rules and C5.0[1]. Also an association rule algorithm was used which was adapted for multiple relations (see [Weber, 1999]);
- credit card ownership; the target is to determine which clients may be good credit card owners. Algorithms which have been applied are association rules, univariate deviation detection and k-nearest neighbor;
- other descriptive knowledge, such as properties of regions and transactions. Algorithms which have been applied are association rules on the set of transactions (implementation by [Pijls, 1999]), visualization tools for the separate tables and PROGOL for branch analysis.

Especially the results of the experiments with C5.0, APRIORI (by Pijls) and adapted association rules are interesting with respect to the algorithms discussed here.

In all these articles, the tables have been preprocessed intensively to obtain one loan-oriented, client-oriented or transaction-oriented table. Tables with multiple related records were summarized by computing averages, maximums and minimums.

8.4 Preparation

In this section the preparations are described which were performed on the dataset before it could be used by GILDER and FARMER.

8.4.1 Tools

The dataset consists of several ASCII files (one for each table), which are formatted in a straight-forward way: every line contains one record; the values of the attributes are separated by ';'s. There would be several ways to prepare such files for usage in GILDER and FARMER:

- write an interface to the files of this format by implementing new boolean (output) procedures;
- write a preparation program which translates these ASCII files to the default PROLOG-style input.

In the experiment, the latter mechanism was chosen. A library was written which provides functions that are equivalents for adding predicate definitions, modes and facts. The library is capable of automatically detecting the values of types, sorting floats and outputting all kinds of definitions and facts.

[1] C5.0 is the commercial version of C4.5, and allows for larger datasets.

Furthermore, the output of the library uses an adapted PROLOG-style format, which also allows numbers instead of constant names. GILDER and FARMER were adapted in such a way that they recognize this format easily. In this way, the output of the library is optimized for loading by the algorithms and is an efficient intermediate representation. This makes repeating experiments much more enjoyable, while at the same time the library can also be used in future experiments[2].

It should be noted that an expected disadvantage of the library mechanism is that it depends on the built-in tables of GILDER and FARMER, which are loaded into core memory. Especially large datasets, such as the table of transactions, will not fit into the memory of most computers. This could turn the application of the algorithms in most situations very difficult.

8.4.2 Mechanism

All tables are translated into modes and facts in a similar, straightforward fashion. In the following example the mechanism is illustrated for the loan table.

Example The loan table has the following attributes:

- loan_id;
- account_id;
- starting date;
- amount;
- duration;
- payments;
- status.

This table is translated into facts of the following predicate:

$$predicate(loan(LoanID, AccountID, Year, Month, Amount, Duration, Payments, Status)). \tag{8.1}$$

Furthermore, predicates are added for inspecting the values of attributes which are not IDs (the IDs do not have any meaning themselves, they are just there to distinguish records in a table or to link to a record in another table). The inspection predicates are slightly different for GILDER and FARMER, which was necessary because FARMER can not use continuous values dynamically. For both algorithms, the same kind of predicates are added for inspecting discrete, unordered attributes. For example we consider the Status attribute:

$$predicate(isStatus(Status, Status)). \tag{8.2}$$

[2] In fact, it has already been used (in combination with the basic PROLOG parser of FARMER) in all experiments described in this document to prepare data.

For this predicate, modes are added which link the predicate to an external procedure that compares the first argument with the second. Each mode contains a different constant as the second argument, for example:

$$\begin{gathered} mode(10, isStatus(+, A), \mathbf{extern}(=)). \\ mode(10, isStatus(+, B), \mathbf{extern}(=)). \\ \vdots \end{gathered}$$

For the continuous attributes, the following tests are added in case of GILDER:

$$\begin{gathered} predicate(isYear(Year)). \\ predicate(isMonth(Month)). \\ predicate(isAmount(Amount)). \\ predicate(isDuration(Duration)). \\ predicate(isPayments(Payments)). \\ predicate(isStatus(Status, Status)). \end{gathered}$$

A date is split into a month and a year; the day is currently not put into the facts. Experiments by Pijls indicate however that this could be useful too. Better results therefore may be obtained by doing this in the future. All integer or real valued attributes have a unary predicate to inspect the value. These predicates are linked to the discretisation procedure, which takes care of the boundaries that are put into the tree. For example:

$$mode(10, isYear(+), \mathbf{extern}(\mathbf{discretisation})). \tag{8.3}$$

In case of FARMER, predicates are added for checking whether a value is within a certain interval. This results in predicates like:

$$\begin{gathered} predicate(isAmount(Amount, Amount, Amount)). \\ predicate(isDuration(Duration, Duration, Duration)). \\ \vdots \end{gathered}$$

These predicates are linked to an external procedure which checks whether the first argument is within the boundaries (including the boundaries) of its second and third argument. The modes become:

$$\begin{gathered} mode(10, isAmount(+, 0, 100), \mathbf{extern}(\mathbf{interval})). \\ mode(10, isAmount(+, 100, 1000), \mathbf{extern}(\mathbf{interval})). \\ \vdots \end{gathered}$$

For the FARMER experiments the boundaries are currently determined in an ad-hoc fashion: all values for an attribute are sorted and divided into n subsets, where n is a parameter of the procedure and determines the number of intervals.

The values which bound the intervals obtained in this way, are chosen as the bounds of the discretization. The number of records having a certain value is currently not taken into account, although this might be better practice.

In modes of this kind, the mode numbering mechanism has been used in order to avoid sequential tests for several intervals.

Which mode declarations will furthermore be added, depends on the experiment. Further specifications will therefore be given for each experiment separately. □

8.5 Investigations using Farmer

In this section some experiments with Farmer are discussed. For each experiment, the bias and the results are given. The last section evaluates the experiments altogether.

8.5.1 Transactions

As a descriptive data mining task, it can be interesting to discover some general knowledge about the dependencies of attributes within transactions. This experiment can be considered as an extension of the experiment which was performed by Pijls.

The first section describes the additional specification of the bias which was used by Farmer. The following sections summarize some of the results.

Bias

In this experiment the interest is in the transactions and therefore a new predicate $T(\mathit{TransID})$ is added and defined to be the key predicate. Furthermore, the following modes are added (for reasons of convenience, the types of the parameters are also given):

$$mode(1, transaction(+\mathit{TransID}, \backslash \mathit{AccountID}, -\mathit{Year}, -\mathit{Month}, -\mathit{TransType},$$
$$-\mathit{Amount}, -\mathit{Balance}, -\mathit{TKSymbol}, bank)).$$
$$mode(1, transaction(+\mathit{TransID}, \backslash \mathit{AccountID}, -\mathit{Year}, -\mathit{Month}, -\mathit{TransType},$$
$$-\mathit{Amount}, -\mathit{Balance}, -\mathit{TKSymbol}, cash)).$$
$$mode(1, transaction(+\mathit{TransID}, \backslash \mathit{AccountID}, -\mathit{Year}, -\mathit{Month}, -\mathit{TransType},$$
$$-\mathit{Amount}, -\mathit{Balance}, -\mathit{TKSymbol}, creditcard)).$$
$$mode(2, permanentorder(\backslash \mathit{OrderID}, +\mathit{AccountID}, -\mathit{Amount}, -\mathit{Characterization})).$$
$$mode(1, account(+\mathit{AccountID}, -\mathit{LocationID}, -\mathit{Year}, -\mathit{Month}, -\mathit{Frequency})).$$
$$mode(2, demographic(+\mathit{LocationID}, -\mathit{Region}, -\mathit{UrbanRatio}, -\mathit{Salary}, -\mathit{Entrepeneurs})).$$
$$mode(2, disposition(\backslash \mathit{DispID}, -\mathit{ClientID}, +\mathit{AccountID}, -\mathit{DispType})).$$
$$mode(2, creditcard(\backslash \mathit{CardID}, +\mathit{DispID}, -\mathit{CardType}, -\mathit{Year}, -\mathit{Month})).$$
$$mode(2, client(+\mathit{ClientID}, -\mathit{Year}, -\mathit{Month}, -\mathit{Sex}, -\mathit{LocationID})).$$

$mode(2, loan(\backslash LoanID, +AccountID, -Year, -Month, -Amount, -Duration,$
$-Payments, -Status)).$
$mode(1, isAddress(+LocationID, +LocationID)).$

These modes are a translation of the diagram in Figure 8.1. Several choices have been made by picking this bias:

- the attribute bank, cash or credit card occurs in every rule. Rules without them are not generated. This was necessary in order to be able to use the same datafile in GILDER too. The error in the algorithm which required this has not yet been removed, but has been identified.
- only two different records in the **permanent order** table can be considered at most;
- the circles in the diagram are broken: it is not allowed to look for accounts having the same address as some client, nor is it possible to look for clients having the same address as an account. As a result, it is neither possible to look for dispositions of clients with the same address.
- every ID is an output variable with a backslash to prevent a record from being considered which has already been looked at. The exception is the *LocationID*, because it is considered to be useful information that the same address might be obtained in several ways. To make this also explicit in the rules, a predicate *isAddress* is added to compare the addresses found.

Results 1

With a minimum support of 50% and minimum interest function $f(x) = 10x$, the following 7 interesting queries are found in 1727 seconds:

1. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isAddress(+3.0, +3.0), isFreq(+4.0, POPLATEK_MESICNE), district(+3.0, \ 19.0, \ 20.0, \ 21.0, \ 22.0), isYear(+1.1, 83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1), isAddress(+3.1, +3.1) (0.616169)

2. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isAddress(+3.0, +3.0), isFreq(+4.0, POPLATEK_MESICNE), isYear(+1.1, 83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1), isAddress(+3.1, +3.1), district(+3.1, \ 19.0, \ 20.0, \ 21.0, \ 22.0) (0.616169)

3. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isAddress(+3.0, +3.0), district(+3.0, \ 19.0, \ 20.0, \ 21.0, \ 22.0), isYear(+1.1, 83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1), isAddress(+3.1, +3.0) (0.573067)

4. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isAddress(+3.0, +3.0), district(+3.0, \ 19.0, \ 20.0, \ 21.0, \ 22.0), isYear(+1.1, 83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1), isAddress(+3.0, +3.1) (0.573067)

5. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isAddress(+3.0, +3.0), isYear(+1.1, 83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1), isAddress(+3.1, +3.0), district(+3.1, \ 19.0, \ 20.0, \ 21.0, \ 22.0) (0.573067)

6. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isAddress(+3.0, +3.0), isYear(+1.1, 83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1), isAddress(+3.0, +3.1), district(+3.1, \ 19.0, \ 20.0, \ 21.0, \ 22.0) (0.573067)

7. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.1, -2.1, -3.0), isAddress(+3.0, +3.0), district(+3.0, \ 19.0, \ 20.0, \ 21.0, \ 22.0) (0.677572)

This small example set will be used to demonstrate some of the current shortcomings of FARMER, and illustrates the idea for interesting queries that was introduced in Section 5.4.4.

As a first step, the set can be pruned manually by removing redundant atoms, such as *isAddress*(3.0, 3.0).

```
1. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isFreq(+4.0,
   POPLATEK_MESICNE), isYear(+1.1, 83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER) (0.616169)

2. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isYear(+1.1,
   83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1),
   isAddress(+3.1, +3.0) (0.573067)

3. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), account(+0.0, -1.1, -2.1, -3.0, -4.0), isYear(+1.1,
   83.000000, 97.000000), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0, OWNER), client(+5.0, -6.0, -1.2, -2.2, -3.1),
   isAddress(+3.0, +3.1) (0.573067)

4. T(-23.0), tcash(+23.0, \ 0.0, \ 1.0, \ 2.0, \ 24.0, \ 11.0, \ 25.0, \ 26.0), disposition(\ 7.0, -5.0, +0.0, -8.0), isDispType(+8.0,
   OWNER), (0.677572)
```

This set can be reduced further by removing the last two queries: the third query means the same as the second, because the predicate *isAddress* is cumutative, and the third is not interesting with respect to the manually obtained second query. As every account has an owner, also these specifications can be dropped.

The results of this experiment thus can be summarized as:

1. 61.2% of the transactions:

   - is in cash;
   - is performed on behalf of an account which receives monthly issuances and is granted in the period 1993–1997.

2. 57.3% of the transactions:

   - is in cash;
   - is performed on behalf of an account which belongs to a branch in the same district as the district of the client owning the account.

As 67% of the transactions is in cash and 92.7% of the accounts receives monthly issuances and is granted in the period 1993–1997, it could have been expected that 62.1% of the transactions were of the first kind. As the difference with 61.2% is not very large, this query probably is not interesting too. An algorithm such as discussed in Section 5.4.4 could have filtered this out. The same conclusion can be drawn for the second query.

The only remaining conclusion is that most transactions are carried out in cash. A similar conclusion was also drawn from the experiment performed by Pijls and others.

Results 2

The experiment is repeated for a minimum support of 10%. To prune as much queries as possible, only the longest queries are listed. After 32 hours and 25 minutes the computation finished with 990 interesting queries. Memory usage was, by approximation, 200MB, while processor utilization was between 20% and 25%. A sample of the conclusions which can be drawn from this set of queries is the following:

- only the districts South Moravia and North Moravia generate more than 10% of the transactions;
- heavily urbanized districts with high average salaries generate more than 10% of the transactions, where it is remarkable that most of the accounts associated with these transactions have permanent household orders;
- most of the transactions are generated by accounts having a disponent born between 1956 and 1965; clients born between 1974 and 1983 generate just enough transaction to satisfy the support criterion;
- 12% of the transactions is generated by an account for which there is a disponent and an owner living in the same district. Also 12% of the transactions is generated by an account for which there is a male and a female disponent living in the same district. A combination of both (one the owner, the other the disponent), does not satisfy the criterion, which could be an indication that ownership is reasonably well spread between the sexes;
- transactions for accounts which receive monthly issuances, and are in a district with the entrepeneurs attribute within the range [117, 131], have not enough permanent household orders to reach the 10%, while this criterion is reached easily for regions out of this range.

The amount of results of this experiment is quite large, and also here the results contain a lot of redundant queries. Although it took some time to finish the experiment, this time is still reasonable taking into account that there are one million transactions.

The initial results are not very well comparable to those obtained by [Pijls, 1999b] as they are sorted for length and not for support. Inspection of all queries however yields that all sets which were discovered by [Pijls, 1999b] are also here obtained, except for those queries which are day related or contain explicit information about withdrawals and credits. As was already mentioned, this information was currently not put into the bias of the search.

8.5.2 Clients

In this experiment, FARMER is applied to discover relations between the properties of clients. The bias is discussed in the first section, the results are given in the second section.

Bias

The bias which is used, is almost the same as the bias discussed in Section 8.5.1, with the following two modifications:

- as key a new predicate $P(-ClientID)$ is used;
- the mode declaration for the **disposition** relation is changed and becomes

$$mode(1, disposition(\backslash DispID, +ClientID, \backslash AccountID, -DispType)).$$

It can easily be checked from Figure 8.1 that these changes are necessary when the focus moves to the clients.

Results 1

FARMER was executed using a minimum support of 10% and minimum interest function $f(x) = 10x$. In 658 seconds it discovered 1147 queries which complied to the support and interest criterion. Some interesting rules which have been extracted, are:

- `P(-5.0), disposition(\ 7.0, +5.0, \ 0.0, -8.0), isDispType(+8.0, OWNER), permanentorder(\ 9.0, +0.0, -10.0, -11.0, -12.0), isKSymbol(+12.0, SIPO), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 38.000000, 47.000000) (0.115084)`
- `P(-5.0), disposition(\ 7.0, +5.0, \ 0.0, -8.0), isDispType(+8.0, OWNER), permanentorder(\ 9.0, +0.0, -10.0, -11.0, -12.0), isKSymbol(+12.0, SIPO), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 47.000000, 56.000000) (0.108752)`
- `P(-5.0), disposition(\ 7.0, +5.0, \ 0.0, -8.0), isDispType(+8.0, OWNER), permanentorder(\ 9.0, +0.0, -10.0, -11.0, -12.0), isKSymbol(+12.0, SIPO), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 56.000000, 65.000000) (0.110987)`
- `P(-5.0), disposition(\ 7.0, +5.0, \ 0.0, -8.0), isDispType(+8.0, OWNER), permanentorder(\ 9.0, +0.0, -10.0, -11.0, -12.0), isKSymbol(+12.0, SIPO), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 65.000000, 74.000000) (0.110987)`

These rules have been picked out randomly and are simplified manually, and may be compared to the following more general rules. The numbers between brackets after the queries are, at position one, the support of the query, and at position two, the fraction of this support in comparison with the corresponding query above):

- $P(-5.0), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 38.000000, 47.000000)(0.182343, 0.63114021)$
- $P(-5.0), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 47.000000, 56.000000)(0.174893, 0.6218202)$
- $P(-5.0), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 56.000000, 65.000000)(0.181598, 0.61116863)$
- $P(-5.0), client(+5.0, -6.0, -1.0, -2.0, -3.0), isYear(+1.0, 65.000000, 74.000000)(0.175824, 0.63123919)$

It may be concluded that of the people born before 1965, older people are more often the owner of an account and have monthly household orders. It is important to note however that the absolute difference is only 20 clients.

Also here, the amount of results is currently too large to determine which results are really interesting. The importance of this experiment is therefore that it also shows that FARMER can build the set of frequent queries reasonably fast.

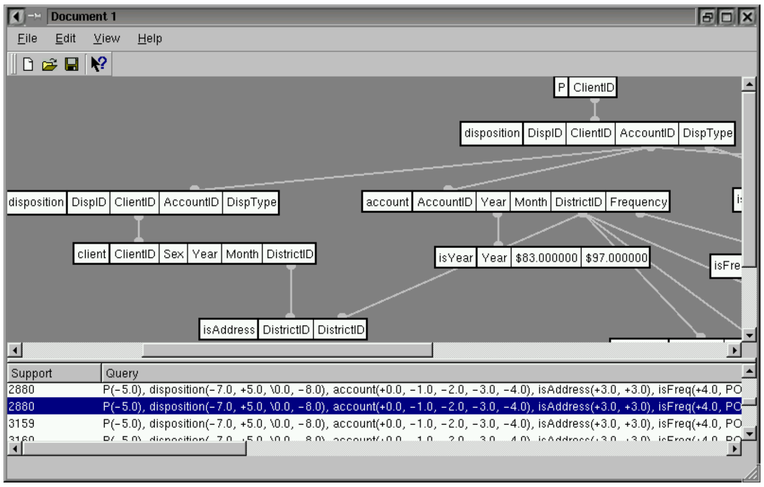


Figure 8.2: A screen of VisualFarmer. The top part displays the query which is highlighted at the bottom of the screen.

Results 2

FARMER was also executed using a minimum support of 50% and an adapted bias which included the following mode:

$$mode(1, disposition(\backslash DispID, \backslash ClientID, +AccountID, -DispType)).$$

This mode allows for the inspection of the other clients which have the disposition over the clients's account. In Figure 8.2 a screen of the visualization tool for FARMER is shown. It shows a graphical representation of the query which states that the other client lives in the same district as the address where the account was established.

8.5.3 Conclusion

The experiments make clear that the FARMER algorithm is a reasonably fast data mining tool on average scale applications, but that the results it produces are not yet usable most of the time. Some other postprocessing algorithms, such as the procedure discussed in Section 5.4.4, are necessary to prune the results even further and make them valuable.

FARMER did discover all rules which were also obtained by APRIORI, with the exception of some rules which were due to the restricted bias that was used. Some new knowledge was also discovered, which was illustrated using rules about districts and relations between clients.

8.6 Predictions using GILDER

In this section some predictive experiments with GILDER are described. For each experiment, first the bias is discussed and then the results. The section concludes with some final remarks.

8.6.1 Prediction of loan status

For the bank it can be interesting to know which clients pay off their loan and which do not. It would allow the bank to adjust their acceptance policy. GILDER has been applied to build a decision tree for this.

Bias

The examples in this case are the records in the loan table, so the predicate **loan** is appointed as the key and a new attribute 'basic status' is assigned as the target. This new attribute is obtained by merging 'finished good loan' and 'running good loan' into a new value 'good loan' and by merging 'finished bad loan' and 'running bad loan' into a new value 'bad loan'.

For the other tables, the following modes are added:

$$mode(2, permanentorder(+OrderID, -AccountID, -Amount, -Characterization)).$$
$$mode(1, account(+AccountID, -LocationID, -Year, -Month, -Frequency)).$$
$$mode(3, demographic(+LocationID, -Region, -UrbanRatio, -Salary, -Entrepeneurs)).$$
$$mode(2, disposition(-DispID, -ClientID, +AccountID, -DispType)).$$
$$mode(2, creditcard(-CardID, +DispID, -CardType, -Year, -Month)).$$
$$mode(2, client(+ClientID, -Year, -Month, -Sex, -LocationID)).$$

These modes have been obtained in a similar fashion as in Section 8.5.1. Note that backslashes have not been used here.

Results

Table 8.2 shows the distribution of good and bad loans. As can be seen in this table, the absolute number of bad loans is not very large. Nevertheless, some experiments are done to build a decision tree, and cross validations are computed. In the best experiment, a ten-fold cross validation accuracy of 84% is achieved. Some of the decision trees in this experiment reach over 90% accuracy, but others only 24%. Inspection of the decision trees also yields that most decisions are eventually taken on a weak basis. Many labels are given because there is only one client more of one type than of the other type. Probably, the decision trees achieved in this experiment are therefore not a very good predictive instrument.

The tests which are chosen in the top of the tree for most of the training sets, might give an indication about important attributes however. These tests are the following:

- is there a second disponent for the account? Loans for such accounts are good in 100% of the available cases, for other accounts only in 85.8% of the cases;

- is there a permanent household order for the account? Loans for such accounts are good in 95.5% of the cases, for other accounts only in 76.7% of the cases;

- is there a male disponent? Although never chosen as first test, almost all decision trees chose this test. The loans which satisfy this test are more often a good loan. Taking into account that many transactions have a male and a female disponent (which is a result of the descriptive experiment with FARMER), this could be an indication that single male disponents pay off their loans better than single female disponents;

- in some of the decision trees a check for a credit card occurs, but not in all and certainly not in top of the tree.

As the currently generated decision trees are too large with too many unbenificial splits, it is likely that a pruning mechanism could improve the accuracy of GILDER considerably.

The experiments which were performed using C5.0 gave some similar, but also some different results. C5.0 was also applied several times, which had as purpose the determination of the important attributes. As attributes for C5.0 finally were chosen 'multi access', minimum balance before the creation date of the loan and sum of amounts (incoming minus outgoing). It is of course reasonable that both latter attributes should have some importance. GILDER does not currently exploit any information from the transaction table. A straigthforward addition to the bias could be this:

$$mode(1, transaction(transaction(\backslash TransID, +AccountID, +Year, +Month,$$
$$-TransType, -Amount, -Balance, -TKSymbol, -Type)).$$
$$mode(1, previousmonth(+Year, +Month, -Year, -Month), generatemonth).$$

The latter mode is linked to an external procedure which, for a given year and month, returns a set of previous months (which of course also have a year). The transaction table is consulted for transactions in these previous months. The results should be similar and tests should be generated for the balances and amounts.

The importance of household orders was not shown in experiments with C5.0, probably because there were no attributes for permanent orders. Experiments with association rules however also revealed the importance which was found here.

The conclusions which were drawn for single males and credit cards were not listed for any other experiment. GILDER also determined that this knowledge is not very important.

type	absolute	relative
good loan	606	88.8%
bad loan	76	11.2%

Table 8.2: The fraction of good and bad loans in the set of loans.

8.6.2 Prediction of mode of transaction

Each transaction in the database has a *mode*[3]: a field which tells whether the transaction was performed using a credit card, was a remittance to another bank, and so on. In this experiment GILDER is applied to characterize credit card transactions, bank transactions and transactions in cash. Banks do not like cash transactions as this obliges them to keep cash on hand. On the other hand, they appreciate credit card transactions as they receive higher profits from them. If the bank had a characterization of the transaction modes, this would allow them to address the causes which lead to certain modes.

The first section described the bias which was used by GILDER. The next sections summarize some of the results.

Bias

The examples in this case are the transactions in the transaction table, so the predicate **transaction** is appointed as the key and a new attribute 'mode' is assigned as the target. This new attribute is obtained by merging 'credit in cash' and 'withdrawal in cash' into a new value 'cash' and by merging 'collection from other bank' and 'remittance to another bank' into a new value 'bank'.

For the other tables, the following modes are added (for reasons of convenience, the types of the parameters are also given):

$$mode(2, permanentorder(+OrderID, -AccountID, -Amount, -Characterization)).$$
$$mode(1, account(+AccountID, -LocationID, -Year, -Month, -Frequency)).$$
$$mode(2, demographic(+LocationID, -Region, -UrbanRatio, -Salary, -Entrepeneurs)).$$
$$mode(2, disposition(-DispID, -ClientID, +AccountID, -DispType)).$$
$$mode(2, creditcard(-CardID, +DispID, -CardType, -Year, -Month)).$$
$$mode(2, client(+ClientID, -Year, -Month, -Sex, -LocationID)).$$
$$mode(2, loan(-LoanID, +AccountID, -Year, -Month, -Amount, -Duration,$$
$$-Payments, -Status)).$$

These modes are a translation of the diagram in Figure 8.1, and have been obtained in a similar way as in Section 8.5.1. Also here the circle has been broken. All transactions are considered however.

[3]Not to be confused with mode declarations.

Results 1

In Appendix B.1, a decision tree is listed for this classification problem. Parameters are:

- make a leaf for 10000 examples;
- lookahead punishment functions $p_1(n) = 1.3$ and $p_2(n) = 0.0006$;
- no gain ratio;
- MDL principle for continuous attributes.

The constants of the punishment functions are not taken to be very large, as in this case some fine-tuning may be interesting. Gain ratio is not used as it results in trees which even more resemble decision lists. The MDL principle has shown its value, although it is still remarkable that the *Amount* continuous attribute is often chosen. Slightly other parameters have also been tested for the punishment functions and the leaf threshold, for which sometimes even higher accuracies were obtained. The resulting trees were however more complex and were only variations of this tree, so this tree is discussed here as a representative.

GILDER consumed 3703 seconds to build the decision tree. The classification rate on the training set is 95.74%. Memory consumption is 470MB. A sample of 10%[4] of the total dataset has been created; this sample is used for a cross validation experiment with the same parameters[5]. The cross validation accuracy is 93.3%. No other experiments are known for this classification task, so this number can only be compared with the densities of the three class labels, which are given in Table 8.3.

For all experiments, the first split of the decision tree algorithm is the same ("is this a household transaction?"), which seems reasonable as household transactions are often in cash. Also the second test ("is there a permanent household order?") is very reasonable, as this checks for the one exception to household transactions payed in cash. It is strange, however, that all household transactions on accounts which have permanent household orders, are indeed bank transactions. It would have been likely that still other cash transactions remain for such accounts. This gives rise to the more fundamental question how the type of transaction (e.g. "household") is determined, and whether this attribute can be trusted. Other reasons for suspicion may be transactions with unspecified types or types denoted with an underscore. The experiment was therefore repeated without modes to check for these attributes; a summary of the results of this experiment is given later in the next section.

From this experiment, the following conclusions may be drawn furthermore:

- old-age pensions are payed by credit card in all cases (!);

[4]The sample is obtained by randomly choosing one transaction from 10 consecutive transactions, starting from the beginning of the file. This should be sufficient as the only order of the transactions seems to be their order in time.

[5]In fact, the parameters were determined first in such smaller experiments.

Mode	Fraction
credit card	7.47%
cash	67.76%
bank	24.77%

Table 8.3: The fractions of the bank, cash and credit card transactions in the total set of transactions.

- loan payments are bank transactions;
- the unspecified transactions are split on the amount in unexpected ways, for example amounts between 2332 and 2899 appear to be payed in cash in 94% of the cases, while amounts between 1920 and 1998 are bank transactions in 484 of the 574 cases (which is 84%).
- the existence of loans is frequently checked, and always is an indication that the transaction is in cash;
- the amount of loans is often used as check, and high amounts are always an indication of credit card transactions;
- in an attempt to split credit card from cash transactions for some transactions, GILDER determines that testing on the month December is the best current test. GILDER does not succeed in finding a good next split (which could be an indication that it is the month itself that is important), and finally chooses 'cash' as label. Indeed, the number of credit card transactions in that node is also relatively lower (6.1% against 13.8% in other months).

Finally, it is remarkable that GILDER does *not* check whether there is a credit card for the account. Although this could be caused by lookahead punishments, it is at least an indication that this attribute is not very distinctive at all.

Results 2

In this experiment the type of transaction has not been considered and left out of the bias. Furthermore the more elaborate punishment function is used, although still with small punishment constants. In 4912 seconds the decision tree in Appendix B.2 is constructed. A tenfold crossvalidation was run on a subset of the transactions to validate the parameters, which resulted in an accuracy of 85.1%. There are several remarkable properties of the decision tree:

- although the MDL principle is applied, the tree contains many tests for the amount of the transaction, which results in a partitioning of the amounts in bank payments and cash payments. A remarkable conclusion is for example that payments lower than 14 are performed by bank transaction most of the time, while all 144092 ($\approx$ 14%) payments between 14.0 and 14.9 are performed in cash;

- important facts are the existence of loans and permanent orders, which in most of the cases are indications for bank transactions. The height of the loan and the order is tested frequently too;
- the age of the client is important, a year of birth before 1939 being an indication for a credit card transaction;
- the tree is impractically large, which could be an indication that GILDER is fine-tuning too much.

Also here, it could be beneficial to apply a decision tree pruning algorithm.

8.6.3 Conclusion

The experiments have shown that GILDER can very well be applied to large datasets and that the results are understandable. Most of the decision trees are however quite large, which makes a pruning mechanism for GILDER also desirable.

In the loan prediction experiments, rules were found which are similar to those discovered in other experiments. However, some rules were also missing here due to a restricted bias.

8.7 Summary

From the experiments in this chapter several conclusions may be drawn, for the algorithms separately, but also for the general ideas which lay behind them.

It has been shown that the mode mechanism can easily be used to specify a search structure for an ordinary database. The technique was based on the distinction between two types of predicates: predicates which refer to tables and predicates which test attribute values. The capability to define external procedures for modes has shown its value for the latter predicates. Furthermore, the mode mechanism clearly formalized which relations in a database will be considered and which will not.

On the other hand, it also appears that not all possibilities of the mode mechanism may be necessary. Apart from one single exception, all modes had exactly one input. Special purpose algorithms for such biases may be developed to take advantage of this situation.

The results of both algorithms were spoiled by their size. While FARMER generated too many queries, the trees of GILDER were too deep. For both algorithms, an illustration was given of a procedure to solve this, but they have not yet been implemented.

The results of the experiments were similar to those found in earlier experiments, although some results were missing due to the restricted biases that were used in some experiments.

The main purpose of the experiments was to show that the algorithms could cope with large datasets. Whether this target was reached is not sure. The longest experiment which was performed took one and a half day to process one million transactions. In comparison with similar ILP systems, this is impressive. The algorithms however loaded all data in core memory, which required a total amount of 450MB main memory. A massive computer therefore still is obligatory.

Chapter 9

Conclusion

In this thesis, the possibilities for combining logic with traditional algorithms were investigated. The answer to this question could be summarized with

> yes, there are possibilities of efficiently combining them.

We prefer however not to judge at this moment whether the algorithms in this Thesis are the perfect proof of this. In a few moments, it will become clear why.

As a starting point for the investigation two algorithms were taken: WARMR and TILDE. The idea existed that these algorithms already incorporated many features from traditional algorithms (association rules and decision trees, to be precise), but that these algorithms missed some essential properties to become 'killer apps'. It was argued that the lack of efficiency was probably their most important deficiency.

To tackle this problem, two new algorithms were built. Both of them borrow many ideas from WARMR and TILDE, but are a complete reimplementation, based on some basic ideas:

- not use PROLOG, but use only a small subset which is executable by a tiny procedure;
- obtain a faster evaluation strategy by putting the queries in a tree datastructure;
- use a less complex equivalency mechanism than the one based on θ-subsumption.

The algorithms were implemented and tested. As the experiments yield a speed-up and an enhanced accuracy in some cases, one could conclude that the targets were reached.

However, the world does not stand still, and other research has also proceeded. Only during the project, and after some important decisions were already made and some texts were already written, we became aware of related ongoing work of other researchers. One could argue that maybe better results can be obtained by incorporating their work thoroughly. At this moment, we have only investigated the possiblities of adding some of these ideas in the framework we had developed already, but currently we are convinced that for some purposes parts of the framework itself may better be adapted. We will give a short list of other work here, and will make clear how some of the ideas are incorporated currently:

- in [Knobbe, 1999] the possibilities are investigated of not using logic as representation mechanism. Instead, the formalism of *selection graphs* is used. It is shown that these graphs can be converted into PROLOG and SQL queries. Especially the latter feature has a huge advantage in practical cases: candidates can easily be evaluated by any RDBMS[1], and even efficiently by specialized RDBMSs. In this way, the results of database research are exploited to obtain a better performance. Furthermore, the selection graphs are less expressive than logic, and consequently some isomorphisms are far more easily checked, although a price has to be payed for this loss of generality.

  The idea of selection graphs is especially reflected in the bias which was used to investigate the financial dataset. The selection graph procedure offers three building blocks, which are reflected in the predicates that give attributes, check attributes, and link two variables. The idea may also be found in the restricted bias that we introduced in Section 5.4.4.

- in [Blockeel, 2000] an evaluation strategy is discussed for a set of PROLOG queries which is called *packed query evaluation*. A closer look at this strategy reveils that it is based on a similar tree as the one used in GILDER and FARMER. This can be seen as a confirmation that the evaluation strategies of GILDER and FARMER are sensible.

- in [Weber, 1999] a second restricted first-order logic mechanism for association rules is introduced, which seems to be based on one of the conclusions that was also drawn in this thesis: only those queries which have for each predicate at most one occurence, are reasonably efficiently compared.

- in personal correspondence with Luc de Raedt, I learned that the isomorphism problem that startled me for so long, is indeed a hot topic which has not been solved convincingly until now.

We thus conclude that, although the original goals have been reached, some interesting research may not have been taken into account sufficiently. However, this thesis provides a good insight in the problems which occur in *multi relational data mining*, and it shows that there are possibilities to solve them efficiently and practically.

[1] Relational Data Base Management System

Appendix A

Some thoughts on CLP and ILP

This Appendix contains some short thoughts about the usability of *Constraint Logic Programming* in the non-monotonic setting of ILP. As it could not very well be fitted in the course of the main text, it is given here.

A.1 Overview

This Appendix is organized as follows: in the first section a short introduction to *Constraint Logic Programming* is given. The second section discusses its applicability in WARMR-like algorithms.

A.2 Constraint Logic Programming

Most ILP systems are based on PROLOG, a programming language which is based on Horn clauses. Mathematical computations are not a part of this formalism; thus, to allow computations in this language, ad-hoc solutions are used: for example, the special purpose infix predicate `is` gives an unbound variable on its left hand the value which is expressed on its right hand. Consider this clause:

$$\mathit{sum\_higher}(X_1, X_2, X_3) \leftarrow \mathit{Temp\ is}\ X_1 + X_2, \mathit{Temp} > X3, \tag{A.1}$$

which defines that *sum_higher* is true if $X_1 + X_2$ is higher than X_3. The *is* predicate gives a value to the *Temp* variable just like an assignment in an imperative programming language. This clause is excellent if X_1, X_2 and X_3 have a value. However, this means that it is unusable in the following clause:

$$p(X_1, X_2) \leftarrow \mathit{sum\_higher}(X_1, Y, X_2), \tag{A.2}$$

which states that $p(X_1, X_2)$ is true if there exists a number Y such that $X_1 + Y > X_2$. Although this might seem trivial, PROLOG will be unable to prove $p(1, 0)$. In order to prove $p(1, 0)$, the system has to prove that $sum\_higher(1, X_2, 0)$ is true while X_2 is not given a value yet. The *is* predicate can not compute the value at its right hand, so the prove procedure will abort.

PROLOG is unable to answer this question as it is completely unaware of the rules of mathematics. It only has some built-in functions to perform computations. Some extension of PROLOG is necessary to solve the example in a general way. The development of logical systems which are aware of (some) mathematics is called *Constraint Logic Programming.*

In CLP the definition of the *sum_higher* predicate would look like

$$sum\_higher(X_1, X_2, X_3) \leftarrow X_1 + X_2 > X_3. \tag{A.3}$$

In order to prove $sum\_higher(1, Y, 0)$ an X_2 has to be found such that $1+X_2 > 0$. The mathematical system will return that the equation is true for $X_2 > -1$ and X_2 will be bound to this interval. As there exist values for X_2 for which *sum_higher* is true, the CLP system also succeeds in proving $p(1, 0)$.

A.3 CLP in non-monotonic ILP

In non-monotonic ILP dependencies are found which are true on a sufficient number of examples. Existing ILP systems are capable of discovering such knowledge which uses predicates only. By adding computations and comparisons to the rules, new interesting knowledge can be discovered. In a supermarket, the following WARMR-like query could now be discovered:

$$\begin{gathered} livesat(X, Y), housevalue(Y) \geq 1000000, \\ buys(X, Z_1), type(Z_1, wine), buys(X, Z_2), type(Z_2, fish), \\ price(Z_1) + price(Z_2) \geq 100. \end{gathered} \tag{A.4}$$

An ILP system which can discover such rules would clearly be a powerful tool in knowledge discovery.

However, is it necessary to use all the possibilities of a CLP system? Although computations are implemented ad-hoc in PROLOG, the validity of the example query can also be proved by PROLOG, as Y, Z_1 and Z_2 are bound to values by $housevalue(Y)$, $buys(X, Z_1)$ and $buys(X, Z_2)$, respectively, before they are used in a computation or comparison. In general, a variable which is not bound to a value yet, can be forced to a value by using a predicate which defines the domain of values to be tested. As long as the domain is finite, this mechanism can be applied. Only when the domain is infinite, as for example the domain of real numbers, this mechanism is not executable as it is impossible to test all the values in the domain. In many applications a finite domain is sufficient, so a CLP system is not necessary.

Of course, this does not turn CLP useless. It is still necessary to use CLP when the domain is infinite or very large. The following query can not be proved in Prolog if X and Y are unbound, real-typed variables:

$$3 * X + 5 * Y = 10, 2 * X + Y = 4, X > Y, \tag{A.5}$$

as it is impossible to iteratively test all values of X and Y.

Another application of CLP could be found in the implementation of the ILP system. In the discussion above, a rather traditional approach to non-monotonic ILP is assumed: first queries are created, next those queries are verified on the examples. In a system which also allows comparisons and computations as part of the rules, this strict two-phase mechanism will not work: to determine the interesting values in comparisons (for example 100 in $price(Z_1) + price(Z_2) \geq 100$), some algorithm is necessary which uses the examples. A CLP system might play a role in the implementation of this algorithm.

A.4 Summary

In this Appendix some thoughts were presented concerning the combination of ILP with CLP. It was illustrated that there may be some possibilities for the application of CLP in non-monotonic ILP, but it was also discussed that in a lot of situations CLP is not necessary.

Appendix B

Bank database results

This Appendix lists the decision trees which were produced by GILDER for the experiments in Chapter 8, and could be used to verify some of the conclusions which are drawn there.

B.1 Decision tree of Section 8.6.2, results 1

The numbers after the atoms give the numbers of examples which go to the left and right branch, respectively, for each of the classes.

```
isTKSymbol( +22.0, SIPO ) < 0 2811 115254 < >  65226 588850 101065 >?
 Yes: permanentorder( +0.0, \9.0, \10.1, \11.0 ) < 0 1079 115254 < >  0 1732 0 >?
 | Yes: isKSymbol( +11.0, SIPO ) < 0 0 115254 < >  0 1079 0 >?
 | | Yes: tbank
 | | No:  tcash
 | No:  tcash
 No:  isTKSymbol( +22.0, _ ) < 0 616 52817 < >  65226 588234 48248 >?
   Yes: tbank
   No:  isTKSymbol( +22.0, DUCHOD ) < 30338 0 0 < >  34888 588234 48248 >?
     Yes: tcreditcard
     No:  isTKSymbol( +22.0, POJISTNE ) < 0 23 18477 < >  34888 588211 29771 >?
       Yes: tbank
       No:  isTKSymbol( +22.0, UVER ) < 0 0 13580 < >  34888 588211 16191 >?
         Yes: tbank
         No:  isAmount:$2899.000000=>( +10.0 ) < 0 283212 13737 < >  34888 304999 2454 >?
           Yes: isTKSymbol( +22.0, UNSPEC ) < 0 125803 13737 < >  0 157409 0 >?
           | Yes: isAmount:$89.000000<( +10.0 ) < 0 125803 13382 < >  0 0 355 >?
           | | Yes: isAmount:$1920.000000<( +10.0 ) < 0 33638 5574 < >  0 92165 7808 >?
           | | | Yes: isAmount:$2332.000000<( +10.0 ) < 0 18560 1178 < >  0 15078 4396 >?
           | | | | Yes: tcash
           | | | | No:  isAmount:$1998.000000<( +10.0 ) < 0 14988 3912 < >  0 90 484 >?
           | | | |   Yes: tcash
           | | | |   No:  tbank
           | | | No:  tcash
           | | No:  tbank
           | No:  tcash
           No:  isAmount:$3993.000000=>( +10.0 ) < 17493 29660 2062 < >  17395 275339 392 >?
             Yes: loan( +0.0, \1.1, \2.1, \10.1, \13.0, \14.0, \15.0 ) < 0 5963 326 < >  17493 23697 1736 >?
             | Yes: tcash
             | No:  permanentorder( +0.0, \9.0, \10.1, \11.0 ) < 17373 19373 1345 < >  120 4324 391 >?
             |   Yes: isAmount:$2659.000000<( +10.1 ) < 0 13947 1185 < >  17373 5426 160 >?
             |   | Yes: tcash
             |   | No:  isKSymbol( +11.0, LEASING ) < 0 576 41 < >  17373 4850 119 >?
             |   |   Yes: tcash
             |   |   No:  isAmount:$3499.000000<( +10.0 ) < 2830 1996 49 < >  14543 2854 70 >?
             |   |     Yes: tcreditcard
             |   |     No:  isAmount:$2332.000000<( +10.1 ) < 0 638 12 < >  14543 2216 58 >?
             |   |       Yes: tcash
             |   |       No:  tcreditcard
             |   No:  tcash
             No:  isAmount:$24801.000000=>( +10.0 ) < 6334 230350 392 < >  11061 44989 0 >?
               Yes: isAmount:$10029.000000<( +10.0 ) < 6334 129085 0 < >  0 101265 392 >?
               | Yes: loan( +0.0, \1.1, \2.1, \10.1, \13.0, \14.0, \15.0 ) < 3034 26132 0 < >  3300 102953 0 >?
               | | Yes: tcash
```

```
| | No:  isAmount:$10030.000000<( +10.0 ) < 3249 102951 0 < >  51 2 0 >?
| |   Yes: tcash
| |   No:  tcreditcard
| No:  tcash
No:  isAmount:$64900.000000<( +10.0 ) < 540 91 0 < >  10521 44898 0 >?
  Yes: tcreditcard
  No:  loan( +0.0, \1.1, \2.1, \10.1, \13.0, \14.0, \15.0 ) < 4873 13652 0 < >  5648 31246 0 >?
    Yes: tcash
    No:  isAmount:$43492.000000<( +10.0 ) < 2049 6280 0 < >  3599 24966 0 >?
      Yes: tcash
      No:  isMonth:$11.000000<( +2.0 ) < 284 4771 0 < >  3315 20195 0 >?
        Yes: tcash
        No:  isAmount:$24802.000000=>( +10.0 ) < 74 0 0 < >  3241 20195 0 >?
          Yes: tcreditcard
          No:  isAmount:$42630.000000=>( +10.0 ) < 3237 19661 0 < >  4 534 0 >?
            Yes: isAmount:$26258.000000<( +10.0 ) < 2609 17227 0 < >  628 2434 0 >?
            | Yes: isAmount:$29878.000000<( +10.0 ) < 2227 12312 0 < >  382 4915 0 >?
            |   Yes: isAmount:$29880.000000<( +10.0 ) < 2173 12311 0 < >  54 1 0 >?
            |   | Yes: tcash
            |   | No:  tcreditcard
            |   No:  tcash
            | No:  tcash
            No:  tcash
```

B.2 Decision tree of Section 8.6.2, results 2

The numbers after the atoms give the numbers of examples which go to the left and right branch, respectively, for each of the classes.

```
isAmount:$14.900000=>( +11.0 ) < 0 145073 1855 < >  65226 446588 214464 >?
  Yes: isAmount:$14.000000<( +11.0 ) < 0 144092 0 < >  0 981 1855 >?
    Yes: tcash
    No:  tbank
  No:  isAmount:$9985.000000=>( +11.0 ) < 47831 270872 210174 < >  17395 175716 4290 >?
    Yes: isAmount:$2899.000000<( +11.0 ) < 47831 132002 91312 < >  0 138870 118862 >?
    | Yes: isAmount:$6998.000000<( +11.0 ) < 0 40574 18465 < >  47831 91428 72847 >?
    | | Yes: permanentorder( -9.0, +0.0, \10.0, \11.1, \12.0 ) < 0 32822 18453 < >  0 7752 12 >?
    | | | Yes: isAmount:$6994.000000<( +11.1 ) < 0 8132 18418 < >  0 24690 35 >?
    | | | | Yes: isAmount:$9985.000000=>( +11.1 ) < 0 6594 18418 < >  0 1538 0 >?
    | | | | | Yes: isKSymbol( +12.0, UVER ) < 0 2086 1852 < >  0 4508 16566 >?
    | | | | | | Yes: tcash
    | | | | | | No:  isAmount:$7000.000000<( +11.0 ) < 0 4382 16564 < >  0 126 2 >?
    | | | | | |   Yes: isAmount:$7097.000000<( +11.0 ) < 0 4350 15513 < >  0 32 1051 >?
    | | | | | |   | Yes: isAmount:$7097.000000<( +11.1 ) < 0 4093 15513 < >  0 257 0 >?
    | | | | | |   | | Yes: isAmount:$7100.000000<( +11.0 ) < 0 3986 15513 < >  0 107 0 >?
    | | | | | |   | | | Yes: tbank
    | | | | | |   | | | No:  tcash
    | | | | | |   | | No:  tcash
    | | | | | |   | No:  tbank
    | | | | | |   No:  tcash
    | | | | | No:  tcash
    | | | | No:  tcash
    | | | No:  tcash
    | | No:  loan( -15.0, +0.0, \1.1, \2.1, \11.1, \16.0, \17.0, \18.0 ) < 0 20718 17672 < >  47831 70710 55175 >?
    | |   Yes: isMonth:$1.000000<( +2.0 ) < 0 16894 16356 < >  0 3824 1316 >?
    | |   | Yes: isPayments:$7240.000000<( +17.0 ) < 0 2166 904 < >  0 14728 15452 >?
    | |   | | Yes: tcash
    | |   | | No:  isAmount:$140880.000000<( +11.1 ) < 0 5603 7848 < >  0 9125 7604 >?
    | |   | |   Yes: isYear:$97.000000<( +1.1 ) < 0 987 663 < >  0 4616 7185 >?
    | |   | |   | Yes: tcash
    | |   | |   | No:  isYear:$94.000000<( +1.0 ) < 0 3950 6725 < >  0 666 460 >?
    | |   | |   |   Yes: isAmount:$3000.000000<( +11.0 ) < 0 3761 6678 < >  0 189 47 >?
    | |   | |   |   | Yes: isPayments:$2646.000000<( +17.0 ) < 0 3683 6678 < >  0 78 0 >?
    | |   | |   |   | | Yes: tbank
    | |   | |   |   | | No:  tcash
    | |   | |   |   | No:  tcash
    | |   | |   |   No:  tcash
    | |   | |   No:  isYear:$94.000000<( +1.1 ) < 0 5735 5779 < >  0 3390 1825 >?
    | |   | |     Yes: isAmount:$4780.000000<( +11.0 ) < 0 2726 2013 < >  0 3009 3766 >?
    | |   | |     | Yes: tcash
    | |   | |     | No:  tbank
    | |   | |     No:  tcash
    | |   | No:  tcash
    | |   No:  permanentorder( -9.0, +0.0, \10.0, \11.1, \12.0 ) < 47531 55733 54710 < >  300 14977 465 >?
    | |     Yes: isAmount:$2897.000000<( +11.1 ) < 11302 39516 54480 < >  36229 16217 230 >?
    | |     | Yes: isAmount:$3991.000000<( +11.1 ) < 0 26100 30732 < >  11302 13416 23748 >?
    | |     | | Yes: isAmount:$6992.000000=>( +11.1 ) < 0 17614 30249 < >  0 8486 483 >?
    | |     | | | Yes: isAmount:$4000.000000<( +11.0 ) < 0 11479 27816 < >  0 6135 2433 >?
    | |     | | | | Yes: isKSymbol( +12.0, LEASING ) < 0 696 260 < >  0 10783 27556 >?
    | |     | | | | | Yes: tcash
    | |     | | | | | No:  isAmount:$4075.000000<( +11.0 ) < 0 10777 26410 < >  0 6 1146 >?
    | |     | | | | |   Yes: isAmount:$4075.000000<( +11.1 ) < 0 10222 26410 < >  0 555 0 >?
    | |     | | | | |   | Yes: isKSymbol( +12.0, UVER ) < 0 111 0 < >  0 10111 26410 >?
    | |     | | | | |   | | Yes: tcash
```

```
| |     | | | | |   | | No:  tbank
| |     | | | | |   | No:  tcash
| |     | | | | |   No:  tbank
| |     | | | | No:  tcash
| |     | | | No:  tcash
| |     | | No:  isAmount:$3991.000000<( +11.0 ) < 11302 8992 40 < >  0 4424 23708 >?
| |     | |   Yes: disposition( \7.0, \5.0, +0.0, \8.0 ) < 11302 8992 40 < >  0 0 0 >?
| |     | |   | Yes: client( +5.0, \6.0, \1.1, \2.1, \3.0 ) < 11302 8992 40 < >  0 0 0 >?
| |     | |   | | Yes: isYear:$39.000000=>( +1.1 ) < 11302 1451 0 < >  0 7541 40 >?
| |     | |   | | | Yes: tcreditcard
| |     | |   | | | No:  tcash
| |     | |   | | No:  tcash
| |     | |   | No:  tcash
| |     | |   No:  isMonth:$1.000000<( +2.0 ) < 0 3478 21979 < >  0 946 1729 >?
| |     | |     Yes: isAmount:$2900.000000<( +11.0 ) < 0 3297 21924 < >  0 181 55 >?
| |     | |     | Yes: isAmount:$2998.000000<( +11.0 ) < 0 3228 19512 < >  0 69 2412 >?
| |     | |     | | Yes: isAmount:$2998.000000<( +11.1 ) < 0 2828 19500 < >  0 400 12 >?
| |     | |     | | | Yes: isKSymbol( +12.0, LEASING ) < 0 93 69 < >  0 2735 19431 >?
| |     | |     | | | | Yes: tcash
| |     | |     | | | | No:  tbank
| |     | |     | | | No:  tcash
| |     | |     | | No:  tbank
| |     | |     | No:  tcash
| |     | |     No:  tbank
| |     | No:  isKSymbol( +12.0, LEASING ) < 0 2374 44 < >  36229 13843 186 >?
| |     |   Yes: tcash
| |     |   No:  isKSymbol( +12.0, _ ) < 7307 6447 145 < >  28922 7396 41 >?
| |     |     Yes: isAmount:$3993.000000<( +11.0 ) < 0 4592 18 < >  7307 1855 127 >?
| |     |     | Yes: tcash
| |     |     | No:  tcreditcard
| |     |     No:  isAmount:$4210.000000<( +11.0 ) < 17047 2699 3 < >  11875 4697 38 >?
| |     |       Yes: disposition( \7.0, \5.0, +0.0, \8.0 ) < 17047 2699 3 < >  0 0 0 >?
| |     |       | Yes: client( +5.0, \6.0, \1.1, \2.1, \3.0 ) < 17047 2699 3 < >  0 0 0 >?
| |     |       | | Yes: isYear:$39.000000=>( +1.1 ) < 17047 1844 3 < >  0 855 0 >?
| |     |       | | | Yes: tcreditcard
| |     |       | | | No:  tcash
| |     |       | | No:  tcash
| |     |       | No:  tcash
| |     |       No:  isAmount:$1931.000000=>( +11.1 ) < 413 1587 22 < >  11462 3110 16 >?
| |     |         Yes: tcash
| |     |         No:  isAmount:$2361.000000<( +11.1 ) < 2376 1793 12 < >  9086 1317 4 >?
| |     |           Yes: tcreditcard
| |     |           No:  disposition( \7.0, \5.0, +0.0, \8.0 ) < 9086 1317 4 < >  0 0 0 >?
| |     |             Yes: client( +5.0, \6.0, \1.1, \2.1, \3.0 ) < 9086 1317 4 < >  0 0 0 >?
| |     |             | Yes: isYear:$34.000000<( +1.1 ) < 8972 533 2 < >  114 784 2 >?
| |     |             | | Yes: tcreditcard
| |     |             | | No:  tcash
| |     |             | No:  tcash
| |     |             No:  tcash
| |     No:  tcash
| No:  permanentorder( -9.0, +0.0, \10.0, \11.1, \12.0 ) < 0 109457 117365 < >  0 29413 1497 >?
|   Yes: isAmount:$2898.000000=>( +11.1 ) < 0 69373 115381 < >  0 40084 1984 >?
|   | Yes: isMonth:$1.000000<( +2.0 ) < 0 47109 106522 < >  0 22264 8859 >?
|   | | Yes: isKSymbol( +12.0, _ ) < 0 18444 65395 < >  0 28665 41127 >?
|   | | | Yes: isAmount:$30.000000<( +11.0 ) < 0 17124 64212 < >  0 1320 1183 >?
|   | | | | Yes: isAmount:$99.000000<( +11.0 ) < 0 17107 59374 < >  0 17 4838 >?
|   | | | | | Yes: isAmount:$100.000000<( +11.0 ) < 0 16310 59295 < >  0 797 79 >?
|   | | | | | | Yes: isAmount:$199.000000<( +11.0 ) < 0 16151 54848 < >  0 159 4447 >?
|   | | | | | | | Yes: isAmount:$200.000000<( +11.0 ) < 0 15749 54839 < >  0 402 9 >?
|   | | | | | | | | Yes: tbank
|   | | | | | | | | No:  tcash
|   | | | | | | | No:  tbank
|   | | | | | | No:  tcash
|   | | | | | No:  tbank
|   | | | | No:  tcash
|   | | | No:  isAmount:$1500.000000<( +11.0 ) < 0 10151 28468 < >  0 18514 12659 >?
|   | | |   Yes: isAmount:$1500.000000<( +11.1 ) < 0 7751 28316 < >  0 2400 152 >?
|   | | |   | Yes: isKSymbol( +12.0, SIPO ) < 0 5524 26109 < >  0 2227 2207 >?
|   | | |   | | Yes: isAmount:$1920.000000<( +11.0 ) < 0 3670 21445 < >  0 1854 4664 >?
|   | | |   | | | Yes: isAmount:$1920.000000<( +11.1 ) < 0 2839 21405 < >  0 831 40 >?
|   | | |   | | | | Yes: isAmount:$1999.000000<( +11.0 ) < 0 2836 19406 < >  0 3 1999 >?
|   | | |   | | | | | Yes: isAmount:$2000.000000<( +11.0 ) < 0 2558 19401 < >  0 278 5 >?
|   | | |   | | | | | | Yes: tbank
|   | | |   | | | | | | No:  tcash
|   | | |   | | | | | No:  tbank
|   | | |   | | | | No:  tcash
|   | | |   | | | No:  tbank
|   | | |   | | No:  tcash
|   | | |   | No:  tcash
|   | | |   No:  isAmount:$1500.000000=>( +11.1 ) < 0 5151 12522 < >  0 13363 137 >?
|   | | |     Yes: isKSymbol( +12.0, LEASING ) < 0 1188 377 < >  0 3963 12145 >?
|   | | |     | Yes: tcash
|   | | |     | No:  isAmount:$30.000000<( +11.0 ) < 0 3619 11973 < >  0 344 172 >?
|   | | |     |   Yes: isKSymbol( +12.0, UVER ) < 0 899 1226 < >  0 2720 10747 >?
|   | | |     |   | Yes: tbank
|   | | |     |   | No:  isMonth:$11.000000<( +2.0 ) < 0 696 1126 < >  0 2024 9621 >?
|   | | |     |   |   Yes: tbank
|   | | |     |   |   No:  isBalance:3110=>( +25.0 ) < 0 318 444 < >  0 1706 9177 >?
|   | | |     |   |     Yes: tbank
|   | | |     |   |     No:  isAmount:$29.000000<( +11.1 ) < 0 1650 9177 < >  0 56 0 >?
|   | | |     |   |       Yes: tbank
```

```
|   | | |     |   |       No:  tcash
|   | | |     |   No:  tcash
|   | | |     No:  tcash
|   | | No:  isKSymbol( +12.0, _ ) < 0 8692 5476 < >  0 13572 3383 >?
|   | |   Yes: isAmount:$178.000000<( +11.0 ) < 0 8453 4730 < >  0 239 746 >?
|   | |   | Yes: tcash
|   | |   | No:  tbank
|   | |   No:  tcash
|   | No:  tcash
|   No:  tcash
No:  isAmount:$14891.000000=>( +11.0 ) < 1774 53338 4290 < >  15621 122378 0 >?
  Yes: permanentorder( -9.0, +0.0, \10.0, \11.1, \12.0 ) < 1318 43385 4290 < >  456 9953 0 >?
  | Yes: isAmount:$10056.000000<( +11.1 ) < 86 1962 4290 < >  1232 41423 0 >?
  | | Yes: tbank
  | | No:  tcash
  | No:  tcash
  No:  isAmount:$25122.000000=>( +11.0 ) < 4698 78538 0 < >  10923 43840 0 >?
    Yes: tcash
    No:  isAmount:$64900.000000<( +11.0 ) < 540 91 0 < >  10383 43749 0 >?
      Yes: tcreditcard
      No:  loan( -15.0, +0.0, \1.1, \2.1, \11.1, \16.0, \17.0, \18.0 ) < 4858 13386 0 < >  5525 30363 0 >?
        Yes: tcash
        No:  isAmount:$43492.000000<( +11.0 ) < 2049 6280 0 < >  3476 24083 0 >?
          Yes: tcash
          No:  isMonth:$11.000000<( +2.0 ) < 278 4625 0 < >  3198 19458 0 >?
            Yes: tcash
            No:  isAmount:$25378.000000=>( +11.0 ) < 244 353 0 < >  2954 19105 0 >?
              Yes: tcash
              No:  isAmount:$42630.000000=>( +11.0 ) < 2950 18571 0 < >  4 534 0 >?
                Yes: isAmount:$29878.000000<( +11.0 ) < 2227 12312 0 < >  723 6259 0 >?
                | Yes: isAmount:$29880.000000<( +11.0 ) < 2173 12311 0 < >  54 1 0 >?
                | | Yes: tcash
                | | No:  tcreditcard
                | No:  tcash
                No:  tcash
```

Bibliography

[1] R. Agrawal, H. Manilla, R. Srikant, H. Toivonen, A.I. Verkamo. Fast Discovery of Association Rules. In: U.M. Fayyad et al. (eds.). *Advances in Knowledge Discovery and Datamining.* AAAAI/MIT Press. 1996.

[2] P. Berka. *Workshop Notes on Discovery Challenge PKDD–99.* Prague, 1999.

[3] H. Blockeel. *Top-down Induction of First Order Logical Decision Trees.* PhD Thesis. Katholieke Universiteit Leuven, 1998.

[4] H. Blockeel, L. De Raedt. *Top-down Induction of Logical Decision Trees.* 1997.

[5] H. Blockeel. *Top-down Induction of Logical DEcision Trees User's Manual.* 1997.

[6] H. Blockeel *et al. Executing Query Packs in ILP.* 2000.

[7] M. Bongard. *Pattern Recognition.* Spartan Books, 1970.

[8] L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone. *Classification and regression trees.* 1984.

[9] T. Connolly, C. Begg, A. Strachan. *Database Systems: A Practicle Approach to Design Implementation and Management.* Addison-Wesley, 1995.

[10] L. Dehaspe, W. Van Laer, L. De Raedt. CLAUDIEN*: The Clausal Discovery Engine User's Guide.* 1996.

[11] L. Dehaspe, L. De Raedt. *Mining Association Rules in Multiple Relations.* 1997.

[12] L. Dehaspe, M. Forrier. *Transformation-based Learning meets Frequent Pattern Discovery.* 1997.

[13] L. Dehaspe, H. Toivonen. *Frequent query discovery: a unifying ILP approach to association rule mining.* 1998.

[14] L. Dehaspe. *Frequent Pattern Discovery in First Order Logic.* PhD Thesis. Katholieke Universiteit Leuven. 1998.

[15] L. Dehaspe, H. Toivonen, R.D. King. *Finding frequent substructures in chemical compounds.* 1998.

[16] T.G. Dietterich, R.H. Lathrop, T. Lozano-Pérez. *Solving the Multiple-Instance Problem with Axis-Parallel Rectangles.* 1997.

[17] B. Dolšak, S. Muggleton. The application of inductive logic programming to finite-element mesh design In: *Inductive Logic Programming*, vol. 38 of APIC Series. Academic Press, London, 1992.

[18] U.M. Fayyad, K.B. Irani. On the handling of continuous-valued attributes in decision tree generation. In: *Machine Learning, 8*, 87–102. 1992.

[19] U.M. Fayyad *et al. Datamining and Knowledge Discovery.* 2(1):5–7. 1998.

[20] U.M. Fayyad *et al. Advances in Knowledge Discovery and Datamining.* AAAI/MIT Press, 1996.

[21] J.M. de Graaf, W.A. Kosters, J.J.W. Witteman. Interesting Association Rules in Multiple Taxonomies. In *Proceedings Twelfth Belgium/Netherlands Artificial Intelligence Conference* (BNAIC00). 2000.

[22] A. Karalič. *First Order Regression.* PhD Thesis. University of Ljubljana, 1995.

[23] J.–U. Kietz, M. Lübbe. An Efficient Subsumption Algorithm for Inductive Logic Programming. In *Proceedings of the 11th International Conference on Machine Learning.* Morgan Kaumann, 1994.

[24] A. Knobbe, H. Blockeel, A.P.J.M. Siebes, D.M.G. van der Wallen. *Multi-Relational Data Mining.* Internal report. CWI, 1999.

[25] J. Köbler, U. Schöning, J. Torán. *The Graph Isomorphism Problem.* Birkhauser Boston, 1997.

[26] R. Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the Internation Conference on Artificial Intelligence.* 1995.

[27] J.N. Kok, W.A. Kosters. Natural Data Mining Techniques. In *Bulletin of the EATCS 71*, pp. 133–142. 2000.

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

[29] B. Liu, W. Hsu, Y. Ma. Mining Association Rules with Multiple Minimum Supports. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD–99).* 1999.

[30] C.J. Mertz, P. Murphy. *UCI repository of machine learning databases.* `http://www.cs.uci.edu/~mlearn/MLRepository.html`.

[31] P. Mikšovský, Filip Železný, Olga Štěpánková, Michal Pěchouček. Financial Data Challenge. In *Workshop notes on Discovery Challenge PKDD-99*, Laboratory of Intelligent Systems, University of Economics. Prague, September 1999.

[32] T.M. Mitchell, M. Hill. *Machine Learning*. McGraw–Hill, 1997.

[33] S. Muggleton, J. Firth. *CProgol4.4: a tutorial introduction*

[34] S. Muggleton. Inverse entailment and Progol. In *New Generation Computing*, 13:245–286. 1995.

[35] S-H. Nienhuys-Cheng, R. de Wolf. *Foundations of Inductive Logic Programming*, LNAI 1228. Springer Verlag, 1997.

[36] W. Pijls, J.C. Bioch. Mining Frequent Itemsets in Memory-Resident Databases. In E. Postma et al. (eds) *Proceedings Eleventh Belgium/Netherlands Artificial Intelligence Conference* (BNAIC99), pp. 75–82. 1999.

[37] W. Pijls. *Discovery Challenge, Financial Data*. In *Workshop notes on Discovery Challenge PKDD-99*, Laboratory of Intelligent Systems, University of Economics. Prague, September 1999.

[38] G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, 5, pp. 153–163. 1970.

[39] R. Potharst. *Classification using Decision Trees and Neural Nets*. PhD Thesis. Erasmus Universiteit Rotterdam, 1999.

[40] P. van der Putten. *A Data Mining Scenario for Stimulating Credit Card Usage by Mining Transaction Data*. Proceedings of Benelearn'99, 1999.

[41] P. van der Putten. *Promoting Credit Card Usage by Mining Transaction Data*. In *Workshop notes on Discovery Challenge PKDD-99*, Laboratory of Intelligent Systems, University of Economics. Prague, September 1999.

[42] P. van der Putten. *Data Fusion for Data Mining: A Problem Statement*. Internal Report. Sentient Machine Research, 2000.

[43] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[44] J.R. Quinlan. *FOIL: A Midterm Report*. Internal report. Basser Department of Computer Science, University of Sydney, 1993.

[45] J.R. Quinlan. Improved Use of Continuous Attributes in C4.5. In *Journal of Artificial Intelligence Research 4*, Morgan Kaufmann. 1996.

[46] J.R. Quinlan. Learning First-Order Definitions of Functions. In *Journal of Artificial Intelligence Reasearch 5*:139–161. 1996.

[47] L. De Raedt, H. Blockeel, L. Dehaspe, W. Van Laer. *Three Companions for First Order Datamining.*

[48] L. De Raedt, L. Dehaspe. *Clausal Discovery* . 1996.

[49] S. Reeves, M. Clarke. *Logic for Computer Science.* Addison Wesley, 1990.

[50] J. Rissanen. A universal prior for integers and estimation by minimum description length. In *Information and Computation 80.* 1983.

[51] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995.

[52] M. Sebag. Distance Induction in First Order Logic. In *Proceedings of the 7th International Workshop on Inductive Logic Programming (ILP–97)*, LNAI vol.1297. Springer-Verlag, 1997.

[53] A. Srinivasan, R.D. King. Feature construction with ILP: a study of quantitative predictions of biological activity by structural attributes. In *Proceedings of the Sixth International Workshop on Inductive Logic Programming*, LNAI vol. 1314. Springer-Verlag, 1996.

[54] I. Weber. A declarative language bias for levelwise search of first–order regularities. In L. Bing, W. Hsu, W. Ke (eds.). *Proceedings ISMIS'99*, LNAI 1609. Springer-Verlag, 1999.

[55] D. Wood. *Theory of Computation.* John Wiley, 1987.