KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

COMPLEX AGGREGATES IN RELATIONAL LEARNING

Promotor:
Prof. Dr. ir. H. BLOCKEEL

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Celine VENS

Maart 2007

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

COMPLEX AGGREGATES IN RELATIONAL LEARNING

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Celine VENS

UDC 681.3∗I26

Maart 2007

D/2007/7515/26
ISBN 978–90–5682–792–2

Beknopte samenvatting

Relationele leertechnieken leren patronen uit relationele gegevensbanken, die gewoonlijk uit meerdere tabellen bestaan, die met elkaar gerelateerd zijn. Deze relaties kunnen bijvoorbeeld een één-op-veel of veel-op-veel cardinaliteitsverhouding hebben. Een voorbeeld waarvoor een predictie gemaakt moet worden kan dus gerelateerd zijn aan een verzameling objecten die mogelijk relevant zijn voor de predictie. Bestaande relationele leermethoden behandelen deze verzamelingen op één van volgende manieren: door het opleggen van condities aan de elementen in de verzameling of door het gebruik van aggregaatsfuncties om ze samen te vatten. Bestaande methoden zijn niet in staat om beide benaderingen te combineren, waardoor ze bepaalde patronen niet kunnen leren. Het belangrijkste doel van dit eindwerk is het combineren van beide benaderingen, dus het aggregeren over een deelverzameling van elementen die aan een specifieke selectieconditie voldoen.

Deze combinatie van aggregaten en selecties brengt verscheidene moeilijkheden met zich mee. Ten eerste wordt de zoekruimte substantieel uitgebreid en ten tweede is de algemeen-naar-specifiek ordening van de hypothesen, die verondersteld wordt door veel relationele leersystemen, geschonden. Dit impliceert dat men, gebruik makende van klassieke verfijningsoperatoren, ofwel efficiëntie ofwel volledigheid moet opgeven bij het doorzoeken van de hypotheseruimte. In dit werk ontwikkelen we een algemeen bruikbaar verfijningsraamwerk dat de volledige zoekruimte beschouwt en deze in een algemeen-naar-specifieke, en dus efficiënte, manier doorloopt.

Complexe aggregaten worden ingebouwd in een bestaand relationeel leersysteem dat beslissingsbomen construeert. We argumenteren dat de algemeenheidsordening van de zoekruimte in deze context niet geschonden kan worden, en dat bijgevolg klassieke verfijningsoperatoren gebruikt kunnen worden. Om de efficiëntie te verhogen worden twee technieken voorgesteld: een toepassing van het voorgestelde verfijningsraamwerk en een opwaardering van het relationeel beslissingsboomleeralgoritme naar een systeem dat relationele gerandomiseerde bossen leert.

Het gebruik van complexe aggregaten wordt ook bestudeerd in het consequent van een hypothese. Meerbepaald onderzoeken we het gebruik van complexe aggregaten in de lineaire modellen die gebouwd worden in de bladeren van relationele modelbomen. Daarvoor wordt het relationele beslissingsboomleeralgoritme uitgebreid om modelbomen te leren. De belangrijkste contributie hierbij is het ontwikkelen van een efficiënte heuristiekfunctie die geschikt is voor het leren van modelbomen.

Tenslotte wordt het gebruik van complexe aggregaten geëvalueerd in twee toepassingen.

Abstract

In relational learning one learns patterns from relational databases, which usually contain multiple tables that are interconnected via relations. These relations may be of one-to-many or many-to-many cardinality ratios. Thus, an example for which a prediction is to be given may be related to a set of objects that are possibly relevant for that prediction. Relational classifiers differ with respect to how they handle these sets: some use properties of the set as a whole (using aggregation), some refer to properties of specific individuals of the set, however, most classifiers do not combine both. This imposes an undesirable bias on these learners. This dissertation describes a learning approach that avoids this bias, by using complex aggregates, i.e., aggregates that impose selection conditions on the set to aggregate on.

This combination of aggregates and selections presents several difficulties. First, the search space is substantially increased, and second, the generality order of the hypotheses that is assumed by many relational learners is violated. This implies that one either has to give up on efficiency or on completeness when searching the hypothesis space using classical refinement operators. We develop a general refinement framework that considers the complete search space, and traverses it in a general-to-specific, hence efficient, way.

Complex aggregates are included in an existing relational learner that constructs relational decision trees. We argue that in this context, the generality ordering can not be violated, and classical refinement operators can be applied. To improve efficiency, we present two techniques: an application of the developed refinement framework and an upgrade of the relational decision tree algorithm to a relational random forest inducer.

The use of complex aggregates is also studied in the consequent of a hypothesis. More precisely, we investigate the use of complex aggregates in the linear models built by a relational model tree learner. This involves upgrading the relational decision tree algorithm to a relational model tree learning system. The main contribution in this work is the development of an efficient heuristic function suitable for learning model trees.

Finally, the use of complex aggregates is evaluated in two real life applications.

Acknowledgements

I would like to express my gratitude to all people who have made this great Ph.D. adventure possible.

First and foremost, I would like to thank my supervisor prof. Hendrik Blockeel. I am grateful for the opportunity he gave me to start a Ph.D. in the machine learning group. I very much appreciate his knowledge and enthusiasm about machine learning. Whenever the results were less encouraging, a discussion with Hendrik gave me a new energy boost to go on. I also thank him for proofreading this text over and over again and for suggesting many improvements. I am still amazed about his punctuality to provide feedback on papers or on any other technical text. I have enjoyed many great experiences with Hendrik, also after work. I remember a walk to a very charming tea house in the middle of nowhere at Lake Louise in Canada (with an over-enthusiastic taxi driver to take us there), and a safari-trip together with other colleagues in South-Africa.

I am grateful to prof. Maurice Bruynooghe and prof. Dirk Roose for being in my doctoral committee and for providing valuable comments on this text. I especially thank Maurice, who was also my supervisor at the start of my Ph.D. and who was always there to advise me. I would like to thank prof. Sašo Džeroski, prof. Stefan Kramer, and dr. Arno Knobbe for accepting to be members of my jury and prof. Van Brussel for chairing the jury. A special word of thanks goes to Sašo. I will never forget our first meeting. It was at ECML 2003 and he told me to pay close attention to Leo Breiman's invited talk about random forests, because "he had an idea". When I wanted to discuss this with him afterwards, he told me to come and find him later for a swim in the sea. So, our first discussion actually took place in the Adriatic Sea, at the coast of Cavtat, Croatia. Since then he has been like an extra supervisor to me. During his frequent visits to Leuven, he always made time to discuss research, and came up with many great ideas.

A dissertation is usually not made by only one person. Therefore, I would like to thank all my co-authors without whom many parts of this text would not have been possible: Aneta Ivanovska, Anneleen Van Assche, Hendrik Blockeel, Jan Ramon, Jan Struyf, Sašo Džeroski, Tom Croonenborghs, and Werner Uwents.

I gratefully acknowledge the financial support of the Fund for Scientific Research of Flanders (FWO Vlaanderen) and the GOA on Inductive Knowledge Bases.

The people that are responsible for a fantastic work climate are my colleagues from the machine learning group. Since lately the group expands faster than I can get to know everyone, I can not thank them all. However, a few people deserve a special word of thanks. First there is Anneleen, who started working at the machine learning group together with me and has been a great colleague ever since. We have had many enjoying collaborations and also had fun in less work related things, such as in discovering Europe and the rest of the world. Then there is Stefan, who, together with Anneleen, has been my office mate for the last 4.5 years. I especially thank him for the great salsa lessons he teaches every week, which have been a welcome distraction for me the last few months. Then there are Joaquin and Werner, who make up the rest of the 03.46 team and contribute to the great atmosphere in the office. The two people without whom this Ph.D. would have taken many more years to get finished are the two Jans. I thank them for patiently answering my endless stream of ACE-related questions and for providing feedback or suggesting interesting ideas for my research.

I am grateful to my friends in Leuven and at home for the many enjoying evenings we have, far away from complex aggregates and ILP clauses.

I would like to express my deepest gratitude to my mom and dad, for their love and support, for their encouragement, and for making my weekends at home a pleasure to look forward to. I also thank Ares's parents Dit and Luc, and Eris and David (who would have thought that my master's thesis companion would become my brother-in-law?) for their support and interest in my work.

I saved the last "thank you" for the most important person in my life. I would like to thank him especially for his support during the last half year. I am very lucky to have a partner who did most of the housekeeping, helped me with technical details about LaTeX, listened to my complaints, and dragged me away from my desk every now and then, all this while being confronted with writing his own Ph.D. Thank you, Ares.

Celine Vens
Heverlee, March 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we present an overview of this work and motivate it within the context of several scientific fields. The work is mainly situated in the fields of relational data mining and inductive logic programming. We begin this dissertation by giving an introduction to these domains, starting from the more general field of artificial intelligence, and passing through its sub-domain of machine learning.

1.1 Context

1.1.1 Artificial intelligence

For the general public, artificial intelligence (AI) is most probably known from some of its successful applications, such as the *Deep Blue* chess computer, or from science fiction movies, such as *Bicentennial Man* (1999) or more recently *I Robot* (2004), where robots show intelligent behaviour. However, AI has more applications, which are present in every-day life. For example, e-mail spam filters, the behaviour of characters in video games, the personalized results returned by Internet search engines, or the handwriting recognition component of personal digital assistants (PDA's) all involve AI. As omnipresent as AI may be, it is hard to define the concept exactly, which is illustrated by the fact that definitions abound. Therefore, we provide the intuitive definition given by John McCarthy, considered as the father of AI, as an answer on the question *"What is AI?"* in the frequently asked questions section of his website[1].

Definition 1.1 (Artificial Intelligence) *[Artificial intelligence] is the science and engineering of making intelligent machines, especially intelligent computer programs.*

The concept of *intelligence* itself also has numerous definitions, which seem to agree that one of the main characteristics of intelligent behaviour is the ability to learn from experience.

The term *artificial intelligence* dates back to 1956, when it was introduced at the Dartmouth Summer Research Conference on Artificial Intelligence, which

[1] http://www-formal.stanford.edu/jmc/whatisai.html

was organized by John McCarthy, Marvin Minsky, Nathan Rochester and Claude Shannon. The conference lasted for a month and was essentially a brainstorming session to proceed on the basis of the conjecture that *every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it* (McCarthy et al. 1955).

Last summer the field of AI celebrated its 50th anniversary with a conference commemorating the founding Dartmouth College. Also in Belgium, a symposium was organized last November, emphasizing the most influential AI realizations in Flanders. Interestingly, two of the ten talks were devoted to the field of machine learning, which brings us to our next topic.

1.1.2 Machine learning

Machine learning is the sub-domain of AI concerned with developing algorithms and techniques that allow computer programs to learn. While the concept of learning is difficult to define, Langley (1996) provides the following practical definition.

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

Since the first workshop on machine learning, which was held in 1980 and was attended by some 30 participants, the field has considerably grown. For example, in 2006 the 23rd International Conference on Machine Learning hosted 140 technical presentations. Given the huge amount of researchers in the field, it should come as no surprise that the research domain knows several subfields that each has its own view on learning or focusses on particular learning tasks. A significant part of the research in machine learning has been concerned with extracting new knowledge out of available data, also called inductive learning. The available data is most often represented as a table, where each row represents an example (data entity), and each column a property (called *attribute*) of the example. If the aim of the learning task is to predict the value of one specific attribute (called the *target attribute*), one speaks of predictive learning. If, on the other hand, the task is to provide general descriptions of the examples, it is called descriptive learning.

Example 1.1 *Consider the table in Fig. 1.1. It presents 7 examples in the context of a university's student restaurant. Each example represents a day and is described by 4 attributes: the day of the week (Monday to Friday), the weather that day (sunny or rainy), whether there was a party the night before, and whether many students ate at the restaurant that day. If descriptive learning techniques were applied to this dataset, they might come up with descriptions as "on Fridays it always rains" or "on Thursdays many students eat at the restaurant". If predictive learning algorithms were used to predict, e.g., the attribute*

STUDENT_RESTAURANT

Day_Of_Week	**Weather**	**Party_Last_Night**	**Plates_Sold**
Monday	sunny	yes	few
Wednesday	sunny	no	many
Friday	rainy	no	many
Wednesday	sunny	yes	few
Thursday	rainy	no	many
Friday	rainy	yes	few
Monday	rainy	no	few

Figure 1.1: *Student restaurant* example.

`Plates_Sold`*, the following property might be discovered: "on Mondays or on days that had a party the night before, few students come to eat; on any other day many students come to the restaurant".*

Often, one wants to learn descriptions of examples that are contained in a large database, which brings us to the concept of data mining.

1.1.3 (Relational) data mining and inductive logic programming

The field of data mining is concerned with finding generally valid regularities (called patterns) in large databases. It is usually considered to be part of the broader task of knowledge discovery in databases, which also encompasses the tasks of data pre-processing and post-processing of the results.

Progress in computer technology has made generation and storage of data easy and inexpensive, resulting in the availability of huge volumes of data about various domains (e.g., finance, bio- and chemoinformatics, marketing and retailing,...).

Example 1.2 *Most supermarkets offer some kind of loyalty card or rewards card to their customers. By presenting the card, the customer is typically granted either a discount on the current purchase, or an amount of points that can be collected for later purchases. Meanwhile, the store is entitled to record the customer's purchases and to use the stored information as part of its marketing research.*

The availability of such large volumes of data created a problem of how to make use of this data. There is a great demand for powerful tools that turn the vast amounts of data into useful and manageable knowledge. Therefore, the field of data mining has gained considerable importance in business and scientific applications.

Example 1.3 *Let us return to the retailing example given in Ex. 1.2. The large amounts of data recording the purchases of all customers in the loyalty program allow the supermarket to extract valuable knowledge that can be used in marketing strategies. For example, it may be found that if a customer buys beer he is also likely to buy pizza. This regularity may be exploited by positioning the beer and the pizza close together in the store in order to improve sales.*

Traditional data mining algorithms require the data to reside in one table and use ideas and techniques from the fields of statistics and machine learning to extract valid patterns. In practice, however, this single-table assumption turns out to be a limiting factor for many applications. Given the availability of a large number of commercial relational database systems, more and more data is stored in multiple, interconnected, tables. Whereas the first reaction from the data mining community was to transform the data to fit in one table and apply well-established existing algorithms (this approach is still an active research topic), a new research area began to emerge where techniques were developed to learn patterns directly from the relational data. An important contribution to this topic was the development of Inductive Logic Programming (ILP), a research field at the intersection of machine learning and logic programming. Within ILP, the relational database as well as the learned patterns are represented as first order logic programs, which makes it relatively easy to express background knowledge, and provides an expressive, well understood, and theoretically founded basis. It is exactly in this area of relational learning and ILP that the research presented in this dissertation is situated.

1.2 Motivation and contributions

In relational learning one learns patterns from relational datasets, which usually contain multiple tables that are interconnected via relations. These relations may be of one-to-many or many-to-many cardinality ratios. Thus, an example for which a prediction is to be given may be related to a set of objects that are possibly relevant for that prediction. Current relational learners usually handle these sets in one of two possible ways: by imposing conditions on the elements that they contain or by using an aggregate function to summarize them. The main topic of this thesis is the investigation of the combination of both approaches, i.e., aggregating over a subset of elements that fulfil a specific condition. This combination of aggregates and selections presents several difficulties, which are addressed in this work.

For implementing and testing the developed techniques in a practical system, we used the pattern language of decision trees, mainly for two reasons:

- decision trees, together with rule sets, are amongst the most popular patterns, mainly because of their easy interpretability, and

- a relational decision tree learner, which is still a widely used ILP system today, has been developed at our research group.

However, many of the developed techniques carry over to other patterns, such as rule sets.

The main contributions of this dissertation can be summarized as follows:

- We propose complex aggregate conditions, which are a combination of aggregates and selections, in the ILP framework. The classical ILP refinement operator that is used to search the space of possible hypotheses is redefined to take into consideration complex aggregates. The main problems of combining aggregates with selections are, first, a substantial increase of the hypothesis space and, second, the violation of the generality order of the hypotheses assumed by the refinement operator.

- A general refinement framework for complex aggregate conditions is developed that solves the generality problem. The framework is applicable to any relational learner that learns complex aggregates. Next to keeping the generality order in the hypothesis space, the refinement operator based on this framework can yield substantial efficiency improvements.

- The first order decision tree algorithm is upgraded to a random forest inducer. Essentially, an ensemble of decision trees is constructed in which only a sample of all possible tests is tried at each node. Random forests provide a solution for the increase of the feature space and are able to lift the predictive performance.

- We also propose to use complex aggregates to construct predictions in the conclusion part of a hypothesis. More precisely, we investigate the use of complex aggregates in the linear equations in the leaves of model trees. For that purpose, the first order decision tree learner is upgraded to a model tree induction system.

- An important contribution in the development of the first order model tree learner is the design of a suitable heuristic function. For this purpose, we first study existing heuristics, which are all either too expensive to compute in our application or are not specifically targeted for predicting linear equations. We propose a new heuristic function that overcomes both deficiencies.

1.3 Structure of the text

In this chapter we have briefly outlined the context of this work. **Chapter 2** provides a more in-depth explanation of the research field of data mining, its sub-domain focussing on relational databases, and the related subject of inductive logic programming. It describes the basic background and terminology upon which the following, more technical, chapters are built.

Chapter 3 introduces complex aggregates. After redefining the classical refinement operator used in ILP, problems are discussed that occur when refining clauses with complex aggregates. These problems are the explosion of the search space considered at each refinement step, the semantics of the aggregate functions, and the violations of the generality ordering underlying the refinement operator. In order to solve this last problem, a new refinement operator is proposed.

In **Chapter 4** the complex aggregate conditions are integrated into a first order decision tree learner. We explain that in this context, the generality problem does not occur. Two techniques are presented to deal with the size of the search space: the use of random forests in first order logic, and restructuring the search space by making use of the refinement framework presented in Chapter 3.

While in Chapters 3 and 4 the complex aggregates are used in the antecedent (condition part) of a hypothesis, in **Chapter 5** we investigate their use in the consequent (conclusion part). This research is done in the context of first order model trees. The first order decision tree algorithm is upgraded to learn model trees that contain complex aggregates in the linear equations constructed at their leaves. For this purpose, an efficient heuristic function is developed.

Chapter 6 discusses two applications in which complex aggregates are used in real life problems. The first application deals with an agricultural dataset in which the task is to predict the level of contamination with genetically modified variants of an oilseed rape field. The second application discusses the task of predicting gene function in the genome of baker's or brewer's yeast.

Finally, **Chapter 7** presents general conclusions for this dissertation and suggests a number of directions for future work.

1.4 Bibliographical note

Most parts of this work have been published before. The following list contains the key articles. A complete publication list of the author can be found at the end of this text (page 169).

- Introduction of complex aggregate conditions and development of a first order random forest algorithm:
  - A. Van Assche, C. Vens, H. Blockeel, and S. Džeroski, First order random forests: Learning relational classifiers with complex aggregates, Machine Learning 64, pp. 149-182, 2006.
- Development of a general refinement framework for complex aggregate conditions:

  - C. Vens, J. Ramon, and H. Blockeel, Refining aggregate conditions in relational learning, Proceedings of European Conference on Principles and Practice of Knowledge Discovery in Database, Berlin, Germany (Fürnkranz, J. and Scheffer, T. and Spiliopoulou, M., eds.), pp. 383-394, 2006.

- Development of an efficient heuristic function for learning model trees:
  - C. Vens, and H. Blockeel, A simple regression based heuristic for learning model trees, Intelligent Data Analysis 10 (3), pp. 215-236, 2006.

- Development of a first order model tree learner that constructs linear equations with complex aggregates in the leaves:
  - C. Vens, J. Ramon, and H. Blockeel, Re-Mauve, a relational model tree learner, ILP'06, 16th International Conference on Inductive Logic Programming, Post-proceedings (Muggleton, S. and Otero, R., eds.), accepted.

- The biological application:
  - J. Struyf, C. Vens, T. Croonenborghs, S. Džeroski, and H. Blockeel, Applying predictive clustering trees to the inductive logic programming 2005 challenge data, Inductive Logic Programming, 15th International Conference, ILP 2005, Late-Breaking Papers (Kramer, S. and Pfahringer, B., eds.), pp. 111-116, 2005.

- The agricultural application:
  - A. Ivanovska, C. Vens, and S. Džeroski, Using ILP to study the presence of genetically modified variants in organic oil seed rape, ILP'06, 16th International Conference on Inductive Logic Programming, Short Papers (Muggleton, S. and Otero, R., eds.), pp. 107-109, 2006.

Chapter 2

Background

This chapter describes the basic fundamental background on which this work is based. We start by introducing the research areas of knowledge discovery in databases and data mining (Sect. 2.1). We describe what patterns in data are, and discuss two popular forms, linear equations and decision trees, together with methods for finding them in Sect. 2.2. In Sect. 2.3 we move from data mining to relational data mining, covering concepts as propositionalization and inductive logic programming. Section 2.4 focusses on the use of predictive attributes in relational data mining and provides connections with the rest of the thesis.

2.1 Knowledge discovery in databases and data mining

Over the last decades many companies and organizations have generated large amounts of data, stored in a database. Query languages like SQL can be used to process the data, if the user knows what information he is looking for. However, given the often massive datasets the user is faced with, new questions related to making business decisions, predicting future gains given historical data,... have naturally arisen, and can not be formulated by a specific query. This has led to the emergence of a new research area known as knowledge discovery in databases (KDD), defined as follows (Fayyad et al. 1996):

Definition 2.1 (Knowledge discovery in databases) *Knowledge discovery in databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable structure in data.*

The process of KDD contains many steps, which are often iterated. The main step in this process is data mining (Fayyad et al. 1996).

Definition 2.2 (Data mining) *Data mining is a step in the KDD process consisting of applying computational techniques that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data.*

Other steps in KDD include data understanding, data preparation and evaluation of the discovered structure. The whole process of KDD is described in detail in the *Cross Industry Standard Process for Data Mining* (Chapman et al. 2000).

The term *data* used in the above definitions denotes a set of facts, also called instances or examples. A dataset is often contained in a single table where each row represents an example, and each column a property, called an attribute or feature. An attribute is either nominal, when it takes values in a predefined finite set of discrete values, or numeric, when it is continuous. This setting where data resides in a single table is also referred to as *attribute-value* data mining or *propositional* data mining . When data from more than one table needs to be considered, we use the term *relational* data mining (see Sect. 2.3).

The central term in the definition of data mining is *pattern*. Frawley et al. (1991) define a pattern in a dataset as a statement that describes relationships in a subset of the dataset with some certainty, such that the statement is simpler (in some sense) than the enumeration of all facts in the dataset. To find these patterns, data mining algorithms are used. Many of them stem from the fields of machine learning (Mitchell 1997) and statistics (Hastie et al. 2001). From the viewpoint of machine learning, algorithms perform a search through some *hypothesis space*, where a hypothesis corresponds to a pattern. Thus, we can see data mining as searching a hypothesis space in order to learn interesting patterns that are valid in the data (Džeroski 2001a).

There are two main data mining tasks. In *predictive* data mining one is concerned with predicting the value of a distinguished attribute, called the *target attribute* or simply the *target*. Thus, the N attributes in a dataset consist of one target and $N-1$ *predictive attributes*. If the target is nominal, it is also called the *class* and the prediction task at hand is called *classification*. If, on the other hand, the target is numeric, the task is called *regression*. In *descriptive* data mining, on the other hand, there is no target attribute; all attributes play the same role and the task is to find generally valid patterns. Examples are the discovery of association rules (Agrawal et al. 1996), or clustering (Michalski 1987), which is concerned with partitioning the examples in similar subgroups. In this text, we will mainly concentrate on prediction.

When a pattern is found, the next step is to determine the quality of it. Usually the dataset is split in two parts: the *training set* is the part of the data used to learn the patterns, and the rest, called the *test set*, is used to test the validity of the patterns obtained. A frequently used ratio is employing two thirds of the data for training and one third for testing. However, discarding one third of the data from being used to learn the model can sometimes result in inferior models. Therefore, a commonly used technique is to use all data for learning a model and using *cross validation* to estimate the quality of the model. Cross validation partitions the dataset in N equally sized folds (e.g., $N = 10$) and the learning process is repeated N times, each time using one of the folds as test set and the remaining folds as training set. The resulting

SPENDING_MONEY

<u>Name</u>	Age	Gender	Siblings	Par_Income	Spending_Money
Jimmy	15	male	2	48000	20
Kelly	17	female	0	45000	50
Andy	14	male	1	30000	5
Lenny	16	male	1	38000	35
Jenny	15	female	3	50000	25

Figure 2.1: *Spending money* example.

statistics are then averaged. For classification tasks the evaluation measure takes into account the ratio of examples that are well classified, called the *(predictive) accuracy*. For regression tasks, a frequently used measure is the *mean squared error*: the average, calculated over all examples, of the squared difference between the target value and the predicted value.

We illustrate the introduced terminology with an example.

Example 2.1 (Spending money) *Consider a dataset that records information about the amount of spending money a teenager receives from his parents. The dataset is represented by a single table, which is shown in Fig. 2.1. There are 6 attributes, of which* Spending_Money *is the target. Since the target is numeric, we have a regression problem. However, one could easily obtain a classification problem by discretizing the target attribute into, e.g., the values* {low, medium, high}*. The attribute* Gender *is a nominal attribute and takes values in the set {male, female}, while* Age, Siblings*, and* Parent_Income *are numeric. The attribute* Name *is used as a key attribute, in order to identify the examples, and hence does not play a role in prediction tasks.*

2.2 Patterns and their learning algorithms

As discussed in the previous section, the task of data mining is to find patterns that are valid in the data. There exist several forms of patterns, each having its own set of learning algorithms. For example, in predictive data mining, patterns can be linear equations, decision trees, predictive rules, probabilistic models (e.g., bayesian models), artificial neural networks, support vector machines, instance based models,... We discuss the first two in detail, because they will be of particular interest in the rest of the text. For a discussion of the other approaches we refer to Mitchell (1997), Witten and Frank (1999), or Witten and Frank (2005).

2.2.1 Linear equations

The use of linear equations for prediction is probably the most widely used statistical method in data mining. Linear equations take the form

$$Y = c_0 + c_1X_1 + c_2X_2 + ... + c_NX_N$$

and predict a numeric value Y as a linear combination of predictive attributes X_i. Hence, they can be used for performing regression tasks. The patterns are sometimes referred to as linear regression models[1].

An example of a linear regression model for the *Spending money* dataset (Ex. 2.1) is

$$Sp\_Money = -221.89 + 16.17Age + 2.47Nb\_Siblings - 0.0001Parent\_Income$$

Linear equations represent hyperplanes in multi-dimensional spaces. In the special case where only one predictive attribute is used, the equation can be depicted as a straight line in a two-dimensional space. This kind of regression model is called *simple linear regression* model, in contrast to *multiple linear regression* models.

When employing a linear regression model for prediction, the task reduces to finding suitable values for the coefficients c_i. The most widely used method for this purpose is to use the method of *least squares*, i.e., one tries to find the set of coefficients $C = (c_0, ..., c_N)$ that minimizes the squared error between target value and predicted value, summed over all examples. More formally, one searches for the optimal set of coefficients C^*, such that

$$C^* = \arg\min_C \sum_i (y_i - \sum_j c_j x_{ij})^2$$

where $C = (c_0, ..., c_N)$, y_i is the target value for example i and x_{ij} is the value for the j-th predictive attribute for example i (with $x_{i0} = 1$). In the case of simple linear regression the coefficients c_0 and c_1 in the formula $Y = c_0 + c_1X$ are calculated as

$$c_1 = \sum_i (x_i - \overline{x})(y_i - \overline{y}) \;/\; \sum_i (x_i - \overline{x})^2$$
$$c_0 = \overline{y} - c_1\overline{x}$$

where $\overline{x}$ denotes the average value of the predictive attribute X and $\overline{y}$ the average value of the target Y. Also for multiple linear regression the method of least squares can be applied. Note that in both cases, since the coefficients can be calculated directly from a formula, no search through the space of possible equations takes place. In the case of multiple linear regression, the task

[1]When saying a model is linear, one actually refers to linearity in the *parameters*. For example, the equation $Y = c_0 + c_1Z_1^2 + c_2log(Z_2)$ is also a linear regression equation, thus, in the equation above, the X_i can in fact be any function of the original predictive attributes.

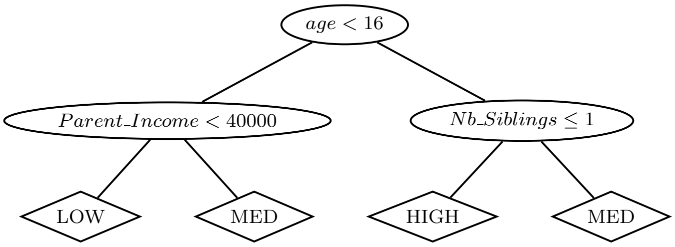


Figure 2.2: Classification tree for the *Spending money* example.

of parameter estimation is often accompanied by the task of variable selection (Kohavi and John. 1997), to determine what attributes to include in the equation.

While linear equations are mostly used for regression tasks, they can also be employed for classification, using generalized linear models, e.g., logistic regression (McLachlan 1992).

2.2.2 Decision trees

A decision tree is a tree-shaped hierarchical predictive model that can be used both for classification and regression tasks. In the former case, the tree is called a classification tree, in the latter case a regression tree. Each inner node of a tree contains a test on one of the predictive attributes, and each leaf node yields a prediction for the target attribute.

An example classification tree for the *Spending money* example (Ex. 2.1) is given in Fig. 2.2. In this tree the target attribute Spending_Money was discretized in the intervals $[0, 15], [16, 30]$, and $[30, \infty]$ denoted by *low, medium*, and *high* respectively.

Prediction for an unseen example is done by sorting the example down the tree, starting at the root. Depending on the outcome of the test at each node, either the left branch (the test succeeds) or the right branch (the test does not succeed) is taken. Eventually, the example ends up in a leaf and the predicted value is obtained. The tree depicted in Fig. 2.2 only contains tests on numeric attributes. Tests on nominal attributes may have a branch for each possible value (possibly resulting in non-binary trees) or may test whether the value for the unseen example is element of a subset of the possible values, e.g., $gender \in \{female\}$.

Decision trees are among the most popular types of patterns used in data mining. This is not surprising as they are easy to interpret by users. Also, decision tree induction algorithms typically possess some nice properties (Kramer and Widmer 2001):

- Low computational complexity: the induction of decision trees has a time

Table 2.1: A generic TDIDT algorithm.

```
procedure TDIDT ($E$: examples):
    if stop_criterion ($E$)
    then return leaf(local_model($E$))
    else
        $S$ := set of all possible splits
        $s^*$ := $\arg\max_{s \in S}$ quality($s$, $E$)
        $\mathcal{P}$ := partition induced on $E$ by $s^*$
        for all $P_j$ in $\mathcal{P}$:
            $T_j$ := GROW_TREE($P_j$)
        return node($s^*$, $\bigcup_j \{(j, T_j)\}$)
```

complexity linear in the number of examples.

- Effective handling of noise: sophisticated pruning techniques have been developed for dealing with noise in the data.

- Well-understood theoretical basis: the basic characteristics of decision tree induction algorithms are well understood.

Although decision tree induction algorithms have low computational complexity, the general task of finding the smallest decision tree that fits a given dataset is NP-hard (Zantema and Bodlaender 2000). Therefore, induction algorithms use heuristic, usually greedy, search techniques to build the tree. Just as prediction for an unseen example is done by sorting the example from the root down to a leaf, also the tree building process usually starts at the root and builds the tree in a top-down manner. This is called the Top Down Induction of Decision Trees (TDIDT) approach, of which the algorithms CART (Breiman et al. 1984) and C4.5 (Quinlan 1993) are popular implementations. A generic variant of the TDIDT approach is shown in Table 2.1.

The method recursively partitions the examples such that maximally homogeneous (with respect to the target attribute) subsets are obtained, and recursively re-applies this procedure on the subsets until a stop criterion holds (e.g., all subsets are sufficiently homogeneous, or a predefined minimum number of examples in the leaf is reached). Thus, the main task is to decide which split condition results in the best partitioning of the instances. An estimate of the quality of a split is usually computed with a heuristic function, such as information gain (Quinlan 1993) or gini-index (Breiman et al. 1984) in the case of classification trees, or reduction of variance (Breiman et al. 1984) for regression trees. If the stop criterion holds, a leaf is constructed and a prediction for the target attribute is given. For classification tasks the majority class of the training instances resulting in the leaf is taken; for regression tasks their average target value. This procedure results in a greedy search for the best decision tree.

Often it is hard to determine a good stop criterion for the growing phase: stopping the growing phase too early may result in trees that are not able to capture all relevant information in the data, whereas growing overly large trees may lead to difficulties in the presence of noise in the data. In the last case, we say the tree is overfitting the data: there may exist an alternative tree that performs worse on the training set, but with better predictive performance on the unseen test set. A successful approach to dealing with overfitting is to grow an initial tree that is allowed to overfit the data and afterwards post-prune the tree, i.e. cutting away nodes if the resulting pruned tree performs no worse than the original one. Popular pruning techniques include reduced error pruning (Quinlan 1988) and rule post pruning (Quinlan 1993).

2.3 Relational data mining and inductive logic programming

In the previous section we have introduced the concepts of knowledge discovery in databases and data mining and have described two kinds of patterns together with their learning algorithms. Since these learning algorithms assume that the input data is stored in a single table where each example is represented by a fixed number of attributes, they are called attribute-value or propositional (as the patterns found can be expressed in propositional logic) techniques. Propositional data mining techniques are popular, mainly because they are efficient, easy to use, and widely accessible. WEKA (Witten and Frank 1999), Orange (Demsar and Zupan 2004), and KDB2000 (Appice et al. 2002) are only a few of the non-commercial, freely available data mining tools that let the user play around and experiment with several learning techniques to mine his data. In practice, however, the single-table assumption turns out to be a limiting factor for many data mining tasks. Indeed, examples of data mining problems that involve data residing in multiple related tables abound. For an overview, see Džeroski (2001b). In Sect. 2.3.1 we present two such example data mining tasks. These tasks can be solved in two ways: transforming the relational problem into a propositional form and applying a propositional learning technique (Sect. 2.3.2), or tackling the problem in its original relational representation using techniques from relational data mining (Sect. 2.3.3). Relational data mining (Džeroski and Lavrač 2001b) is a sub-area of data mining that looks for patterns that involve multiple related tables. A lot of relational data mining algorithms come from the field of inductive logic programming, which is also presented in Sect. 2.3.3.

2.3.1 Relational data

The data explored by relational data mining approaches typically resides in a relational database. A relational database (Elmasri and Navathe 2004) consists of several tables that are related to one another. Each table represents some real world object or a relation between objects and has several attributes that define properties of the data residing in the table. One of the attributes (or a combination of several of them) forms the primary key, this means that the value for this attribute is different for each row in the table. Tables are interconnected via foreign key links. A foreign key is an attribute that takes the value of a primary key of another table. Relationships between tables can be of the following cardinality ratios: one-to-one, one-to-many, many-to-one, and many-to-many.

When performing data mining on a relational database, one of the tables contains the target attribute, this table is called the target table. The rows in the target table represent the examples.

We discuss two examples that will be used to illustrate several techniques throughout the text: a simple synthetic example called *Account*, and *Mutagenesis*, a well known relational dataset that is often used as benchmark for testing relational data mining techniques.

Example 2.2 (Account) *The ER (entity-relationship) diagram and an extension of the corresponding relational database schema of the synthetic Account example are shown in Fig. 2.3 and Fig.2.4, respectively. The database consists of three tables: Person, Account, and Transaction. A person is to be considered as a bank customer for whom information on his personal accounts is stored. A person is identified by a primary key named PersonID (e.g., his social security number, or client number, or name). Further information stored about a person is his age, and some non-specified target attribute (e.g., the chance that the person will be interested in buying a particular stock option). The table Person is thus the target table. A person can have 0 or more accounts, each having an Id (account number), a type (savings or checkings), and a balance. For simplicity, we assume that an account can have only one owner. For each account, all transactions are saved. A transaction is described by a transactionID, a date, a type (deposit or withdrawal), and an amount. The attribute AccountID in the table Transaction is an example of a foreign key, since it points to the primary key (AccountID) of the table Account.*

Example 2.3 (Mutagenesis) *The Mutagenesis dataset was introduced to the ILP community by Srinivasan et al. (1996), and comes from the field of organic chemistry. The task is to predict mutagenicity of 230 nitro-aromatic compounds. These compounds occur, for example, in automobile exhaust fumes. Mutagenic compounds have often found to be carcinogenic and may cause damage to DNA. For each compound, the molecular structure, in terms of atom*

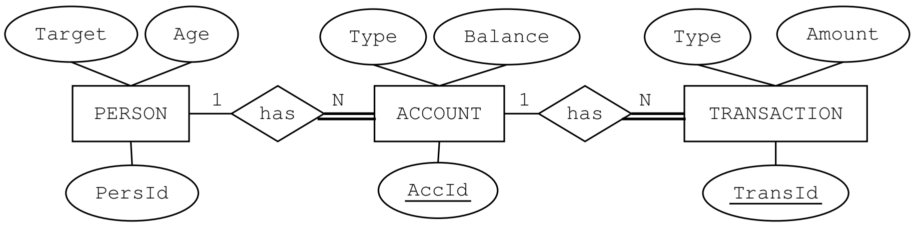


Figure 2.3: ER-diagram of the *Account* example.

PERSON

PersId	Age	Target
john	24	positive
mary	27	negative
billy	25	negative

ACCOUNT

PersId	AccId	Type	Balance
john	123456	checkings	100
john	987654	checkings	200
john	789123	savings	200
john	891234	savings	50
mary	567890	checkings	50
mary	345678	savings	250
billy	456789	checkings	150

TRANSACTION

AccId	TransId	Date	Type	Amount
123456	tr090	010706	withdrawal	50
123456	tr091	030706	deposit	30
987654	tr098	120706	deposit	70
789123	tr100	150706	withdrawal	100
789123	tr101	150706	deposit	80
567890	tr150	220706	withdrawal	20
567890	tr150	240706	deposit	40
456789	tr122	170706	deposit	50

Figure 2.4: Extension of the *Account* database.

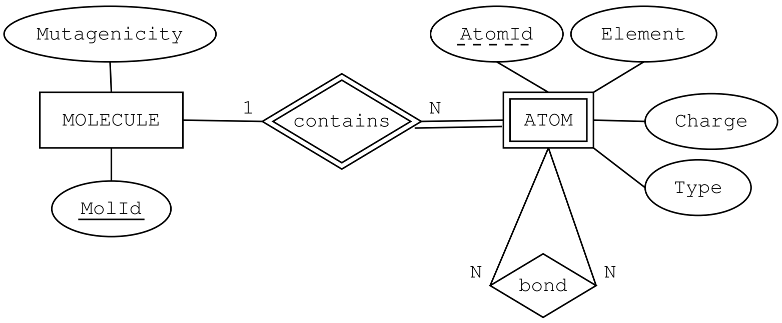


Figure 2.5: ER diagram of the *Mutagenesis* example.

and bond connectivities, is stored. The corresponding ER diagram is shown in Fig. 2.5 and results in three tables: Molecule (the target table), Atom, and Bond. The first table stores the ID and target attribute for the 230 molecules. The target can either be numeric: the logarithm of the mutagenicity level of the compound, or nominal: positive if logarithm is positive and negative otherwise. The atoms consist of an ID, an element (e.g., carbon, nitrogen,...), a type (e.g., aromatic carbon, aryl carbon,...) and a partial charge. The table Bond represents bonds between atoms. Each bond is identified by the two atoms it binds, and its description also contains a type (e.g., aromatic, single, double,...).

In order to discover patterns in relational data, in general two strategies can be taken. One can first transform the data into attribute-value format and apply one of the many available propositional data mining techniques, or one can learn patterns directly from the relational data, applying a relational data mining technique. We discuss these two approaches in the next sections.

2.3.2 Propositionalization

Propositionalization approaches, sometimes called transformation based approaches, consist of two steps:

- transform the relational data into attribute-value format, i.e, into one single table, and
- apply a propositional learning technique to this table.

Sometimes, a third step is considered:

- transform the obtained propositional patterns into relational patterns.

The propositional table is constructed by starting from the target table and subsequently joining it with other tables, by following the foreign key links. If the relational data contains one-to-many or many-to-many relationships, more than one row per example is obtained. In most systems, they are transformed back to one row by summarizing methods, such as aggregate functions[2].

Examples of early propositionalization systems are LINUS (Lavrač et al. 1991; Lavrač and Džeroski 1994) and its successor DINUS (Lavrač and Džeroski 1994). They are not able to introduce non-determinate attributes in the universal table. By non-determinate attributes we mean attributes that may have more than one possible value for an example, due to one-to-many or many-to-many relationships. For example, in the *Account* dataset, the attributes *AccId*, *Type*, and *Balance* are non-determinate, because several values for them may correspond to a given person.

We now discuss two different methods for summarizing non-determinate attributes, such that one row per example is obtained and propositional learning techniques can be applied. Kramer et al. (2001) describe how LINUS can be extended to handle non-determinate attributes. Each non-determinate attribute added to the propositionalized table involves checking the existence of some related entity with a certain property. For example, in the *Account* dataset such a feature might check whether a savings account exists with balance greater than 500. RELAGGS (Krogel and Wrobel 2001) and ROLLUP (Knobbe et al. 2001) are propositionalization systems that use aggregate functions to summarize non-determinate information. For example, one can include into the propositional table the average balance of a person's accounts, the total number of transactions of a person's accounts,...

The advantages of propositionalization approaches can be summarized as follows:

- existing reliable and efficient propositional learning techniques can be used,
- methods for feature selection, meta learning, noise handling,... are directly applicable, and
- feature construction is decoupled from model construction.

The disadvantages include that

- the propositionalization step may require considerable time and effort, and usually involves expensive joining of related tables,
- summarizing the propositional table such that only one row per example is kept almost inevitably leads to a loss of information, (if one does not want loss of information, usually a large amount of redundant data is generated;

[2]An exception is the system by Zucker and Ganascia (1998), in the context of concept learning. A dataset of positive and negative examples is propositionalised. The task is to find a description that covers at least one row in the universal table for each positive example, and covers no rows for any of the negative examples.

De Raedt (1998) showed that in this case propositionalized feature spaces can be exponential in the number of attributes of the original learning problem), and

- application is limited to a restricted class of relational learning problems, e.g., propositionalization can not deal with recursive relations (Džeroski and Lavrač 2001a).

These disadvantages have lead to the development of techniques that learn directly from relational data.

2.3.3 Relational data mining and inductive logic programming

Relational data mining techniques learn patterns from relational data directly, without transforming them first to a propositional table, we call these patterns relational patterns. Most propositional patterns have a relational counterpart. To induce these relational patterns, existing propositional learning techniques have been upgraded to the relational case. For example, a number of relational decision tree algorithms have been developed. Examples are the work by (Knobbe et al. 1999) that induce decision trees from a relational database, or Tilde (Blockeel and De Raedt 1998), and S-Cart (Kramer and Widmer 2001), two techniques inducing so-called first-order decision trees. The former algorithm is an upgrade of the popular C4.5 algorithm (Quinlan 1993), whereas the latter upgrades the propositional CART system (Breiman et al. 1984). Other propositional learning techniques have been upgraded to induce relational rule sets (De Raedt and Van Laer 1995), relational neural networks (Blockeel and Uwents 2004; Uwents and Blockeel 2005), probabilistic relational models (Jaeger 1997; Koller 1999; Getoor et al. 2001), relational instance based models (Emde and Wettschereck 1996; Horvath and Wrobel 2001; Ramon 2002), relational association rules (Dehaspe and Toivonen 2001),...
Van Laer and De Raedt (2001) present a stepwise methodology for upgrading propositional learners to the relational context.

A number of relational data mining systems operate directly on relational databases through the use of SQL to query the data (Knobbe et al. 1999; Knobbe et al. 1999; Appice et al. 2003). However, most of the research in relational data mining is situated in the field of inductive logic programming (ILP) (Muggleton 1992; Lavrač and Džeroski 1994), a research area situated at the intersection of machine learning and logic programming (Lloyd 1987; Bratko 2001). While the key ideas of ILP were formulated in an early work by Plotkin (1969), the area's real development began in 1991, with the organization of the *First International Workshop on Inductive Logic Programming* (Muggleton 1992), a yearly event since then that has expanded into an international conference. In the rest of this section we concentrate on inductive logic programming.

In ILP, input data as well as discovered patterns are expressed as logic programs, an important subset of first order (predicate) logic. ILP approaches are therefore also referred to as first-order approaches. We first provide some terminology from first order logic and logic programming in Sect. 2.3.3.1. We go on by discussing how to transform a relational database into a logic program in Sect. 2.3.3.2. Next, we provide some explanation regarding the semantics of logic programs (Sect. 2.3.3.3), which is necessary in order to discuss the two main representation approaches in ILP (Sect. 2.3.3.4). Finally we are able to formally define the learning task considered in ILP (Sect. 2.3.3.5) and explain how hypotheses are usually formed (Sect. 2.3.3.6).

2.3.3.1 Logic programs: terminology

We start by explaining the basic terminology used in logic programming. Following the example of Jacobs (2004), we do this in a bottom-up manner, starting from the basic language elements.

- As in many other programming languages, the basic elements to write a logic program are *constants* and *variables*. As is the convention in Prolog (Bratko 2001), a programming language rooted in first order logic, we will write variable names starting with a capital.
- A *term* is a constant, a variable, or a *function* symbol immediately followed by a bracketed tuple of terms. In fact, a constant is a function of arity 0, in which case we do not write the brackets. For example $f(g(X,Y),h(Y),j)$ is a term with function symbols f, g, h, and j (which is a constant), and variables X, and Y. The length of the tuple is called the arity of the term. Thus, the arity of f is 3 and the arity of g is 2.
- A predicate symbol immediately followed by a bracketed tuple of terms is called an *atom* or *predicate*. For example, $person(PId, 12, Target)$ and $account(PId, AccId, savings, Bal)$ are predicates of arity 3 and 4, respectively.
- A *literal* is an atom, or the negation (written as $\neg$) of an atom.
- A *clause* is a disjunction of literals, for example $h_1 \vee h_2 \vee ... \vee h_k \vee \neg b_1 \vee \neg b_2 \vee ... \vee \neg b_l$. It is common to write a clause as an implication $h_1 \vee h_2 \vee ... \vee h_k \longleftarrow b_1 \wedge b_2 \wedge ... \wedge b_l$, where $h_1 \vee h_2 \vee ... \vee h_k$ is called the head of the clause (conclusion part), and $b_1 \wedge b_2 \wedge ... \wedge b_l$ is called the body (condition part). Often, the $\wedge$ and $\vee$ symbols are replaced by commas. In a clause all variables occurring in the literals are universally quantified.
- A conjunction of clauses is called a *(clausal) theory*.
- According to the number of literals in the head or body of a clause, we can distinguish some special forms. A *Horn clause* is a clause with maximum

one literal in the head. A *definite clause* is a clause with exactly one literal in the head. A *query* is a clause with no literals in the head. A *fact* is a definite clause with no literals in the body. Usually, the implication arrow is omitted for facts.

- A *program clause* is a clause of the form $h \longleftarrow l_1, l_2, ..., l_m$, where h is an atom and $l_1, l_2, ..., l_m$ are (positive or negative) literals.
- A *predicate definition* is a set of program clauses with the same predicate symbol and arity in their heads.
- And finally, a *logic program* is a set of predicate definitions.

2.3.3.2 Relation between logic programs and relational databases

To apply an ILP technique, the input data needs to be transformed to a logic program. Fortunately, a relational database can be easily converted into a logic program (Das 1992). A base table of a relational database corresponds to a predicate in a logic program. The predicate has the same name as the table. The attributes of the table correspond to the arguments of the predicate. The predicate corresponding to the target table is called the target predicate. An extension of the database is written as an enumeration of facts.

Example 2.4 *To represent the extension of the Account database given in Fig. 2.4 as a logic program, three predicates would be needed: person/3*[3]*, account/4, and transaction/5, of which person/3 is the target predicate. The Prolog representation is shown in Table 2.2.*

Next to base relations, relational databases may contain views. They can be represented in logic programs in the form of clauses.

Example 2.5 *In the Account example, young persons that have saved a specified amount of money during the period of one year may be granted benefits, such as a cinema ticket or a discount for a concert ticket. This could be specified as follows, assuming a predicate saved_last_year is available.*

$$\begin{aligned} saving\_youngster(PersId) \longleftarrow & age(PersId, Age), Age < 24, \\ & saved\_last\_year(PersId, Amount), \\ & Amount > 2000. \\ age(PersId, Age) \longleftarrow & \ person(PersId, Age, Target). \end{aligned}$$

Example 2.6 *In the Mutagenesis dataset, molecules are described by atoms and bonds. Bonds describe connections between two atoms and they are known to be undirected. Thus, whenever a fact* $bond(Mol, At1, At2, Type)$ *occurs, there*

[3]This notation is used in logic programming to describe a predicate by its functor and its number of arguments in the form of functor/numargs.

Table 2.2: Logic program representation of the *Account* database extension of Fig. 2.4.

```
person(john,24,positive).
person(mary,27,negative).
person(billy,25,negative).

account(john,123456,checkings,100).
account(john,987654,checkings,200).
account(john,789123,savings,200).
account(john,891234,savings,50).
account(mary,567890,checkings,50).
account(mary,345678,savings,250).
account(billy,456789,checkings,150).

transaction(123456,tr090,010706,withdrawal,50).
transaction(123456,tr091,030706,deposit,30).
transaction(987654,tr098,120706,deposit,70).
transaction(789123,tr100,150706,withdrawal,100).
transaction(789123,tr101,150706,deposit,80).
transaction(567890,tr150,220706,withdrawal,20).
transaction(567890,tr150,240706,deposit,40).
transaction(456789,tr122,170706,deposit,50).
```

is also an implicit fact bond$(Mol, At2, At1, Type)$. *Instead of writing two bond predicates whenever two atoms are connected, we can write one of them and add the following clauses:*

$sbond(Mol, At1, At2, Type) \longleftarrow bond(Mol, At1, At2, Type).$
$sbond(Mol, At1, At2, Type) \longleftarrow bond(Mol, At2, At1, Type).$

The idea is that when learning patterns that predict mutagenicity, we can use the predicate sbond instead of bond.

Thus, a logic program representing data coming from a relational database usually consists of two parts. The *extensional* part consists of facts representing the base relations and the *intensional* part consists of a number of clauses that represent views from the relational database. Clauses can also be included to supply domain expert background knowledge to the ILP algorithms. For example, in the context of the Mutagenesis dataset, the definition of functional groups (i.e., frequently occurring substructures in the molecule) may be added. The combination of the intensional and extensional parts is called the *knowledge base*. When we refer to a *dataset* in this text, we mean the knowledge base, although in most cases our datasets will only consist of extensionally defined predicates. There are different interpretations as to which part of the knowledge base belongs to the examples and which part is background. Before introducing the two main paradigms, we first provide some explanation on the semantics of logic programs.

2.3.3.3 Logic programs: semantics

We sketch some basic notions about the semantics of logic programs; they will be needed to formalize the task of inductive logic programming.

- A *substitution* $\sigma = \{V_1/t_1, V_2/t_2, ..., V_n/t_n\}$ is an assignment of the terms $t_1, t_2, .., t_n$ to the variables $V_1, V_2, ..., V_n$, respectively. A substitution σ can be applied to a term, atom, clause, or theory T and is denoted as $T\sigma$, i.e., the term, atom, clause, or theory T with all occurrences of the variables $V_1, V_2, ..., V_n$ simultaneously replaced by the terms $t_1, t_2, .., t_n$.
- An *interpretation* is a function that maps a theory to the values *true* or *false* and can be determined by a set of ground facts (i.e., facts without variables). Assigning a truth value to a theory proceeds as follows:
  - All ground facts that determine the interpretation are assigned the value true.
  - Ground clauses are assigned a truth value by considering the truth value of their ground atoms and a set of general rules for interpreting logic operators (e.g., $\neg A$ is true iff A is false; $A \longleftarrow B$ is true iff B is false or A is true,...).

  – Atoms, clauses or theories T that contain variables are considered true if $T\sigma$ is true for each substitution σ that makes $T\sigma$ ground.

- An interpretation that makes a clause or a theory true is called a *model* for that clause or theory.

- A theory T_1 is said to *(logically) imply*, or *entail*, a theory T_2 (denoted by $T_1 \models T_2$) if and only if every model for T_1 is also a model for T_2.

We illustrate these concepts with the Account example.

Example 2.7 *The interpretation* $\mathtt{I}_1 =$
$\{\mathtt{person(john, 24, positive)}$, $\mathtt{account(john, 123456, checkings, 100)}$, $\mathtt{account(john, 987654, checkings, 200)}$, $\mathtt{account(john, 789123, savings, 200)}$, $\mathtt{account(john, 891234, savings, 50)}\}$ *is a model for the theory* T_1*:*

$$person(PId, Age, positive) \longleftarrow account(PId, AccId, savings, Bal).$$

because the theory is true for each of the substitutions $\{\{PId/john, Age/24, AccId/123456, Bal/100\}, \{PId/john, Age/24, AccId/987654, Bal/200\}, \{PId/john, Age/24, AccId/789123, Bal/200\}, \{PId/john, Age/24, AccId/891234, Bal/50\}\}$ *that make the theory ground. For the first two substitutions the theory is true because the body of* T_1 *is false, and for the last two substitutions it is true because both body and head are true.*
The interpretation $\mathtt{I}_2 =$
$\{\mathtt{person(billy, 25, negative)}$, $\mathtt{account(billy, 456789, checkings, 150)}\}$ *is also a model, but the interpretation* $\mathtt{I}_3 =$
$\{\mathtt{person(mary, 27, negative)}$, $\mathtt{account(mary, 345678, savings, 250)}\}$
is not. Theory T_1 *entails theory* T_2*:*

$$\begin{aligned} person(PId, Age, positive) \longleftarrow & account(PId, AccId1, savings, Bal1), \\ & account(PId, AccId2, checkings, Bal2). \end{aligned}$$

since each model for T_1 *is also a model for* T_2*. However, the reverse is not the case, i.e.,* $T_2 \not\models T_1$*, since the interpretation* I_3 *is a model for* T_2*, but not for* T_1*.*

2.3.3.4 Two learning paradigms

We distinguish two main learning paradigms in ILP: *learning from interpretations* and *learning from entailment*. They concern the representation of the knowledge base. We briefly introduce the two approaches here. A more detailed discussion can be found in Chapter 4 by Blockeel (1998). An overview of these and other paradigms is given by De Raedt (1997).

Learning from interpretations. In this paradigm each example e is represented as a separate Prolog program. The program contains all the extensionally defined information for the example, as well as a target label. A separate Prolog program B encodes the generally valid background knowledge. Each example corresponds to an interpretation defined by all ground facts that are entailed by $e \wedge B$.

Example 2.8 *If we would include transaction information into interpretation I_1 given in Ex. 2.7, this interpretation would correspond to the example "john" from the Account dataset.*

Learning from entailment. The learning from entailment setting is the most widely used paradigm in ILP. It was described under the name of *normal semantics* by Muggleton and De Raedt (1994). In this approach the knowledge base (examples and background) is represented as one single logic program, where each example is represented by a ground fact (the target predicate). The extensionally defined predicates that are not target predicates are considered to be part of the background knowledge.

Example 2.9 *Consider the logic program version of the Account database (see Fig. 2.2). In the learning from entailment setting, the examples are the three* `person/3` *predicates. The rest of the program (i.e. the predicates* `account/4` *and* `transaction/5`*) are background knowledge.*

While in the learning from interpretations setting, it is not really necessary to include a key attribute, since each example is contained in a separate program, the learning from entailment setting does require a key attribute in each extensionally defined predicate.

In the remainder of this section, we will assume the learning from entailment setting.

2.3.3.5 Inductive logic programming

The task of inductive logic programming can now be formalized as follows

Definition 2.3 (Inductive logic programming) *Given a set of examples E, and background knowledge B, the task is to find a hypothesis H such that $\forall e \in E : B \wedge H \models e$*

In the above definition, H is typically a definite predicate definition that defines the target predicate, and each example $e \in E$ is a ground fact with the target as one of its arguments, as explained above. ILP is usually defined as a concept learning task (Džeroski and Lavrač 2001a), where the examples are either positive or negative and clauses are searched that entail all positive and no negative examples. We take a slightly different approach that covers both classification and regression, as well as descriptive learning.

Example 2.10 *Given the small extension of the Account example (Table 2.2), a possible hypothesis H that discriminates positive from negative examples is:*

$$\begin{aligned} &person(PId, Age, positive) \longleftarrow Age < 26, account(PId, AccId, checkings, \\ &\qquad\qquad\qquad\qquad\qquad\qquad Bal), Bal > 180, !. \\ &person(PId, Age, negative) \longleftarrow . \end{aligned}$$

stating that a person under 26 owning an account that has a balance of over 180 is positive, and all other persons are negative. The ! (cut-symbol) in the first clause of the hypothesis means that if this clause succeeds, none of the other clauses are evaluated (this ensures that only one prediction is given).

The above hypothesis indeed classifies the example John as positive and the examples Mary and Billy as negative. In practice, however, it will not always be possible to find a hypothesis that perfectly classifies the data, e.g., if the target contains noise. Therefore, we usually try to find the *best* hypothesis that assigns as many examples as possible to the right class.

Example 2.11 *The following is a hypothesis for the Mutagenesis example with an accuracy of 66.5% measured over a tenfold cross validation.*

$$\begin{aligned} &molecule(Mol, positive) \longleftarrow atom(Mol, At, Elem, 27, Charge), !. \\ &molecule(Mol, positive) \longleftarrow atom(Mol, At, Elem, 29, Charge), !. \\ &molecule(Mol, positive) \longleftarrow atom(Mol, At, Elem, 32, Charge), !. \\ &molecule(Mol, negative). \end{aligned}$$

The hypothesis states that a molecule is mutagenic if it contains an atom of type 27, or if it contains no atom of type 27 but does contain one of type 29, or if it contains no atoms of types 27 and 29 but contains an atom of type 32. In all other cases the molecule is predicted non-mutagenic.

We go on with discussing how ILP systems search the space of hypotheses.

2.3.3.6 Searching the space of hypotheses

Probably the most important challenge in ILP, or relational data mining in general, is how to deal with the large hypothesis space. In propositional learning, hypotheses take rules of the form

$$target = t \longleftarrow attr_1 = value_{1k} \land attr_2 = value_{2l} \land ... \land attr_n = value_{nm}.$$

In the relational case, given the fact that hypotheses may contain quantifiers and variables, and that attributes from several background relations can be used, the space of possible hypotheses is much larger, which makes the task of learning more complex. Therefore techniques are used to limit the hypothesis space and to traverse it in an efficient way.

To limit the hypothesis space, one usually imposes restrictions on the syntax of the hypotheses that may be considered. This is done by the use of a language bias (Nédellec et al. 1996), i.e., the user defines what predicates may occur in the resulting hypothesis.

Example 2.12 *Table 2.3 shows how a possible language bias for the Account example would look like in* Tilde *(Blockeel and De Raedt 1998), an ILP system that learns relational decision trees. First, types for the three possible predicates are declared. The first declaration states that a predicate* person *takes an argument of type* persid*, one of type* age*, and one of type* class*. After the type declarations a number of* rmode *specifications follow. These rmodes define how the predicates used in hypotheses may look like. A "+" symbol before a variable name means that the variable is an input variable, it needs to be unified with a variable already occurring in the hypothesis under construction. A "-" means the variable is a new variable. A "+-" means that the variable can but need not be unified with some other variable. The "#" symbol is placeholder for a constant, it will be replaced by each element from the list following the symbol. For more information on* Tilde*'s language bias, see (Blockeel et al. 2006).*

Table 2.3: Tilde's language bias for the *Account* example.

```
% types
type(person(persid,age,class)).
type(account(persid,accid,acctype,balance)).
type(transaction(accid,trans,transtype,amount)).

% rmodes: basic language constructs
rmode(account(+PersID, -AccID, -Tp, -Bal)).
rmode(account(+PersID,+-AccID, #["savings","checkings"],-Bal)).
rmode((account(+PersID, +-AccID, -Tp,Bal), Bal ≥ #[500,2000,5000,10000])).
rmode(transaction(+AccID, -Tr, -Tp, -Am)).
rmode(transaction(+AccID, +-Tr,#["deposit","withdrawal"], -Am)).
rmode((transaction(+AccID, +-Tr, -Tp, Am), Am ≥ #[500,1000,2000])).
```

Hypotheses are typically learned one clause at a time. Traversing the space of possible clauses in an efficient way is done by imposing an order relation on the clauses and exploiting this order relation to systematically search the space. The most common ordering is based on θ-subsumption (Plotkin 1969), and imposes a lattice structure on the space of clauses.

Definition 2.4 (θ-subsumption) *A clause c_1* θ-subsumes *a clause c_2 (denoted $c_1 \leq_\theta c_2$) if there exists a substitution θ such that $c_1\theta \subseteq c_2$.*

Example 2.13 *Clause C_1 :*

$person(PersId, Age, pos) \longleftarrow account(PersId, AccId, Type, Balance)$

θ-subsumes clause C_2 :

$$person(PersId, Age, pos) \longleftarrow account(PersId, AccId, savings, Balance)$$

under the substitution $\theta = \{Type/savings\}$. *Clause* C_1 *also* θ*-subsumes clause* C_3 :

$$\begin{aligned} person(PersId, Age, pos) \longleftarrow & account(PersId, AccId, Type, Balance), \\ & Balance > 500. \end{aligned}$$

In this last example $\theta = \emptyset$.

The θ-subsumption relation has an interesting property regarding generality:

Theorem 2.1 *If* $C_1 \leq_\theta C_2$ *then* C_1 *is at least as general as* C_2.

We define generality in terms of coverage of a clause.

Definition 2.5 (Coverage) *An example e is covered by a clause C if the clause, together with the background B, entails the example, i.e.,* $B \wedge C \models e$.

Definition 2.6 (Generality) *A clause* c_1 *is more general (resp. more specific) than a clause* c_2 *if the set of examples covered by* c_1 *is a superset (subset) of the set of examples covered by* c_2. *We say that* c_1 *is a generalization (specialization) of* c_2.

Thus, θ-subsumption introduces a syntactic notion of generality, which allows to traverse the search space in a general-to-specific order, using simple syntactic computations. Constructing a clause then proceeds as follows. One starts with the most general clause $h \longleftarrow$, i.e., the top of the lattice structure imposed by θ-subsumption. Then so-called *refinements*, or *specializations*, are generated using a *refinement operator* based on θ-subsumption.

Definition 2.7 (refinement operator) *Given a hypothesis language* $\mathcal{L}$ *and some ordering* $<$, *a refinement operator* ρ *is a function mapping a clause c onto a set of clauses* $\rho(c)$ *which are called refinements of c:* $\rho(c) = \{c' | c' \in \mathcal{L}, c < c'\}$.

Typically, the generated refinements are minimal (i.e., most general) refinements of the clause that is being refined. For a refinement operator based on the θ-subsumption order relation, these minimal refinements are

- apply a substitution to the clause, and
- add a basic language construct to the body of the clause.

The basic language constructs are usually defined in a language bias. For the example in Table 2.3, they are specified by the rmodes. Examples of basic language constructs in that context are the literal $account(PersId, AccId, Tp, Bal)$ or the conjunction of literals $(account(PersId, AccId, Tp, Bal), Bal \geq 500)$. Thus, starting from the most general clause, the search space is traversed by repeatedly generating refinements, until at some point in the lattice a clause is obtained that obeys a certain stop criterion. Remark that θ-subsumption can also be used to prune parts from the search space: if the coverage of a clause C_1 is too low, then the clauses θ-subsumed by C_1 need not be searched, since their coverage will be even lower. The search to find the best clause is often based on heuristics, such as the number of positive and negative examples that are covered by it. Given the large number of refinements at each refinement step, greedy methods are often preferred over exhaustive search methods.

2.4 Using background information in relational learning

In this thesis we focus on how the information that is related to an example can be used to predict the target of that example in relational learning. In terms of relational databases, the information that is related to an example E consists of the attributes in the row in the target table that corresponds to E and in those rows in other tables that are linked to E via foreign key relations. For example, for the Account problem, the information that can be used to predict the target for a particular person P is the age of P which is contained in the target table, the values for the balances and types of the accounts of P, and information related to the transactions that are linked to the accounts of P.

For all information that is related to an example E exactly one of the following two properties holds:

- the information is contained in the target table, or in a background table with a one-to-one or a many-to-one relation between the target table and the background table, or

- the information is contained in a background table with a one-to-many or a many-to-many relation between the target table and the background table.

In the first case, the information can be directly extracted from its table and be used in the predictive model. For example, suppose that we also keep track of the branch office a person in the Account database is associated with. Since many persons would be affiliated to the same branch office, there would typically be a many-to-one relation between the person and office tables. With each person corresponds exactly one tuple in the office table. Hence, information about the office (number of employees, level of service,. . .) can be directly included in the predictive model.

The second case, however, is more difficult to handle. The difficulty lies in the non-determinacy of the information: for each example there are several related tuples in the related table. For example, it is non-trivial how the accounts related to a person can be used to predict the target for that person. In present-day relational learners, one of the following strategies is used:

- the existence of a particular tuple for which a number of conditions apply is checked, or
- the information related to the example is summarized by using aggregate functions such as *max*, *min*, *avg*,... and can then be used as if contained in the target table.

For example, the first strategy, which is typically used by ILP learners, could check whether *the person has a savings account with balance larger than 5,000.* The second strategy could check whether *the average balance of the accounts related to the person is larger than 5,000*, or whether *the number of accounts related to the person is larger than 3.* However, none of the currently existing learners is able to learn patterns where both strategies are combined. For example, it is not possible to check whether *the number of accounts with balance larger than 5,000 is larger than 3*, or whether *the average balance of the accounts that have a related transaction involving an amount larger than 1,000 is larger than 5,000.*

In the following three technical chapters, we incorporate such combinations of the two strategies in the ILP context. In Chapt. 3 we provide a theoretical foundation for this combination. Two important problems that occur are that the space of refinements that is considered by an ILP learner substantially expands and that the generality ordering underlying θ-subsumption is violated when applying a refinement operator based on θ-subsumption for generating refinements that combine the two strategies. These problems are discussed in detail and solutions are presented. In Chapt. 4 we incorporate such refinements in an existing ILP learner (in particular, in a decision tree learner). We present two techniques to deal with the enlarged refinement space in the context of that learner. In Chapt. 5 we investigate the use of features that combine the two strategies in predictive functions. For example, in linear equations, the attributes that are used as independent variables may be combinations of the two approaches. For instance, assuming that *Target* is a numeric attribute, we might construct linear equations of the form $Target = 3X_1 + 2X_2$ with X_1 the minimum balance of the savings accounts and X_2 the number of transactions related to accounts with balance exceeding 2,000.

Chapter 3

Combining Aggregates and Selections

3.1 Introduction

As explained in Chapt. 2, in relational learning an individual to be classified may be related to a set of other objects, via one-to-many or many-to-many relationships. Properties of this set, or of some of the objects it contains (or perhaps both) may be relevant for the classification. Among the many approaches to relational learning that currently exist, an important distinction can be made with respect to how they handle these relationships.

Blockeel and Bruynooghe (2003) present a categorization of current approaches. Whereas inductive logic programming (ILP) (Muggleton 1992) is biased towards testing the existence of specific elements in the set, other techniques use aggregate functions, which compute a feature of the set to summarize it. In the context of the *Account* example (Ex. 2.2), an ILP system might predict a person to be positive if, for example, he has an account that is related to a transaction of type *deposit* with an amount higher than 10,000. Systems that use aggregates, on the other hand, could for example predict a person as positive if the sum of the balances on his accounts is higher than 50,000. Examples of the latter approach include certain propositionalization techniques (e.g., (Krogel and Wrobel 2001; Knobbe et al. 2001)), probabilistic relational models (Koller 1999) and relational probability trees (Neville et al. 2003). These methods are optimized for highly non-determinate (e.g. business) domains, whereas ILP is geared more towards structurally complex domains, e.g., molecular biology, language learning, etc.

Current relational learners usually do not combine both approaches, which imposes an undesirable bias on them (Blockeel and Bruynooghe 2003). Such a combination would involve aggregating over a subset of elements fulfilling specific conditions ("aggregating over a selection"). For example, a relevant criterion to determine whether a person is positive could be related to the sum *(aggregation)* of the balances on his savings *(selection)* accounts. Such combinations might be expected to naturally appear in certain patterns, but they are very difficult to construct for machine learning systems, both because they increase significantly the search space and because it becomes more difficult to search it in a structured and efficient way due to, e.g., non-monotonicity (Knobbe et al. 2002). These two issues will be described in detail in this chapter.

In terms of relational algebra, combining both approaches boils down to constructing features of the form $\mathcal{F}(\sigma_C(R))$ with $\mathcal{F}$ an aggregate function, $\sigma_C(R)$ a selection function based on a condition C, and R a set of tuples somehow connected to the tuple we want to classify. From this viewpoint, ILP systems typically build a complicated selection condition C but the aggregate function $\mathcal{F}$ is always the "there exists" function, returning *true* if at least one element of R fulfills C, and *false* otherwise. Other relational learners use features of the form $\mathcal{F}(R)$, where $\mathcal{F}$ is taken from a predefined set of aggregate functions, such as avg, sum,...

Perlich and Provost (2003) present a hierarchy of relational concept classes in order of increasing complexity, where the complexity depends on that of any aggregate functions used. The first class corresponds to concepts where no aggregation is involved, i.e. only one-to-one or many-to-one relationships occur in the data. The second class involves one-to-many relations where the objects contain a number of attributes, each of which can be aggregated independently. The third and fourth class concepts rely on so-called multi-dimensional aggregation, which corresponds to what we call "combining selection and aggregation". Perlich and Provost's analysis points out that ILP-like systems are the only ones that can handle concepts of class 3 or 4, but they have the disadvantage that they typically do not use numeric aggregation, and the latter is identified as a crucial weakness. It is exactly that weakness that we eliminate in this chapter and the next one. In this chapter we lay the basis for combining aggregates with selection conditions; in the next chapter we describe a learning algorithm that builds on this.

We start by discussing related work in Sect. 3.2. In Sect. 3.3 we introduce the concept of complex aggregates in first order logic and provide some terminology and definitions. In Sect. 3.4 we discuss refinement of first order logic clauses with aggregates and analyse the difficulties that arise. We conclude in Sect. 3.5.

3.2 Related work

There exists work that shares our goal of learning hypotheses with aggregates. We can distinguish methods that use a fixed set of aggregates that is defined in advance, and methods that construct the aggregates as part of the learning process.

We first discuss some examples of the former group. Krogel and Wrobel (2001) present a system called RELAGGS (RELational AGGregationS), which builds on transformation based approaches to ILP. RELAGGS computes several joins of the input tables according to their foreign key relationships. These joins are compressed using aggregate functions, specific to the data types of the table columns, such that there remains a single row for each example. The result is an appropriate input for conventional data mining algorithms, such as decision tree induction or support vector machines. Simultaneously, Knobbe et al. (2001) propose the system RollUp, another system that uses aggregates to propositionalize a multi-relational database. The feature sets of RollUp and Relaggs overlap but do not coincide. Krogel and Wrobel (2003) present a comparative evaluation of approaches to propositionalization, where they compare Relaggs to logic-oriented transformation approaches. Other systems employ predefined aggregate functions directly in their relational model representations, without first building a propositionalized table. Relational Probability Trees (RPTs) (Neville et al. 2003) extend standard probability estimation trees to a relational setting. The algorithm for learning an RPT uses aggregation functions to dynamically propositionalize the data. Probabilistic relational models (Koller 1999) use aggregates to specify non-deterministic relations in a dependency structure and in conditional probability tables.

The second group, methods that construct aggregates as part of the learning process, are especially useful when one wants to consider more complex aggregates, where the set to be aggregated over is defined by selection conditions: there may be too many such aggregates to compute and store them all during preprocessing.

Several authors argue in favour of including selection conditions into the aggregate functions. Perlich and Provost (2003) provide a detailed examination of aggregation for relational learning. They define various classes of relational learning problems with respect to aggregation. On their domain of interest the results demonstrate that aggregation operators of higher complexity can significantly improve generalization performance. Blockeel and Bruynooghe (2003) discuss the bias that is imposed on relational learners that either use aggregates or use selections of specific elements and provide some ideas to remove it. One of these ideas is to use relational neural networks (Uwents and Blockeel 2005). Also, Krogel et al. (2003), when comparing logic-oriented and database-oriented methods for propositionalization, conclude that a combination of the features produced by both groups of methods seems a valuable venture.

Unfortunately, finding good hypotheses with complex aggregate conditions is difficult. Besides the fact that the hypothesis space is significantly expanded by allowing complex aggregate conditions, it also becomes more difficult to search this space in a structured way, because the effect of refinements of the aggregate condition on the generality of the hypothesis is not well-understood in general. As a result, all current relational learners are somehow limited with respect to the aggregates they can learn. Krogel and Wrobel (2003) (when presenting an extended version of the system Relaggs) introduce in their propositionalized table aggregate functions that apply not only to single attributes, but also to pairs of attributes, one of which has to be nominal and serves as another *group by* condition, besides the target object identifiers. For example, in terms of the

Account example, the set of features could include the sum of the balances of a person's accounts, grouped by account type. The resulting aggregate conditions are still of limited complexity and are not refined further during the search. Knobbe et al. (2002) propose a method for subsequently specializing the set to be aggregated in the generalized selection graph pattern language. Selection graphs are a graphical description of sets of objects in a multi-relational database. By restricting the application of their specialization operator to aggregate functions where its effect is well-understood (which is a strict subset of all aggregate functions), they can search the hypothesis space in a general-to-specific way, but this obviously limits the kind of complex conditions that can be found. Uwents and Blockeel (2005) describe relational neural networks as a subsymbolic approach towards learning complex aggregates. Their approach is not constrained to using predefined aggregate functions and does not make a distinction between searching for aggregate functions and searching for complex conditions, but the resulting theories are also not interpretable in terms of well-understood aggregates and conditions.

In the following, we will analyse the difficulties that arise when combining aggregation and selection in the context of first order logic.

3.3 Preliminaries

This section introduces the concept of complex aggregate conditions in first order logic and reviews refinement under θ-subsumption and coverage, adapting these concepts to our context where needed.

3.3.1 Complex aggregate conditions in ILP: syntax

The kind of clauses that we are interested in are not pure logical clauses: they may contain aggregate literals, which are defined as follows.

Definition 3.1 (Aggregate literal) *An aggregate literal is a literal of the form $F(V, Q, R)$ where F is an aggregate function, V is an* aggregate variable *occurring in the* aggregate query *Q, and R is the result of applying F to the (multi-)set of all answer substitutions for V that Q results in.*

The aggregate functions we consider in this work are those that result in a numeric value, such as *max, min, avg, sum*, and *count*, and that take one aggregate variable. However, most of the theory extends to other aggregate functions as well (e.g., *mode* (most frequent value) that may result in a nominal value, or *correlation* that takes two aggregate variables).

Typically, the result R is compared to an interval. An aggregate literal together with such a condition is called an aggregate condition.

Definition 3.2 (Aggregate condition, syntactic definition) *An aggregate condition is a pair of literals $(F(V, Q, R), between(R, T_1, T_2))$ where $F(V, Q, R)$*

is an aggregate literal, T_1 and T_2 are threshold values, and between/3 checks whether $R \in [T_1, T_2]$.

Often, the interval used is $[-\infty, T]$ or $[T, \infty]$ and the literal $between(R, T_1, T_2)$ is replaced by $R \vartheta T$, where ϑ is a comparison operator, and T a threshold value. We will use the term *standard clauses* to refer to clauses without aggregate conditions.

Remember from Chapt. 2 that many learning systems rely on a general-to-specific ordering of the hypothesis space, which allows them to traverse this space in an efficient way. For example, if the coverage of a hypothesis h is too low, then all hypotheses that are more specific than h can be pruned from the search space. ILP systems learn hypotheses by learning one clause at a time. Clauses are learnt by starting with the most general clause $h \leftarrow$ and gradually refining it using a refinement operator based on θ-subsumption (see definition 2.4). Such a refinement operator employs one of the following basic operations on a clause:

- apply a substitution to the clause, or
- add a basic language construct (a literal or a set of literals as defined in the language bias) to the body of the clause.

Now assume that the body of the clause contains an aggregate condition. For instance, the clause

$$person(PersId, pos) \leftarrow \\ \quad max(Bal, account(PersId, AccId, Type, Bal), M), M \geq 50000$$

classifies a person as positive if the maximum balance on his accounts is higher than 50,000. Such a clause could now be refined not only by extending the clause itself with a literal (i.e., using refinement under θ-subsumption), but also by extending or applying a substitution to the aggregate query

$account(PersId, AccId, Type, Bal)$.

This way the aggregate literal results in a combination of an aggregate and a selection, what we will call a *complex aggregate literal.*

Definition 3.3 (Complex and simple aggregate literal) *A complex aggregate literal is an aggregate literal where the aggregate query Q is a complex query, i.e., has been refined with selection conditions.*
Similarly, a simple aggregate literal is an aggregate literal where the aggregate query has not been refined.

In terms of relational algebra, a complex aggregate literal would be expressed as $\mathcal{F}(\sigma_C(R))$, with $\mathcal{F}$ an aggregate function, R a set of tuples connected to the tuple under consideration, and $\sigma_C(R)$ a non-empty selection condition on R. A simple aggregate literal would be denoted by $\mathcal{F}(R)$. The definition of complex and simple aggregate conditions follows straightforwardly.

Definition 3.4 (Complex and simple aggregate condition) *A complex aggregate condition is an aggregate condition with a complex aggregate literal. Similarly, a simple aggregate condition is an aggregate condition with a simple aggregate literal.*

For example, the preceding clause could be refined into the following clause, containing a complex aggregate condition (note that *Type* has been instantiated into *savings*),

$$\begin{aligned}&person(PersId, pos) \leftarrow\\&\qquad max(Bal, account(PersId, AccId, savings, Bal), M), M \geq 50000\end{aligned}$$

classifying a person as positive if the maximum balance on his *savings* accounts exceeds 50,000.

3.3.2 Complex aggregate conditions in ILP: semantics

In the preceding section an aggregate condition was defined as a pair of literals $F(V, Q, R), between(R, T_1, T_2)$. While this definition defines an aggregate condition in terms of Prolog literals and can be seen as a syntactic definition, we now provide a more general, semantic, definition of aggregate conditions.

Definition 3.5 (Aggregate condition, semantic definition) *An aggregate condition is a function $c : \mathbb{F} \times \mathbb{S} \times \mathbb{I} \rightarrow \mathbb{B} : F(S) \in I \mapsto B$, with $\mathbb{F}$ a set of aggregate functions, $\mathbb{S}$ a set of multi-sets, $\mathbb{I}$ a set of numeric intervals, and $\mathbb{B}$ the set of boolean values. The value B is true if the condition $F(S) \in I$ holds, and false otherwise.*

In this definition we make abstraction of the fact that S is generated by some query Q and some variable V (see definition 3.2).

3.3.3 θ-subsumption for clauses with aggregates

As we are interested in learning classifiers with complex aggregate literals, it is useful to be able to gradually refine aggregate queries, in a similar way as standard clauses would be refined. We therefore define an extension of the classical θ-subsumption relation on clauses with aggregation, and call it α-subsumption. A *standard literal* is any literal in a clause that is not an aggregate literal and does not occur inside an aggregate query. The *standard part* of a clause c, denoted $S(c)$, is the clause consisting of all standard literals of c (and only those).

Definition 3.6 (α-subsumption) *A clause c_1 α-subsumes a clause c_2 (denoted $c_1 \leq_\alpha c_2$) if and only if $\exists\theta : S(c_1)\theta \subseteq S(c_2)$, and for each aggregate literal $F(V_1, Q_1, R_1) \in c_1$, there exists an aggregate literal $F(V_2, Q_2, R_2) \in c_2$ such that $R_1\theta = R_2$, $Q_1\theta \leq_\alpha Q_2$, and the latter α-subsumption only involves a set of substitutions σ over locally defined variables in Q_1, such that $V_1\sigma = V_2$.*

In other words, a clause α-subsumes another clause if, after applying the right variable substitutions to the standard part of the clause as well as to its aggregate queries, the standard part of the first clause becomes a subset of the standard part of the second clause, and each aggregate query becomes a subset of the corresponding aggregate query in the second clause. Thus, a refinement operator based on α-subsumption refines a clause by employing one of the following basic operations:

- applying a substitution to the clause,
- adding a basic language construct to the body of the clause, or
- applying a substitution or adding a basic language construct to the aggregate query of an aggregate literal in the clause.

These three operations are illustrated in the following example.

Example 3.1 *The clause*

$person(PersId, pos) \leftarrow$
$\quad sum(Bal, account(PersId, AccId, Type, Bal), M), M \geq 50000,$
$\quad account(PersId, AccId2, Type2, Bal2), Bal2 \geq 10000$

stating that a person is positive if the sum of the balances on his accounts exceeds 50,000 and if he owns an account with balance larger than 10,000 can be refined under α-substitution in the following ways:

- $person(PersId, pos) \leftarrow$
  $\quad sum(Bal, account(PersId, AccId, Type, Bal), M), M \geq 50000,$
  $\quad account(PersId, AccId2, savings, Bal2), Bal2 \geq 10000$

  (applying a substitution $\theta = \{Type2/savings\}$ to the clause)

- $person(PersId, pos) \leftarrow$
  $\quad sum(Bal, account(PersId, AccId, Type, Bal), M), M \geq 50000,$
  $\quad account(PersId, AccId2, Type2, Bal2), Bal2 \geq 10000,$
  $\quad transaction(AccId2, TrId, TrType, Am), Am \geq 2000$

  (adding a pair of standard literals (basic language construct) to the clause)

- $person(PersId, pos) \leftarrow$
  $\quad sum(Bal, account(PersId, AccId, Type, Bal), M), M \geq 50000,$
  $\quad account(PersId, AccId2, Type2, Bal2), Bal2 \geq 10000,$
  $\quad count(AccId3, account(PersId, AccId3, Type3, Bal3), C), C \geq 5$

  (adding an aggregate condition to the clause)

- $person(PersId, pos) \leftarrow$
  $\quad sum(Bal, account(PersId, AccId, checkings, Bal), M),$
  $\quad M \geq 50000, account(PersId, AccId2, Type2, Bal2), Bal2 \geq 10000$

  (applying a substitution to the aggregate query)

- $person(PersId, pos) \leftarrow$
  $sum(Bal, (account(PersId, AccId, Type, Bal),$
  $transaction(AccId, TrId, TrType, Am), Am > 2000), M),$
  $M \geq 50000,$
  $account(PersId, AccId2, Type2, Bal2), Bal2 \geq 10000$

  (adding a pair of standard literals (basic language construct) to the aggregate query)

- $person(PersId, pos) \leftarrow$
  $sum(Bal, (account(PersId, AccId, Type, Bal),$
  $count(TrId, transaction(AccId, TrId, TrType, Am),$
  $Cnt), Cnt > 10), M), M \geq 50000,$
  $account(PersId, AccId2, Type2, Bal2), Bal2 \geq 10000$

  (adding an aggregate condition to the aggregate query)

This last refinement illustrates the recursive use of aggregates. It states that the sum of the balances of the accounts that have more than 10 transactions exceeds 50,000.

3.3.4 Specialization and generalization

In Chapt. 2 we have defined the concepts of generalization and specialization of a clause in terms of its coverage (see definition 2.6). Generally, we say that a clause *covers* an example if it covers the example. Clause c_1 is a *specialization* (*generalization*) of c_2 if and only if the set of examples it covers is a subset (superset) of the examples covered by c_2.

For clauses without aggregate literals, it holds that whenever $c_1 \leq_\alpha c_2$ (or equivalently, $c_1 \leq_\theta c_2$), the coverage of c_1 must be a superset of the coverage of c_2, i.e., c_2 is a specialization of c_1. Hence, by using a refinement operator ρ that, given a clause c, yields only clauses θ-subsumed by c, a general-to-specific search through the hypothesis space is obtained. In the next section we will see that for clauses with aggregate literals and the α-subsumption relation, this property is lost: if $c_1 \leq_\alpha c_2$, then, generally, c_2 is not necessarily a specialization of c_1. We will call a refinement *valid* if and only if it constitutes a specialization.

3.4 Refining clauses with complex aggregates

We can now discuss in what way refinement of clauses with aggregate literals is more complicated than refinement of standard clauses.

First, the introduction of complex aggregate conditions into the search space causes a significant increase in size of the hypothesis space. This is discussed in Sect. 3.4.1. Second, while our definition of α-subsumption is syntactically very similar to the original θ-subsumption definition, the property that $c_1 \leq_\alpha c_2$

implies that c_2 is a specialization of c_1, is lost. Refinement under α-subsumption may yield a specialization or a generalization, or even none of both. This behaviour can be related to two issues. On the one hand, in the first order logic context, an aggregate function can be interpreted in several ways when applied to a complex aggregate query (roughly, as being applied to *sets* or to *bags*). When being applied to bags, the number of answer substitutions the aggregate query results in may either increase or decrease. This is discussed in more detail in Sect. 3.4.2. On the other hand, even when using an interpretation that guarantees that the number of answer substitutions decreases, the refinement may still yield a generalization. This behaviour is related to the monotonicity properties of the aggregate conditions, an observation also made by Knobbe et al. (2002) in the context of refining selection graphs. This is discussed in Sect. 3.4.3. The discussion leads to the definition of a refinement operator that guarantees valid refinements.

3.4.1 The search space explodes

ILP systems explore large search spaces. They often do this in a greedy manner: from the current best clause, they generate a number of refinements, take the best among these, and continue the process. The computational complexity of this process depends on the branching factor of the search (how many refinements are generated from a clause).

By introducing aggregates in clauses, and allowing the aggregate queries to be refined as well, the branching factor is multiplied. Assume that a standard clause can be refined in C ways by adding a standard literal to it, and we now also allow the addition of an aggregate literal with any of those C literals as aggregate query, and any variable occurring in the literal as the aggregate variable. If the literal has V variables and we consider N possible aggregate functions, the branching factor increases to $(C + V \cdot N \cdot C)$. The multiplication factor $(1 + V \cdot N)$ can easily be one or two orders of magnitude. Greedy searches slow down with the same factor. If these searches make use of complex aggregate conditions, the branching factor becomes $C + V \cdot N \cdot C + V \cdot N \cdot C \cdot (C + V \cdot N \cdot C)^L$ with L the number of added language elements in the aggregate query.

We illustrate this with an example from the Account dataset.

Example 3.2 *Consider the language bias for the Account example, which is repeated in Table 3.1. If we discard the "+" and "-" symbols, the rmodes give rise to 1, 2, 4, 1, 2, and 3 tests, respectively, resulting in a total of 13 tests. Two of the rmodes result in non-determinate numeric tests (the rmodes that check the balance of the accounts and the amounts of the transactions of a person). By including simple aggregate conditions with the functions max, min, avg, and sum for these two rmodes, the number of tests produced by them is multiplied by 5 (1 for the original rmode, and 4 aggregate functions).*

The rmodes then produce 1, 2, 20, 1, 2, and 15 tests, respectively, yielding 41 tests in total. Now suppose we also consider complex aggregate conditions (say, with only one extra test in the aggregate query that comes from any of the other 5 rmodes), then any of the 16 simple aggregate conditions coming from the third rmode gives rise to $1+2+1+2+15=21$ *complex aggregates and any of the 12 simple aggregates coming from the sixth rmode produces* $1+2+20+1+2=26$ *complex aggregate conditions. This gives a total of* $41+(16*21)+(12*26)=689$ *tests. Including two extra tests in the aggregate query gives rise to* $41+(16*21^2)+(12*26^2)=15{,}209$ *tests.*

Table 3.1: TILDE's language bias for the *Account* example.

```
% types
type(person(persid,age,class)).
type(account(persid,accid,acctype,balance)).
type(transaction(accid,trans,transtype,amount)).

% rmodes: basic language constructs
rmode(account(+PersID, -AccID, -Tp, -Bal)).
rmode(account(+PersID,+-AccID, #["savings","checkings"],-Bal)).
rmode((account(+PersID, +-AccID, -Tp,Bal), Bal ≥ #[500,2000,5000,10000])).
rmode(transaction(+AccID, -Tr, -Tp, -Am)).
rmode(transaction(+AccID, +-Tr,#["deposit","withdrawal"], -Am)).
rmode((transaction(+AccID, +-Tr, -Tp, Am), Am ≥ #[500,1000,2000])).
```

3.4.2 Semantics of the aggregate function

Consider the following clause:

$$\begin{aligned}&person(PersId, pos) \leftarrow \\ &\qquad count(AccId, (account(PersId, AccId, Type, Balance), C), C < 4\end{aligned}$$

with the following refinement, which adds a literal to the aggregate query:

$$\begin{aligned}&person(PersId, pos) \leftarrow \\ &\qquad count(AccId, (account(PersId, AccId, Type, Balance), \\ &\qquad\qquad\qquad transaction(AccId, TransId, TransType, Amount)), \\ &\qquad C), C < 4\end{aligned}$$

The *count* aggregate function counts the number of times the aggregate query succeeds, which, for the refined query, may be larger than the number of accounts if there are multiple transactions per account, or smaller if some accounts have no transactions. In other words, this refinement may lead to generalization or specialization, or none of both. The reason for this is that the *count* function computes the cardinality of the bag, rather than the set, of *AccId* values returned by the aggregate query. While the set of *AccId* values returned by the refined query must be a subset of the set returned by the original one, the bag of *AccId* values returned by the refined query is not guaranteed to be a subbag or superbag of the original one: some accounts may have disappeared, others may have been duplicated.

The situation is similar to what one would get with an SQL query (for relational databases) along the lines of

```
SELECT COUNT(A.AccId)
FROM account AS A, transaction AS T
WHERE A.AccId=T.AccId
```

A solution in the relational database case is to use the COUNT DISTINCT construct. This is semantically meaningful if *AccId* is a key attribute for the *Account* relation.

In general, there are thus two possible outcomes for an aggregate function: applying the function to the bag or to the set of variable substitutions returned by the aggregate query. For example, consider the simple extension of the *Account* database in Table 2.2. The query

$$sum(Balance, (account(PersId, AccId, Type, Balance),\\ transaction(AccId, TransId, TransType, Amount)), C)$$

leads to two possible results for the first example (john):

- $100 + 100 + 200 + 200 + 200 = 800$ (when aggregating over the bag of balances)
- $100 + 200 = 300$ (when aggregating over the set of balances).

However, taking the set over the balance values is often not intuitive. Instead, one most likely wants to take the set of the account objects and then take the sum of the balances (i.e., taking the sum of the balances of the accounts that have a transaction). Thus, if two different accounts have the same balance, this value should be counted twice. More generally, this corresponds to taking the set over the first predicate in the aggregate query, and applying the aggregate function to the corresponding aggregate variable values. This leads to a third possible result:

- $100 + 200 + 200 = 500$ (when aggregating over the set of accounts)

These solutions correspond to the following queries in relational algebra (where $\star$ denotes the natural join operator):

- $\mathcal{F}_{SUM(Balance)}(account \star transaction)$
- $\mathcal{F}_{SUM(Balance)}(\pi_{Balance}(account \star transaction))$

- $\mathcal{F}_{SUM(Balance)}(\pi_{AccountId,Balance}(account \star transaction))$

In the following chapters, we will apply these aggregate conditions in an existing ILP system. Therefore, we need to decide on what semantics we will use. Since the third interpretation is the most intuitive, in the sense that it is what one would expect when reading an aggregate condition ("the sum of the balances of the accounts that have a transaction"), we will use this interpretation. This is consistent with the way of executing aggregates by Knobbe et al. (2002). However, since the first interpretation is the way Prolog executes aggregate conditions, we will also provide this semantics. It will be denoted by the suffix *_bag* added to the aggregate function. Thus, we introduce the following aggregate functions: *count_bag, avg_bag*, and *sum_bag.* As the minimum (maximum) of a set is the same as the minimum (maximum) of a bag defined over the set, there is no need to have a *min_bag* (*max_bag*). The second interpretation is in most cases not intuitive (note that when aggregating over a key attribute, the second and third interpretation are the same). The only aggregate function where this second interpretation may be useful is *count.* We will denote it as *count_distinct*, which corresponds to the COUNT DISTINCT aggregate function in SQL.

We will now formalize the concepts of bag-, set-, and object-defined aggregate conditions. Therefore, we introduce the notion of *result bag.*

Definition 3.7 (Result bag, result set, object result bag) *The result bag defined by a query Q and a variable V is the bag of all answer substitutions for V that Q results in.*
The result set defined by a query Q and a variable V is the set of all answer substitutions for V that Q results in.
The object result bag defined by a query $Q = (SimplePart, ComplexSelections)$ and a variable V that occurs in SimplePart is the bag of all answer substitutions for V taken from the set of all answer substitutions for SimplePart that Q results in.

The last type of result bag is called *object* result bag, because in a sense, the aggregate is taken over *objects* (represented by the simple part of Q, i.e., the part of Q that corresponds to a simple aggregate literal[1]), as opposed to the result bag and result set, where the aggregate only considers *values.* The following example illustrates the computation of an object result bag.

Example 3.3 *Consider again the aggregate condition*

$$sum(Balance, (account(PersId, AccId, Type, Balance), \\ transaction(AccId, TrId, TrType, Amount)), C)$$

[1]We assume the simple part of Q can be unambiguously determined, in this text it is assumed to be the first literal in Q.

and the extension of the Account dataset in Table 2.2. In order to compute the object result bag for the first example ("john"), we first obtain the set of answer substitutions for $account(PersId, AccId, Type, Balance)$ *yielded by the aggregate query* $(account(PersId, AccId, Type, Balance), transaction(AccId, TrId, TrType, Amount))$. *This is the set* $\{account(john, 123456, checkings, 100), account(john, 987654, checkings, 200), account(john, 789123, savings, 200)\}$ *(note that the last account of john has no transactions related). Then, for each element in S, we take the corresponding value for Balance. This results in the object result bag* $\{100, 200, 200\}$.

The following property holds for any query Q and variable V occurring in Q: $\text{result\_set}(Q, V) \subseteq \text{object\_result\_bag}(Q, V) \subseteq \text{result\_bag}(Q, V)$. We can now define bag-, set-, and object-defined aggregate conditions.

Definition 3.8 (Bag-, set-, and object-defined aggregate conditions) *A bag-defined aggregate condition is an aggregate condition with an aggregate literal* $F\_bag(V, Q, R)$ *where R is the result of applying* F_bag *to the result bag defined by Q and V.*
A set-defined aggregate condition is an aggregate condition with an aggregate literal $F\_distinct(V, Q, R)$ *where R is the result of applying* $F\_distinct$ *to the result set defined by Q and V.*
An object-defined aggregate condition is an aggregate condition with an aggregate literal $F(V, Q, R)$ *where R is the result of applying F to the object result bag defined by Q and V.*

For completeness, we show how the result of an aggregate literal would be computed in Prolog for the three different semantics:

```
agg_bag(Var, Query, Result) :-
        findall(Var, Query, ResultBag),
        agg(ResultBag, Result).

agg_distinct(Var, Query, Result) :-
        setof(Var, Query^Query, ResultSet),
        agg(ResultSet, Result).

agg(Var, (Simple, Complex), Result) :-!,
        findall(Var, (Simple, once(Complex)), ObjectResultBag),
        agg(ObjectResultBag, Result).
agg(Var, Simple, Result) :-
        findall(Var, Simple, ObjectResultBag),
        agg(ObjectResultBag, Result).
```

In these clauses, *agg* denotes an aggregate function (*max*, *min*, *avg*, *sum*, *count*, *mode*).

3.4.3 Monotonicity

In the previous section we have discussed how the bag semantics of an aggregate condition may violate the property that $c_1 \leq_\alpha c_2$ implies that c_2 is a specialization of c_1. Refining bag-defined aggregate conditions under α-subsumption may yield a specialization or a generalization, or even none of both, because the refinement may increase or decrease the result bag. Refining object- or set-defined aggregate conditions guarantees that the corresponding object result bag or result set decreases. However, even with a decreased result set, the refinement may still yield a generalization instead of a specialization. This behaviour is related to the monotonicity properties of the aggregate condition, which is the subject of this section.

Take the following example clause:

$$\begin{aligned}&person(PersId, pos) \leftarrow\\&\qquad count(AccId, account(PersId, AccId, Type, Balance), C), C \geq 4\end{aligned}$$

Applying a substitution to the aggregate query can yield the following refinement

$$\begin{aligned}&person(PersId, pos) \leftarrow\\&\qquad count(AccId, account(PersId, AccId, savings, Balance), C), C \geq 4\end{aligned}$$

which must have at most the same coverage (the set of people with at least four savings accounts must be a subset of the people with at least four accounts), so the refinement is valid.
However, if we consider the following query

$$\begin{aligned}&person(PersId, pos) \leftarrow\\&\qquad count(AccId, account(PersId, AccId, Type, Balance), C), C < 4\end{aligned}$$

and its refinement

$$\begin{aligned}&person(PersId, pos) \leftarrow\\&\qquad count(AccId, account(PersId, AccId, savings, Balance), C), C < 4\end{aligned}$$

then the result yields a generalization (the set of people with less than four savings accounts is a superset of the people with more than four accounts).

This example shows that refinement under α-subsumption, although applied to object-defined aggregate conditions, does not guarantee valid refinements. The fact that the refinement operator yields a specialization in the first case, and a generalization in the second case, is related to the aggregate condition, which is monotone in the first example and anti-monotone in the second. In the following sections we present a refinement operator that does guarantee valid refinements for aggregate conditions. To this end, we first introduce the concept of monotonicity (Sect. 3.4.3.1). Then we describe an ordering on the aggregate functions (Sect. 3.4.3.2) in order to be able to investigate the monotonicity properties of an aggregate condition (Sect. 3.4.3.3). Afterwards, we discuss valid refinements of aggregate conditions (Sect. 3.4.3.4) and formalize the new refinement operator by means of a visualization of the valid refinements (Sect. 3.4.3.5). Finally, we briefly present a related work (Sect. 3.4.3.6).

3.4.3.1 Monotonicity

Definition 3.9 (Monotonicity) *Given a function $f(x_1, ..., x_n)$: $(D_1, ..., D_n) \rightarrow R$, an order relation $\leq_{D_i}$ on each domain D_i and an order relation $\leq_R$ on R. The function f is*

- monotone *in x_i iff*
  $x_i \leq_{D_i} x_{i'} \Rightarrow f(x_1, ..., x_i, ..., x_n) \leq_R f(x_1, ..., x_{i'}, ..., x_n)$,
- anti-monotone *in x_i iff*
  $x_i \leq_{D_i} x_{i'} \Rightarrow f(x_1, ..., x_i, ..., x_n) \geq_R f(x_1, ..., x_{i'}, ..., x_n)$,
- *and* non-monotone *in x_i otherwise.*

In other words, a function mapping an ordered domain onto an ordered range is monotone if it preserves the order, anti-monotone if it reverses the order, and non-monotone otherwise.

To investigate the monotonicity properties of an aggregate condition $F(S) \in I$, an order relation is needed on each of the domains and on the range. We define these relations as follows:

- $\mathbb{F}$, the set of aggregate functions:
  - $F_1 \preceq_{\mathbb{F}} F_2 \Leftrightarrow \forall S \in \mathbb{S} : F_1(S) \leq F_2(S)$, this is discussed in the next section,
- $\mathbb{S}$, the set of bags (multi-sets)
  - $S_1 \preceq_{\mathbb{S}} S_2 \Leftrightarrow S_1 \subseteq S_2$, we say that A is a subbag of B (denoted $A \subseteq B$) if and only if each element of A is also in B and its multiplicity in B is at least as high as in A,
- $\mathbb{I}$, the intervals defined over $\mathbb{R}$
  - $I_1 \preceq_{\mathbb{I}} I_2 \Leftrightarrow I_1 \subseteq I_2$,
- $\mathbb{B}$, the boolean values
  - *false* $\preceq_{\mathbb{B}}$ *true.*

3.4.3.2 Ordering the aggregate functions

In this section we discuss several parameterized classes of aggregate functions that are ordered and together cover all aggregates of interest. As before, the aggregates of interest are those that result in numeric attributes, i.e., *max*, *min*, *avg*, *sum*, and *count*. The interpretation used for these aggregate functions (bag, set or object semantics) does not influence the ordering relation, unless stated otherwise.

Generalized averages. We define a class of generalized averages as follows:

Definition 3.10 (Generalized average)

$$avg_k(S) = (\frac{\sum_i(x_i^k)}{n})^{1/k} \quad \textit{with } S = \{x_1, ..., x_n\}$$

The function $avg_k(S)$ is defined for $-\infty \leq k \leq \infty$ $(k \neq 0)$ if $S \subseteq \mathbb{R}^+$ and for $k = \{1, -1, \infty, -\infty, 2 * z\}$ with $z \in \mathbb{Z}$ if $S \subseteq \mathbb{R}$. If $S \subseteq \mathbb{R}^+$, then the following order relation holds:

$$i \leq j \Rightarrow avg_i \preceq_{\mathbb{F}} avg_j.$$

While this relation holds for all k, only some of these k-values are commonly used: $avg_1(S) = avg(S)$, $\lim_{k \to \infty} avg_k(S) = max(S)$, and $\lim_{k \to -\infty} avg_k(S) = min(S)$. Moreover, the order relation $min \preceq_{\mathbb{F}} avg \preceq_{\mathbb{F}} max$ also holds for sets S that contain negative numbers.

Generalized sums. For *sum* we can define an aggregate function class very similar to the generalized averages:

Definition 3.11 (Generalized sum)

$$sum_k(S) = (\sum_i(x_i^k))^{1/k} \quad \textit{with } 1 \leq k \leq \infty \textit{ and } S = \{x_1, ..., x_n\}$$

We have that $sum_1(S) = sum(S)$ and $\lim_{k \to \infty} sum_k(S) = max(S)$. In other words, these generalized sums range from *max* to *sum*. If S contains only positive numbers, we obtain the following order relation:

$$i \geq j \Rightarrow sum_i \preceq_{\mathbb{F}} sum_j.$$

Generalized counts. Our last aggregate function of interest is *count*. An aggregate function that can form an aggregate class with *count* is *count distinct*. This function counts the number of *different* values in the multi-set. We have the following order relation:

$$count\_distinct \preceq_{\mathbb{F}} count.$$

Note that these two functions correspond to different semantics of the *count* function.

Summary. While the proposed aggregate classes contain an infinite amount of aggregate functions, for the most important ones, we obtain the following order:

- $\mathit{min} \preceq_{\mathbb{F}} \mathit{avg} \preceq_{\mathbb{F}} \mathit{max}$,
- $\mathit{max} \preceq_{\mathbb{F}} \mathit{sum}$ if $S \subseteq \mathbb{R}^+$,
- $\mathit{count\_distinct} \preceq_{\mathbb{F}} \mathit{count}$.

Remark that there exist other parameterized classes for these aggregate functions.

3.4.3.3 Monotonicity properties of aggregate conditions

Having defined an ordering relation on the domains $\mathbb{F}$, $\mathbb{S}$, $\mathbb{I}$, and $\mathbb{B}$ of an aggregate condition, we can discuss the monotonicity properties of each domain.

For the ease of explanation, we consider an aggregate condition $F(S) \in I$ to be the composition of two functions:

- an aggregate function $a : \mathbb{F} \times \mathbb{S} \rightarrow \mathbb{R}$,
- a member function $m : \mathbb{R} \times \mathbb{I} \rightarrow \mathbb{B}$.

For each of these two functions, we can now describe the monotonicity properties. Afterwards, we describe some issues that come along when composing the two functions. The monotonicity properties are summarized in Fig. 3.1.

Monotonicity of the aggregate function. The function $a(F, S)$ is monotone in F, because the order on $\mathbb{F}$ is defined as such.

The monotonicity of $a(F, S)$ w.r.t. S depends on F. The function $a(F, S)$ is monotone in S if $F \in \{\mathit{count}, \mathit{count\_distinct}, \mathit{max}\}$ (if any of these functions is applied to a subset (subbag) $S' \subseteq S$, its resulting value will decrease). Similarly, $a(\mathit{min}, S)$ is anti-monotone, and $a(\mathit{sum}, S)$ and $a(\mathit{avg}, S)$ are non-monotone in S[2].

Monotonicity of the member function. The member function $m(R, I)$ is monotone in I: decreasing the interval can cease the membership of R.

The monotonicity in R depends on I: $m(R, [v, \infty[)$ (with $v \in \mathbb{R}$) is monotone in R, $m(R,]\text{-}\infty, v])$ anti-monotone, and $m(R, [v, w])$ $(v, w \in \mathbb{R})$ non-monotone.

[2]For *sum* the monotonicity depends on the set S, e.g., if this set contains only positive numbers then $a(\mathit{sum}, S)$ is monotone.

Monotonicity of the composite function. Care is needed when composing the aggregate function and the member function. The monotonicity properties in F and S are inherited by $m(a(F,S),I)$ if this function is monotone in $a(F,S)$. However, the monotonicity in F and S is reversed if the composed function is anti-monotone in $a(F,S)$. For example, a condition $max(S) \in]\text{-}\infty, v]$ is anti-monotone in max and S.

Similarly, when the composed function is non-monotone in $a(F,S)$, the monotonicity properties in F and S are broken. For instance, for a condition $max(S) \in [v,w]$, the monotonicity properties in max and S are lost. Therefore, in the following we do not consider intervals of the form $[v,w]$, with $v,w \in \mathbb{R}$. We only consider the aggregate conditions $F(S) \leq v$ and $F(S) \geq v$.

3.4.3.4 Valid refinements

In order to define valid refinements for an aggregate condition, we can use the monotonicity properties described above. Keeping in mind that the goal is to obtain more specific conditions (i.e. refinements that make the condition false for some of the examples for which it was true), definition 3.9 learns that a function that is monotone (anti-monotone) in one of its inputs can be validly refined by decreasing (increasing) its value for that input.

Before turning to an example, we explain how a multi-set S can be increased or decreased in an ILP system.

Increasing or decreasing S. An aggregate condition that is monotone in S (e.g., $count(S) \geq v$) can be validly refined by decreasing S, i.e., reducing the multi-set to be aggregated. In ILP, this can be achieved by specializing the query Q used to generate S, if object or set semantics is used.

An aggregate condition that is anti-monotone in S (e.g., $count(S) \leq v$) can be validly refined by increasing S, i.e., enlarging the set to be aggregated. If object or set semantics is used, this can be obtained in ILP by generalizing the query that is used to generate the set, thus, by removing literals from it.

For bag-defined aggregate functions there is no general strategy for increasing or decreasing S. To refine bag-defined aggregate conditions in a valid way, only the aggregate function or the interval can be changed.

Example 3.4 *We now illustrate the possible refinements with a small example from the* Account *database. Consider the following clause:*

$$\begin{aligned} &person(PersId, pos) \leftarrow \\ &\qquad max(Bal, account(PersId, AccId, Type, Bal), M), M \geq 15000 \end{aligned}$$

The aggregate condition $F(S) \in I$ in this clause can be refined in three ways:

- *decrease I (because the member function $m(R,I)$ is monotone in I)*

$$max(Bal, account(PersId, AccId, Type, Bal), M), M \geq 20000$$

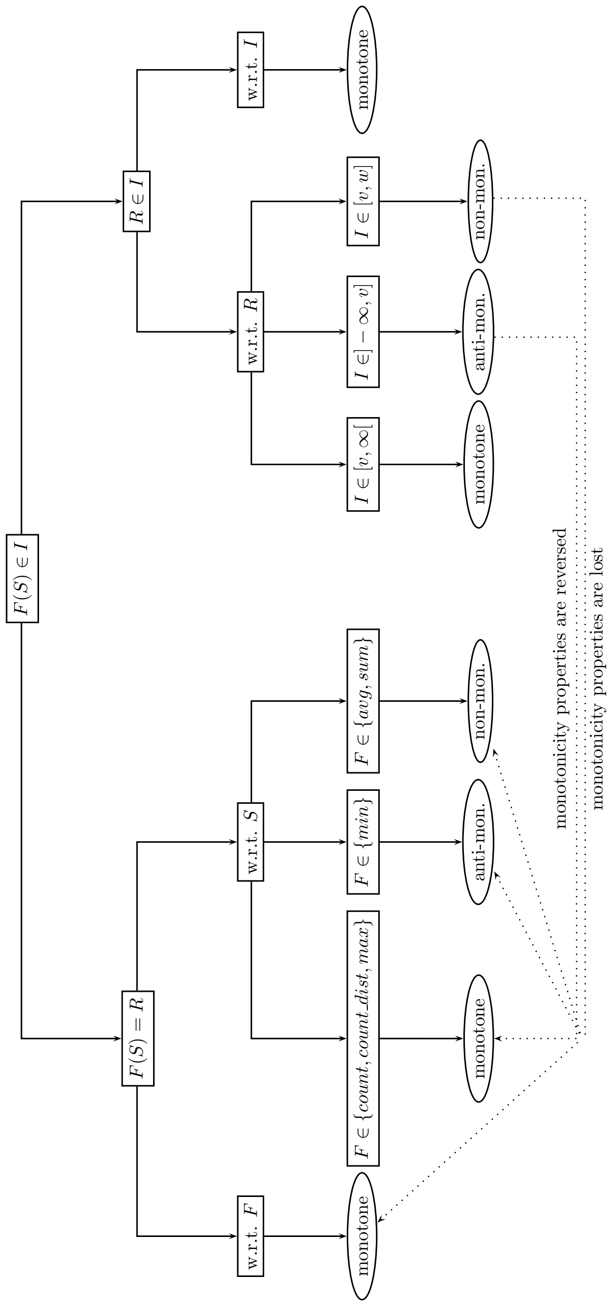


Figure 3.1: Monotonicity Properties of Aggregate Conditions.

- *decrease $max(S)$ (because for $I = [v, \infty[$ the member function $m(R, I)$ is monotone in R). This can be achieved by*
  - *decreasing max (because the aggregate function $a(F, S)$ is monotone in F)*

    $$avg(Bal, account(PersId, AccId, Type, Bal), M), M \geq 15000$$

  - *decreasing S (because $a(max, S)$ is monotone in S).*

    $$max(Bal, account(PersId, AccId, savings, Bal), M), M \geq 15000$$

In the next section, we present a visualization of the valid refinements and give a formal definition of the refinement operator that follows from this discussion.

3.4.3.5 Refinement cubes

We have presented three dimensions along which aggregate conditions can be refined: the aggregate function F, the query Q (or equivalently, the multi-set S_Q, consisting of all answer substitutions generated by Q[3]), and the interval bound v. If object or set semantics is used for the aggregate functions, the whole set of hypotheses spanned by these three dimensions can be visualized in what we call a *refinement cube* (see Figure 3.2). Every discrete point in the cube represents a hypothesis and can be constructed in a finite number of steps, starting from one aggregate condition. A chain of refinements in the cube will be called a *path*. For simplicity, we only consider refinements along one direction at a time. We are only interested in valid refinements, therefore we only allow *monotone paths* in the refinement cubes, i.e. paths that consist only of valid refinements.

For a given aggregate function class, refinement of aggregate conditions proceeds as follows. For every numeric attribute A, we look for a query Q that generates the set of values S_Q for each example. We take the smallest and largest aggregate function in the class (F_{small} and F_{large} respectively) and look for the smallest possible value returned by $F_{small}(S_Q)$ and the largest possible value returned by $F_{large}(S_Q)$ (V_{small} and V_{large} respectively). Then we construct two *start conditions*: $F_{large}(S_Q) \geq V_{small}$ and $F_{small}(S_Q) \leq V_{large}$. For each aggregate function class and each start condition there is a corresponding refinement cube that shows the allowed refinements.

The refinement cubes for the generalized averages. The generalized averages range from *min* to *max*. Possible thresholds for these functions range from the minimum to the maximum value in the dataset for the attribute under consideration. Hence, the two start conditions for this aggregate class are

[3]In this section, we make abstraction of the variable in Q for which the answer substitutions are generated.

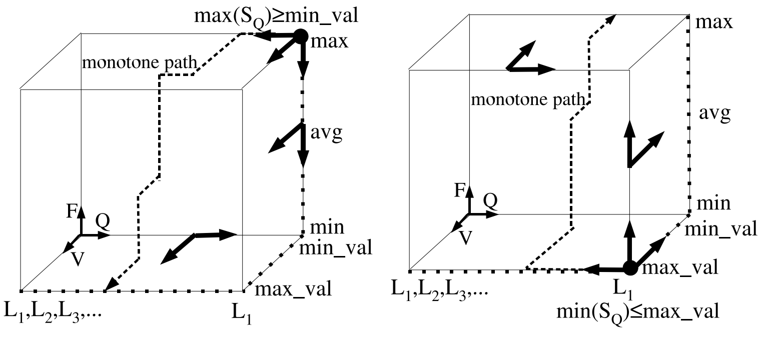


Figure 3.2: The refinement cubes for the generalized averages. (a) Refinement cube for start condition $max(S_Q) \geq min\_value$. (b) Refinement cube for start condition $min(S_Q) \leq max\_value$.

$max(S_Q) \geq min\_value$ and $min(S_Q) \leq max\_value$. The corresponding refinement cubes are shown in Figure 3.2, with the start conditions indicated as a large dot. The arrows show the directions in which we can generate monotone paths starting from these aggregate conditions. Observe that when moving along the F-axis, the monotonicity properties of the aggregate function in S_Q change, so moving along the Q axis is only allowed in the top or bottom faces of the cube.

The refinement cubes for the generalized sums. Figure 3.3 shows the refinement cubes for the generalized sums. The V-axis ranges from the lowest value in the range of *max* (the minimal value for the numeric attribute) to the largest value in the range of *sum* (the maximum of the sum of the values for the attribute A, grouped by example, this value is called *sum_value* in the cube). The start conditions from which we can generate the whole cube are thus $sum(S_Q) \geq min\_value$ and $max(S_Q) \leq sum\_value$. The second start condition is anti-monotone in S_Q, and therefore positioned at the *specific* side of Q. Remark that, if we would use bag semantics for F, it would not be possible to determine this second start condition, since *sum_value* would be unknown.

In this case moving along the F-axis does not change monotonicity (under the assumption that the generalized sums are only applied to sets of positive numbers), so the previous restriction does not apply here.

The refinement cubes for the generalized sums can be connected to the cubes for the generalized averages. This results in a combined refinement

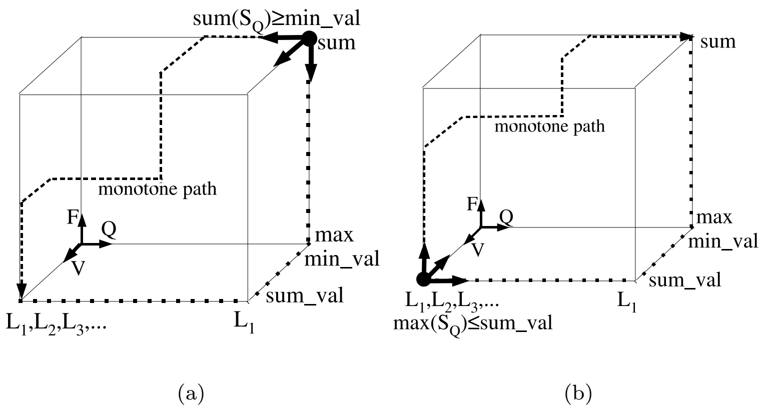


Figure 3.3: The refinement cubes for the generalized sums. (a) Refinement cube for start condition $sum(S_Q) \geq min\_value$. (b) Refinement cube for start condition $max(S_Q) \leq sum\_value$.

space (see Figure 3.4). The start condition for the $\geq$ operator is $sum(S_Q) \geq min\_value$ (Figure 3.4(a)). For the $\leq$ operator the start condition is $min(S_Q) \leq sum\_value$ and is located outside the original cubes. It can be replaced by the two other start conditions indicated in Figure 3.4(b).

The refinement cubes for the generalized counts. For the generalized counts the F-axis only contains the functions *count* and *count_distinct*. The V-axis ranges from 0 to the maximum size of the set generated by Q (*cnt_value*). The start conditions are $count(S_Q) \geq 0$ and $count\_dist(S_Q) \leq cnt\_value$. The monotone paths in the refinement cubes are the same as those for the generalized sums (see Figure 3.3).

We now illustrate the use of the refinement cubes with an example.

Example 3.5 *Consider the following start condition for the generalized sums:*

$$person(PersId, pos) \leftarrow \\ \quad sum(Bal, account(PersId, AccId, Type, Bal), S), S \geq 10000$$

Suppose we only use the functions sum and max, only use the threshold values 10,000 and 15,000, and only specialize the aggregate query in $account(PersId, AccId, savings, Bal)$. *Figure 3.5 schematically shows the refinements that are generated. Note that an aggregate condition can be obtained via more than one path, so in practice one has to take care to generate the conditions only once.*

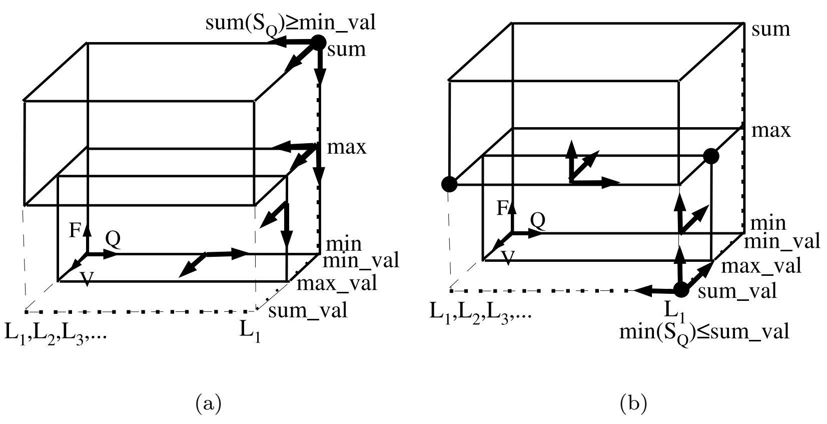


Figure 3.4: The refinement cubes for the generalized sums in combination with the generalized averages. (a) Refinement cube for start condition $sum(S_Q) \geq min\_value$. (b) Refinement cube for start condition $min(S_Q) \leq sum\_value$.

Refinement operator. We now give a formal definition of the refinement operator that follows monotone paths in the refinement cubes. We therefore introduce the notion of μ-subsumption.

Definition 3.12 (μ-subsumption) *A clause c_1 μ-subsumes a clause c_2 (denoted $c_1 \leq_\mu c_2$) if and only if $S(c_1) \leq_\theta S(c_2)$, and for each aggregate condition $(F_1(V_1, Q_1, R_1), between(R_1, T_{11}, T_{12})) \in c_1$, there exists an aggregate condition $(F_2(V_2, Q_2, R_2), between(R_2, T_{21}, T_{22})) \in c_2$ such that $R_1\theta = R_2$, the tuple $(F_2, Q_2, T_{21}, T_{22})$ is obtained from $(F_1, Q_1\sigma, T_{11}, T_{12})$ via a monotone path in the corresponding refinement cube, $V_1\sigma = V_2$, and the substitution σ consists of θ augmented with a substitution over locally defined variables in Q_1.*

In other words, a refinement operator based on μ-subsumption refines a clause by applying one of the following basic operations:

- applying a substitution to the clause,
- adding a basic language construct to the body of the clause, or
- replacing an aggregate condition in the clause by an other aggregate condition, obtained via a monotone path in the refinement cube corresponding to the aggregate condition.

Contrary to α-subsumption, μ-subsumption implies generality:

Theorem 3.1 *If $C_1 \leq_\mu C_2$ then C_1 is at least as general as C_2.*

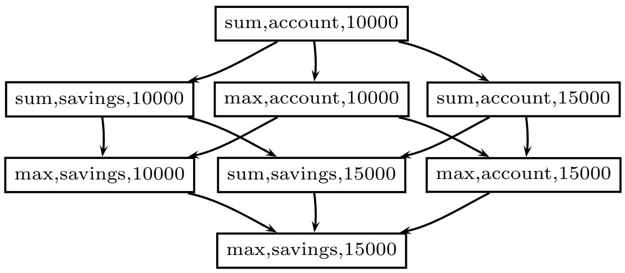


Figure 3.5: Refinements generated for the start condition $sum(Bal, account(PersId, AccId, Type, Bal), S), S \geq 10000$ by the cube for the generalized sums. In each node, the function F, query Q, and threshold value v are shown. The query is abbreviated: *account* stands for $account(PersId, AccId, Type, Bal)$ and *savings* means $account(PersId, AccId, savings, Bal)$.

When using a refinement operator based on μ-subsumption we obtain a search strategy that is

- *efficient*: only valid refinements are generated in each step, which allows to prune the aggregate search space,
- *complete*: every aggregate condition is reachable in a finite number of steps starting from the start conditions.

Note that this efficiency can only be achieved by considering all three dimensions F, Q, and V together. A system that does not allow refinements along the F-axis (e.g., $max \rightarrow avg \rightarrow min$) can not obtain the aggregate conditions in a monotone (general-to-specific) way. For example, the condition $avg(Bal, account(PersId, AccId, savings, Bal), S), S \geq 10000$ can only be obtained in a monotone way via the path $max(Bal, account(PersId, AccId, Type, Bal), S), S \geq 10000 \rightarrow max(Bal, account(PersId, AccId, savings, Bal), S), S \geq 10000 \rightarrow avg(Bal, account(PersId, AccId, savings, Bal), S), S \geq 10000$.

Remark that a refinement based on α-subsumption corresponds to moving along the Q-axis in the general to specific direction.

3.4.3.6 Related work

The presented work is to some extent related to the research started by Ng et al. (1998) on constraint based mining. In that work, the task is to find frequent itemsets that fulfill constraints that possibly involve aggregations. For instance, in addition to minimal support, one can impose that the total price of the items in an itemset must be below some threshold. The set of frequent itemsets that fulfill the constraints is constructed by starting with frequent itemsets that contain only one element and subsequently combining frequent itemsets. The research also involves studies of the monotonicity properties of aggregates and the fact that the aggregate condition $sum(S) \leq T$ is anti-monotone in the set S (assuming S contains only positive numbers, which is the case with prices) is exploited as follows. If an itemset $\{X, Y\}$ violates the total price constraint, then all itemsets that are extensions of $\{X, Y\}$ can be pruned. Thus, in that work, a pattern without aggregates can be pruned on the basis of aggregate constraints.

In the context of knowledge representation, Pelov (2004) studies the monotonicity properties of aggregate conditions when defining semantics of logic programs that contain recursion over aggregate atoms. An example of such a program is the following clause, which states that a person accepts an invitation for some party if at least three of his friends accept the invitation.

$$\begin{aligned}&accept\_invitation(Person) \leftarrow \\ &\quad count(Friend, (friend(Person, Friend), accept\_invitation(Friend)), \\ &\qquad NbFriends), NbFriends \geq 3\end{aligned}$$

The work presents extensions of several semantics of logic programming (including least fixpoint semantics, well-founded semantics,...) to include such aggregate expressions.

3.5 Conclusion

In this chapter we have discussed the problem of combining aggregates and selections. Such combinations might be expected to naturally appear in certain patterns, but they are very difficult to construct for machine learning systems, both because the feature space explodes and because it becomes more difficult to search it in a structured and efficient way. In this chapter we have analyzed these issues.

First, we have introduced the notion of complex aggregate condition and have presented a refinement operator based on α-subsumption, which is a trivial extension of a θ-subsumption based operator to the context of complex aggregate conditions. We have shown that refining an aggregate condition under α-subsumption causes the result set or the object result bag of the aggregate query to decrease (to become a subset of what it originally was), but may have any effect on the (normal) result bag of the aggregate query. Hence, for the object- or set-defined aggregate conditions that are (anti-)monotone in the set to aggregate on we can say that refining an aggregate query can only yield a specialization (generalization), but for the bag-defined aggregate conditions no such statements are possible. A consequence of this is that searching the hypothesis space in a general-to-specific manner becomes more complicated.

There is no obvious refinement strategy for aggregate queries that guarantees that the refinement will yield a specialization, unless we limit the hypothesis space to patterns involving refinements of monotone aggregate conditions, and use the object semantics. This is essentially what Knobbe et al. (2002) do.

However, instead of looking for a refinement strategy for aggregate queries, one can solve the problem from the level of aggregate conditions which have, besides aggregate queries, other parameters (the aggregate function and the threshold value). No current relational learners consider refinements on all three dimensions together, and indeed the effect of such refinements on the generality of a rule, and the interaction between these effects, are non-trivial. Therefore, we have presented an in-depth study of the monotonicity of aggregate conditions along the following dimensions: the aggregate function, the set to be aggregated, and the threshold value. This first dimension has never been explored before, but turns out to be crucial to obtain an efficient refinement strategy for aggregate conditions. This study has led to the notion of μ-subsumption and an associated refinement operator. The presented refinement operator can be beneficial for any relational learning system that learns aggregates and makes use of a general-to-specific ordering of the hypotheses to guide the search (e.g., decision tree learners, rule learners, frequent pattern miners,...).

Chapter 4

Adding Complex Aggregates to First Order Decision Trees

4.1 Introduction

In the previous chapter we have explained that current relational learners handle sets either by testing for the existence of specific elements or by summarizing them using aggregate functions. The fact that no system is able to fully combine both approaches, causes an undesirable bias on current learners. Therefore, we have introduced what we call *complex aggregate conditions*. These are aggregate conditions where the set to be aggregated may be generated by a complex query, thus, may contain a number of selection conditions.

In this chapter we include these complex aggregate conditions in an ILP system that learns first order decision trees. While rule induction is more common in ILP than tree induction, ILP tree learners have been around for several years now. The system that we will use is Tilde(Blockeel and De Raedt 1998). Although we focus on decision trees, the techniques discussed in this chapter are also applicable to learning algorithms for other patterns.

The chapter is organized as follows. First, we describe how first order decision trees are built in Tilde(Sect. 4.2). Then, we discuss how complex aggregate conditions can be included in the search space of first order decision trees (Sect. 4.3). In Sect. 4.4 we discuss two techniques to deal with the feature space explosion. Section 4.5 proposes a number of approaches to deal with aggregating over empty sets. Experiments are presented in Sect. 4.6. We conclude in Sect. 4.7.

4.2 Tilde: a first order decision tree learner

The system Tilde (Blockeel and De Raedt 1998) is included in the ACE-ilProlog data mining system (ACE 2004; Blockeel et al. 2006). It is a relational top-down induction of decision trees (TDIDT) instantiation, and outputs a first order decision tree.

A first order decision tree (Blockeel and De Raedt 1998) is a binary decision tree that contains conjunctions of first order literals in the internal nodes.

Table 4.1: TILDE algorithm for first order logical decision tree induction (Blockeel and De Raedt 1998).

```
procedure GROW_TREE ($E$: examples, $Q$: query):
    $candidates := \rho(Q)$
    $Q_b :=$ OPTIMAL_SPLIT$(candidates, E)$
    if STOP_CRIT $(Q_b, E)$
    then
        $K :=$ PREDICT$(E)$
        return leaf$(K)$
    else
        $conj := Q_b - Q$
        $E_1 := \{e \in E | Q_b$ succeeds in $e \wedge B\}$
        $E_2 := \{e \in E | Q_b$ fails in $e \wedge B\}$
        $left :=$ GROW_TREE $(E_1, Q_b)$
        $right :=$ GROW_TREE $(E_2, Q)$
        return node$(conj, left, right)$
```

Classification with a first order tree is similar to classification with a propositional decision tree: a new instance is sorted down the tree. If the conjunction in a given node succeeds (fails), the instance is propagated to the left (right) subtree. The predicted class corresponds to the label of the leaf node where the instance arrives. A given node n of the tree may introduce variables that can be reused in the nodes of its left subtree, which contains the examples for which the conjunction in n succeeds (with certain bindings for these variables).

In TILDE, first order decision trees are learned with a divide and conquer algorithm similar to C4.5 (Quinlan 1993). The main point where it differs from propositional tree learners is the computation of the set of tests to be considered at a node. The algorithm to learn a first order decision tree is given in Table 4.1.

The OPTIMAL_SPLIT procedure returns a query Q_b, which is selected from a set of candidates generated by the refinement operator ρ, by using a heuristic, such as information gain for classification problems, or variance reduction for regression. The refinement operator operates under θ-subsumption and generates candidates by extending the current query Q (the conjunction of all succeeding tests from the root to the leaf that is to be extended) with a number of new literals that are specified in the *rmode* constructs of the language bias (see Ex. 2.12 for more details about TILDE's language bias). The conjunction put in the node consists of $Q_b - Q$, i.e., the literals that have been added to Q in order to produce Q_b. Remark that the only way that the refinement operator can apply a substitution to a literal in the current query is to add the substituted literal to the end of the query. In the left branch, Q_b will be further refined, while in the right branch Q is to be refined. When the stop criterion holds (typically, this is when a predefined minimum number of examples is reached), a leaf is built. The PREDICT procedure returns the most frequent class of the examples in E in case of classification, or the mean target value in case of regression.

4.3 Adding complex aggregates to Tilde

TILDE was modified to include (complex) aggregate conditions. The set of candidates considered at each node in the tree was expanded to consist of the original candidates, augmented with aggregate conditions (both simple and complex ones). As explained in the previous chapter, a *simple* aggregate condition is an aggregate that is constructed directly from the language bias, without having selection conditions. By *complex* aggregate conditions, we mean aggregate conditions that have been refined with selection conditions. It is practically impossible to declare the complex aggregate conditions as intensional background knowledge, if the relevant ones are not known in advance. The main difficulty is that the aggregate queries themselves are the result of a search through some hypothesis space, hence we want to learn them.

To illustrate how our method works, we provide an example language bias for the *Account* example in Table 4.2. This is an extension of the language bias discussed in Ex. 2.12. Next to the *rmode* construct, the language bias now contains a second important construct, *aggcondition*, to specify the candidates that can be generated. To include aggregate conditions in the search space, the user needs to specify the basic ingredients in the *aggcondition* construct: the aggregate functions, the aggregate query, aggregate variables, and comparison operators. A number of values to compare the result with can be provided by the user, or can be obtained using discretization (Blockeel and De Raedt 1997). Aggregate conditions with the *between* predicate are not supported, only aggregate conditions with one threshold value are provided.

The system then constructs simple aggregate conditions, using these components. The refinement operator ρ includes the aggregate conditions in the set of candidate queries it generates. A simple aggregate condition that will be generated from the first *aggcondition* construct in Table 4.2 is for instance:

$$max(Balance, account(PersId, AccId, Type, Balance), M), M \geq 2000$$

with *PersId* bound to the *PersId* variable in $person(PersId, Class)$. This query states that the maximum balance of the accounts of a person exceeds 2000.

After adding the standard literals coming from the *rmode* constructs and the simple aggregate conditions coming from the *aggcondition* constructs to the set of candidates, the construction of complex aggregate conditions takes place. This can be done in two ways: refining an aggregate condition that occurs

Table 4.2: Advanced language bias for the *Account* example.

```
% prediction
predict(person(+persid,-class)).

% types
typed_language(yes).
type(account(persid, accid, acctype, balance)).
type(transaction(accid, trans, transtype, amount)).
type(person(persid,class)).

% rmodes: basic language constructs
rmode(account(+PersId, -AccId, -Tp, -Bal)).
rmode(account(+PersId,+-AccId, #["savings","checkings"],-Bal)).
rmode((account(+PersId, +-AccId, -Tp,Bal), Bal ≥ #[500,2000,5000,10000])).
rmode(transaction(+AccId, -TrId, -Tp, -Am)).
rmode(transaction(+AccId, +-TrId,#["deposit","withdrawal"], -Am)).
rmode((transaction(+AccId, +-TrId, -Tp, Am), Am ≥ #[500,1000,2000])).

% aggcondition: aggregate condition constructs
aggcondition([max], account(+PersId, -AccId, -Tp, -Bal), Bal, [≤,≥],[2000,5000]).
aggcondition([min], account(+PersId, -AccId, -Tp, -Bal), Bal, [≤,≥],[-100,0,100]).
aggcondition([sum], account(+PersId, -AccId, -Tp, -Bal), Bal, [≤,≥],[0,5000]).
aggcondition([count],account(+PersId, -AccId, -Tp, -Bal), AccId, [≤,≥],[2,5]).
aggcondition([count],transaction(+AccId, -TrId, -Tp, -Am), Tr, [≤,≥],[5,10,50]).
```

higher in the tree or directly introducing an aggregate condition refined with selections. We discuss the two approaches.

4.3.1 Refining an existing aggregate condition

In this approach aggregate conditions that occur higher in the tree are refined. Only aggregate conditions in the current query are considered, since these are the conditions that succeed for the examples in the node to be expanded. For every such aggregate condition C, a local search has to be conducted.

In the previous chapter, we have introduced two refinement operators to refine aggregate conditions. The first one is based on α-subsumption and basically adds a selection condition or applies a substitution to the aggregate query. We have shown that such refinements may result in generalizations and have proposed a second refinement operator to solve this problem. The second refinement operator works under μ-subsumption.

Let us take a look at the generality issue in the context of TILDE. In TILDE, refining the hypothesis under consideration means adding a node to a leaf of the tree. Given that with each node a query is associated, we can reformulate this as follows: given a query Q, the refinements of Q are obtained by extending Q with one or more literals from the language bias. Thus, the examples for which a refinement (Q, R) (meaning Q *and* R) is true, have to be a subset of the examples for which Q is true, which means (by definition) that (Q, R) is always a specialization of Q. In case Q contains an aggregate condition C of which R is a refinement that is a generalization of C, the set of examples for which (Q, R) succeeds equals the set of examples for which Q succeeds. In that case, the refinement (Q, R) will never be chosen by the OPTIMAL_SPLIT procedure from Table 4.1, since it yields no split. This means that pure generalizations do not occur in this context, and hence, a (simple) refinement operator based on α-subsumption can be used to implement ρ. A drawback of this method is that the generated refinement space may contain a number of refinements that are useless.

Pseudo-code for such a refinement operator based on α-subsumption is outlined in Table 4.3. An extra input argument is assumed for the ρ procedure: the parameter MAX_SELECTIONS denotes the maximum number of selection conditions that may be added to an aggregate condition. For every aggregate condition C occurring in the current query, a local search has to be conducted. Therefore, ρ constructs an inner refinement operator ρ_α, which generates candidates by extending the aggregate query of C with all features specified in either the *rmode* or *aggcondition* constructs. Each candidate generated by ρ_α is included as an aggregate query in an aggregate condition with the same aggregate function, comparison operator, and threshold value as C. These new aggregate conditions are added to the set of candidates produced by ρ.

Example 4.1 *If the current query Q at a given node N is*

$person(P, Class),$
$count(Acc, account(P, Acc, Tp, Bal), C), C \geq 5,$

then, given the language bias of Table 4.2, all refinements of Q that represent complex aggregate conditions are given in Table 4.4. No recursive aggregate conditions (see Ex. 3.1) are shown. For example, the last refinement generated by ρ is

$person(P, Class),$
$count(Acc, account(P, Acc, Type, Bal), C), C \geq 5,$
$count(Acc, (account(P, Acc, Tp, Bal),$
$\quad\quad transaction(Acc, Tr, TrTp, Am), Am \geq 2000), C'), C' \geq 5$

This refinement states that a person is positive if he has at least five accounts that have a transaction with an amount larger than 2000 associated with it. If the query above is chosen by the OPTIMAL_SPLIT *procedure, then*

$count(Acc, (account(P, Acc, Tp, Bal),$
$\quad\quad transaction(Acc, Tr, TrTp, Am), Am \geq 2000), C'), C' \geq 5.$

Table 4.3: Refinement operator based on α-subsumption for refining complex aggregate conditions in the current query.

```
procedure $\rho$ ($Q$: query, MAX_SELECTIONS: integer):
    candidates := EXTEND_WITH_RMODES($Q$)
    candidates := candidates $\cup$ EXTEND_WITH_AGGCONDITIONS($Q$)
    for each aggregate condition $(F(V, Q_1, R), R\vartheta T) \in Q$
        if SELECTIONS($Q_1$) $<$ MAX_SELECTIONS
        then candidates := candidates $\cup$ $\rho_\alpha$ $((F(V, Q_1, R), R\vartheta T))$
    return candidates

procedure $\rho_\alpha$ $((F(V, Q_1, R), R\vartheta T)$: aggregate condition):
    candidates := $\emptyset$
    querycand := EXTEND_WITH_RMODES($Q_1$)
    querycand := querycand $\cup$ EXTEND_WITH_AGGCONDITIONS($Q_1$)
    for all $(Q_1, Q_2) \in$ querycand
        candidates := candidates $\cup$ $(F(V, (Q_1, Q_2), R), R\vartheta T)$
    return candidates
```

is the conjunction added in the left child node of N.

Also the refinement operator based on μ-subsumption can be used to refine aggregate conditions in the current query. In that case the aggregate condition C needs to be looked up in the refinement cubes and, based on the direction of the arrows in the cube, the corresponding refinements are generated.

Let us consider an example to compare the two refinement operators. Suppose the aggregate condition C in the current query that we wish to refine is

$avg(Bal, account(PersId, AccId, Type, Bal), R), R > 1000.$

The α-subsumption based refinement operator would generate all refinements shown in Table 4.4. Since the aggregate condition C is non-monotone in the aggregate query, a lot of the refinements may be generalizations and will thus, in combination with C, cover the same set of examples as C. In other words, a lot of them are useless. If we were using a refinement operator based on μ-subsumption, then we can use the refinement cube drawn in Fig. 3.2(a) to generate the refinements. However, we notice that the *avg* function is only monotonically refinable along the dimensions of the aggregate function or threshold value. This means that refinements along the query axis, although they might all be valid, will not be generated. Therefore, and also because the latter refinement framework was chronologically developed in a later stage, we only provide a refinement operator based on α-subsumption for refining aggregate conditions that occur higher in the tree.

Table 4.4: Complex aggregate condition refinements for the query $Q = count(Acc, account(P, Acc, Tp, Bal), C), C \geq 5$. Only the new aggregate query to form the refinement is shown. No recursive aggregates are used.

(account(P,Acc,Tp,Bal),account(P, Acc2, Tp2, Bal2))
(account(P,Acc,Tp,Bal),account(P, Acc, savings, Bal))
(account(P,Acc,Tp,Bal),account(P, Acc, checkings, Bal))
(account(P,Acc,Tp,Bal),account(P, Acc2, savings, Bal2))
(account(P,Acc,Tp,Bal),account(P, Acc2, checkings, Bal2))
(account(P,Acc,Tp,Bal),account(P, Acc, Tp, Bal),Bal $\geq$ 500)
(account(P,Acc,Tp,Bal),account(P, Acc, Tp, Bal),Bal $\geq$ 2000)
(account(P,Acc,Tp,Bal),account(P, Acc, Tp, Bal),Bal $\geq$ 5000)
(account(P,Acc,Tp,Bal),account(P, Acc, Tp, Bal),Bal $\geq$ 10000)
(account(P,Acc,Tp,Bal),account(P, Acc2, Tp2, Bal2),Bal2 $\geq$ 500)
(account(P,Acc,Tp,Bal),account(P, Acc2, Tp2, Bal2),Bal2 $\geq$ 2000)
(account(P,Acc,Tp,Bal),account(P, Acc2, Tp2, Bal2),Bal2 $\geq$ 5000)
(account(P,Acc,Tp,Bal),account(P, Acc2, Tp2, Bal2),Bal2 $\geq$ 10000)
(account(P,Acc,Tp,Bal),transaction(Acc, Tr, TrTp, Am))
(account(P,Acc,Tp,Bal),transaction(Acc, Tr, deposit, Am))
(account(P,Acc,Tp,Bal),transaction(Acc, Tr, withdrawal, Am))
(account(P,Acc,Tp,Bal),transaction(Acc, Tr, TrTp, Am),Am $\geq$ 500)
(account(P,Acc,Tp,Bal),transaction(Acc, Tr, TrTp, Am),Am $\geq$ 1000)
(account(P,Acc,Tp,Bal),transaction(Acc, Tr, TrTp, Am),Am $\geq$ 2000)

4.3.2 Directly inserting a refined aggregate condition

There is a second way in which complex aggregates can be built. It is based on lookahead (Blockeel and De Raedt 1997), a technique commonly used in ILP to make the learner look ahead in the refinement lattice. In most cases, the refinement operator ρ adds only one basic language construct (i.e., the new node contains literal(s) from only one basic language construct). In some cases, however, it is interesting to add more literals at once, e.g., if the first language construct yields no gain, but introduces interesting variables that can be reused by other literals. If the refinement operator adds up to $k+1$ language constructs, one says that it performs a lookahead of depth k. We extend this mechanism to be directly applied to the aggregate query. In the context of the refinement operator described in Table 4.3, ρ_α refines aggregate queries not from the aggregate conditions occurring in the current query, but from the simple aggregate conditions already added to the set of candidates[1]. These aggregate conditions are refined with up to a predefined depth of literals. The adapted ρ procedure is shown in Table 4.5. The differences with Table 4.3 are

[1]The two approaches may also be used together. However, the refinements generated by the lookahead approach usually contain the refinements of the first approach.

Table 4.5: Refinement operator based on α-subsumption for directly adding complex aggregate conditions to TILDE's search space. The procedure ρ_α is the same as in Table 4.3.

```
procedure $\rho$ ($Q$: query, MAX_SELECTIONS: integer):
    $candidates$ := EXTEND_WITH_RMODES($Q$)
    $candidates$ := $candidates$ $\cup$ EXTEND_WITH_AGGCONDITIONS($Q$)
    for each aggregate condition $(F(V, Q_1, R), R\vartheta T) \in \boxed{\text{candidates}}$
        $\boxed{\textbf{while}}$ SELECTIONS($Q_1$) $<$ MAX_SELECTIONS
            $candidates$ := $candidates$ $\cup$ $\rho_\alpha$ $((F(V, Q_1, R), R\vartheta T))$
    return $candidates$
```

highlighted.

With this approach, the query

$$count(Acc, (account(P, Acc, Tp, Bal), \\ transaction(Acc, Tr, TrTp, Am), Am \geq 2000), C'), C' \geq 5$$

of Ex. 4.1 could immediately be inserted, without having the query

$$count(Acc,account(P, Acc, Tp, Bal), C), C \geq 5$$

in one of its ancestor nodes. Obviously, the lookahead technique is computationally expensive, but it may yield significant predictive improvements (Blockeel and De Raedt 1997).

The refinements of aggregate conditions based on lookahead can also be generated by a refinement operator based on μ-subsumption. This brings a significant efficiency improvement, while generating exactly the same refinement space in this case. It is discussed in the next section. Since μ-subsumption was developed in a later stage of this work, we first describe an other approach to improve efficiency. It is based on sampling the refinement space.

4.4 Improving efficiency

In Chapt. 3 we have identified the substantial increase of the search space as one of the major problems when learning theories with complex aggregate conditions. The time needed to traverse this space can in some cases escalate, such that the use of complex aggregates, especially when the lookahead method is used, becomes practically infeasible. In this section we present two techniques to cope with this problem.

In Sect. 4.4.1 we discuss an approach based on random forests. Random forests are collections of trees, where each tree has been built by considering in each node only a random sample of the possible tests for that node. In our ILP setting, this boils down to making a random selection of the refinements generated by α-subsumption. This compensates for the increase of the branching factor of the search when considering complex aggregate conditions. In Sect. 4.4.2 we explore the whole feature space, but traverse it in an efficient way. This approach is based on using a refinement operator based on μ-subsumption, as defined in the previous chapter. This allows to prune the search space. The second approach has the advantage of producing only a single tree, and hence, to keep the model interpretable.

4.4.1 Random forests

Random forest induction (Breiman 2001) is an ensemble method. An ensemble learning algorithm constructs a set of classifiers, and then classifies new data points by combining the predictions of each classifier. A necessary and sufficient condition for an ensemble of classifiers to be more accurate than each of its individual members, is that the classifiers are accurate and diverse (Hansen and Salamon 1990). An accurate classifier does better than random guessing on new examples. Two classifiers are diverse if they make different errors on new data points.

Random forests introduce diversity among the classifiers by changing the feature sets over the different tree induction processes, and additionally by resampling the data. The exact procedure to build a forest with k trees is as follows:

- **for** $i = 1$ **to** k **do**:
  - build training set D_i by sampling (with replacement) from data set D
  - learn a decision tree T_i from D_i *using randomly restricted feature sets*

The part of the algorithm where random forests differ from the normal bagging procedure is emphasized. Normally, when inducing a decision tree, the best feature is selected from a fixed set of features F in each node. In bagging, this set of features does not vary over the different runs of the induction procedure. In random forests however, a different random subset of size $f(|F|)$ is considered at each node (e.g., $f(x) = 0.1x$ or $f(x) = \sqrt{x}, \ldots$), and the best feature from this subset is chosen. This obviously increases variability.

Consider now a classification problem where a new example is to be assigned one of the m possible classes ($\omega_1, \ldots, \omega_m$). Each decision tree T_i from the random forest gives a class label Cl_i to the new example. The label given by the random forest to the new example will then be

$$Cl^* = \arg\max_{\omega_j} \sum_{i=1}^{k} I(Cl_i = \omega_j)$$

where $I(x) = 1$ if x is true and $I(x) = 0$ otherwise. Hence the majority vote of the predicted class labels of the set of k trees in the random forest is the label predicted.

An advantage of using bagging is that out-of-bag error estimates (Breiman 1996) can be used to estimate the generalization errors. This removes the need for a set-aside test set or cross-validation. Out-of-bag error estimation proceeds as follows: each tree is learned on a training set D_i drawn with replacement from the original training set D. For each example d in the original training set, the predictions are aggregated only over those classifiers T_i for which D_i does not contain d. This is the out-of-bag classifier. The out-of-bag error estimate is then the error rate of the out-of-bag classifier on the training set. Note that in each resampled training set, about one third of the instances are left out (actually $1/e$ in the limit). As a result, out-of-bag estimates are based on combining only about one third of the total number of classifiers in the ensemble. This means that they might overestimate the error rate, certainly when a small number of trees is used in the ensemble.

Random forests have some other interesting properties (Breiman 2001). They are efficient since only a sample of $f(|F|)$ features needs to be tested in each node, instead of all features. They do not overfit as more trees are added. Furthermore, they are relatively robust to outliers and noise, and they are easily parallelized.

4.4.1.1 First order random forests with complex aggregates

We have upgraded Tilde with complex aggregates to a first order random forest induction algorithm, called Forf. First, a wrapper was built around the algorithm in order to get bagging. Some adaptations were made to get out-of-bag error estimates.

Next, we built in a filter that allows only a random subset of the tests to be considered at each node[2]. As a result, constructing a new node proceeds as follows: first all possible refinement candidates $\rho(Q)$ from the current query Q are generated, then a random subset of approximate size $f(|\rho(Q)|)$ (where $f(x)$ is a function given by the user, e.g., $f(x) = 0.1x$ or $f(x) = \sqrt{x}$, ...) is chosen. For each query in this subset, a heuristic is computed and the optimal split is placed in the new node. Consequently, only a part of all generated queries needs to be executed on the examples to calculate the heuristics, which obviously results in an efficiency gain.

To summarise, we provide an overview of the resulting algorithm in Table 4.6. The procedure grow_forest takes the number of trees to grow as one of its input parameters. For each tree, it first builds a new set of examples, sampled with replacement from the original set E. Then the procedure grow_tree_2 is

[2]This actually differs from the definition in (Breiman 2001) where a random subset of the attributes, instead of the tests, is chosen. Note that one attribute may yield different tests.

Table 4.6: Algorithm for first order random forest induction. The parts of the algorithm that are in boxes show the differences with the GROW_TREE algorithm from Table 4.1.

```
procedure GROW_FOREST ($N$: nb trees, $f$: function, $E$: examples):
    for $i = 1$ to $N$
        $E_i$ := SAMPLE($E$)
        $T_i$ := GROW_TREE_2($E_i$, $true$, $f$)
    return forest($T_1, T_2, ..., T_N$)

procedure GROW_TREE_2 ($E$: examples, $Q$: query, $\boxed{f\text{: function}}$):
    $candidates := \rho(Q)$
    $\boxed{subsetsize := f(|candidates|)}$
    $\boxed{candidates\_subset := \text{SUBSET}(candidates, subsetsize)}$
    $Q_b$ := OPTIMAL_SPLIT($\boxed{\text{candidates\_subset}}$, $E$)
    if STOP_CRIT ($Q_b, E$)
    then
        $K$ := PREDICT($E$)
        return leaf($K$)
    else
        $conj := Q_b - Q$
        $E_1 := \{e \in E | Q_b$ succeeds in $e \wedge B\}$
        $E_2 := \{e \in E | Q_b$ fails in $e \wedge B\}$
        $left$ := GROW_TREE_2 ($E_1, Q_b, \boxed{f}$)
        $right$ := GROW_TREE_2 ($E_2, Q, \boxed{f}$)
        return node($conj, left, right$)
```

called, which is an adaptation of GROW_TREE (see Table 4.1), differences with this algorithm are denoted in boxes. The refinement operator ρ includes (complex) aggregate conditions in the set of candidate splits it generates, as discussed in Section 4.3. The SUBSET procedure generates a random subset of the candidate set. The size of the subset is a function f of the number of candidates. Hence, each candidate has probability $\frac{f(|\rho(Q)|)}{|\rho(Q)|}$ to be selected. The OPTIMAL_SPLIT procedure returns the optimal split among the set of candidate splits.

A more efficient approach. An important difference between propositional random forests and first order random forests is the generation of tests at each node. In a propositional tree the possible tests are the same at each node. In first order trees however, a node may introduce variables that can be reused in the nodes of its left subtree. Hence, the number of candidate tests depends on the number of variable bindings in the conjunction of all succeeding tests on the path from the root to the node that is to be extended (this conjunction was called the current query earlier in this chapter).

Query sampling reduces the time used for query evaluation in random forests. Still, in FORF, query generation also takes a substantial amount of time, certainly when lookahead within the aggregate queries is performed and a huge amount of queries needs to be generated. In that case, a lot of time is spent on generating queries that may not be evaluated in the end. As such, the algorithm described in Table 4.6 is still performing a lot of redundant actions. A more efficient version of the GROW_TREE_2 procedure would directly generate a random sample of queries, instead of generating them all. This is not trivial since the number of queries in the sample is a function of the total number of possible queries, which is hard to calculate in advance. Moreover, if we iteratively take a random language construct (*rmode* or *aggcondition*) from the language bias to produce a random candidate query, the resulting sample will not be drawn from a uniform distribution over all possible candidates, since the number of candidates differs per language construct and also depends on the current query. Therefore, such an efficient sample generator would consist of two steps. First, for each language construct in the language bias, the number of candidates that can be generated from it would have to be determined (without generating them all). Second, given the number of candidates, a uniform distribution can be used to randomly generate the query sample.

In TILDE, the refinement operator ρ extends the current query with literals from the basic language constructs. This is implemented as follows: for each language construct in the language bias, variable instantiation is performed and for each of these variable instantiations all possible constants are generated (in case of lookahead, these are again refined in the same way). Let us illustrate this with an example from the *Account* dataset.

Example 4.2 *A possible language bias for the Account dataset is given in Table 4.2. Suppose we want to refine a node where the current query Q is*

$$person(P, C), account(P, A_1, T_1, B_1), account(P, A_2, checkings, B_2).$$

Then TILDE *will generate all candidate refinements for this query according to the tree in Figure 4.1. In the figure, only the rmode language constructs are drawn for simplicity and readability (the aggregate conditions are not treated differently by the sampling procedure). At depth 1 of the tree all language constructs that occur in the language bias (both the ones specified by the rmode constructs as those specified by the aggcondition constructs) are added, at depth 2 variable instantiations with respect to query Q are performed and at depth 3 possible constants are filled in. If lookahead is used (either normal lookahead or in complex aggregate conditions), some leaves of this tree are again expanded as if they were the root of the tree.*

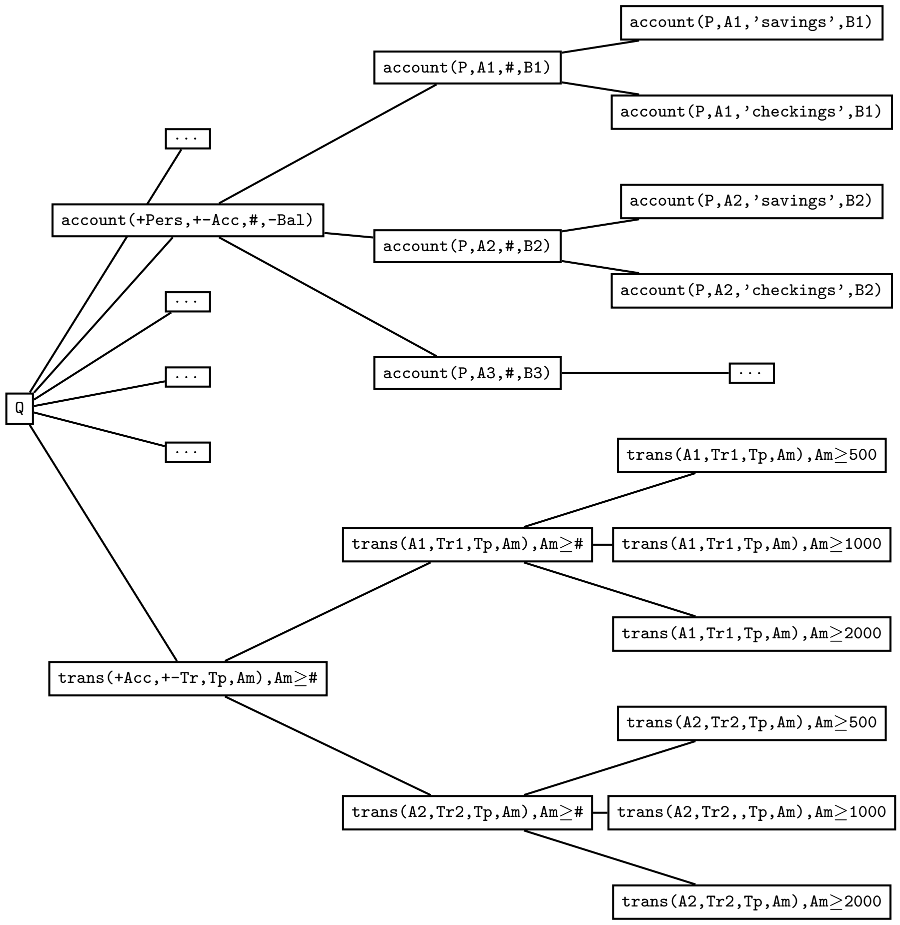

account(P,A1,'savings',B1)
account(P,A1,#,B1)
account(P,A1,'checkings',B1)
...
account(+Pers,+-Acc,#,-Bal)
account(P,A2,'savings',B2)
account(P,A2,#,B2)
account(P,A2,'checkings',B2)
...
...
account(P,A3,#,B3)
...
Q
...
trans(A1,Tr1,Tp,Am),Am≥500
trans(A1,Tr1,Tp,Am),Am≥#
trans(A1,Tr1,Tp,Am),Am≥1000
trans(A1,Tr1,Tp,Am),Am≥2000
trans(+Acc,+-Tr,Tp,Am),Am≥#
trans(A2,Tr2,Tp,Am),Am≥500
trans(A2,Tr2,Tp,Am),Am≥#
trans(A2,Tr2,,Tp,Am),Am≥1000
trans(A2,Tr2,Tp,Am),Am≥2000

Figure 4.1: Generation of candidates to refine the current query Q: $person(P,C)$, $account(P,A_1,T_1,B_1), account(P,A_2,checkings,B_2)$ in TILDE.

In order to count, in an efficient way, the number of candidate tests an rmode (or aggregate condition) produces, we proceeded as follows. The tree from Figure 4.1 is only built partially, in the sense that each instantiated literal of depth two only has a single child node at depth three, representing all constants for that language construct. Hence, in case of lookahead (e.g., with complex aggregates), the corresponding candidates are generated only once, instead of once per available constant. We then count, for each language construct, how much offspring it yields, thereby multiplying each node representing the constants with the number of constants available for that literal (this number can be obtained from the language bias). For example, for the last rmode in Figure 4.1 ($transaction(+AccId, +-TrId, Type, Amount), Amount > \#[500, 1000, 2000]$) there are three constants. While generating the search tree, we find two instantiations for this rmode. Thus, a total of 6 refinements is obtained from this rmode. Doing this for all language constructs we get a probability distribution over them. Using this distribution a sample of queries is randomly generated.

This approach will be especially rewarding if the tree of Figure 4.1 contains many levels, e.g., because of the use of lookahead or complex aggregates, and if its branching factor can be largely reduced (i.e. when a lot of constants need to be filled in). The largest gain over the naive algorithm will occur when using small sample ratios. Obviously, when one would use a ratio of, say 90%, the extra work for counting the number of candidates will not pay off.

4.4.2 Structuring the aggregate space

In the previous section we have proposed a solution for the search space explosion based on sampling the search space. The approach was presented in the context of first order random forests. While the sampling procedure obviously reduces the induction time to build a single tree, the time needed to induce a whole forest of trees can still be a limiting factor.

In this section we present an other approach to deal with the growth of the search space. This approach is not based on ensemble methods, thus we keep the interpretability often praised in decision trees. The approach also does not sample the search space, but performs an exhaustive search at each node of the tree. It is based on μ-subsumption, which was defined in Chapt. 3. In particular, μ-subsumption will be used to structure the aggregate refinement space in a general-to-specific way, allowing to prune parts of that space when looking for the optimal refinement candidate.

We first discuss how TILDEtraverses the search space (Sect. 4.4.2.1). Next, we describe how the new refinement operator was implemented (Sect. 4.4.2.2).

4.4.2.1 Traversing the search space

We first provide more details on the OPTIMAL_SPLIT procedure in the algorithm shown in Table 4.1. This procedure has to perform the following two tasks:

- evaluate all candidates against the examples in the node being split, calculating the heuristic value for each of the candidates, and
- choose the candidate with the highest heuristic value to split the node.

In the first step, all candidate queries need to be executed against all examples. This involves a double loop: one over the candidates and one over the examples. Two strategies can be used: "queries in the outer loop" and "examples in the outer loop" (Blockeel et al. 2002). Both have been used in data mining systems. The latter strategy does not require that all data resides in main memory at the same time, which can be advantageous when processing large datasets.

The set of candidates returned by the refinement operator ρ in Table 4.1 is collected in a data structure called a *query pack* (Blockeel et al. 2002). A query pack is a tree structure that represents the search space at a certain node of the decision tree that is being built. The query pack contains queries in the nodes. The underlying idea is that refinements that are very similar can be executed more efficiently if their common part is executed only once for each example. The pack is organized such that the query contained in a node is a common prefix for all its children. Since the refinement operator takes the current query and adds one (or more if lookahead is used) literal to it, the current query is a common prefix for all refinements. Therefore, in TILDEquery packs usually have a broom-like structure. The corresponding query pack for the refinements in Fig. 4.1 is shown in Fig. 4.2. The query packs allow pruning the search space while executing the queries: if, for a particular example, there exists no variable substitution for the test in a node of the pack, none of the tests in its children has to be tested for that example (hence, the use of query packs assumes the "examples in outerloop" strategy). It is clear that the resulting efficiency gain is highest for small query packs of large depth.

In Sect. 4.3 we described how to add aggregate conditions to the hypothesis language of TILDE. When using a refinement operator based on α-subsumption, no structure is imposed on the aggregate search space, i.e., the aggregate conditions are all added to the query pack on the lowest level, together with the other tests, and are executed in the order given. However, using the monotonicity properties discussed in Sect. 3.4.3, we know that whenever an aggregate condition C fails for an example, none of the tests that can be obtained from C via monotone paths of the refinement cubes has to be executed against that example. This can be exploited when building the query pack. Organizing the pack such that it reflects the monotone paths of the refinement cubes would obviously result in an efficiency gain, while the same search space is searched, and hence, the same tree is obtained.

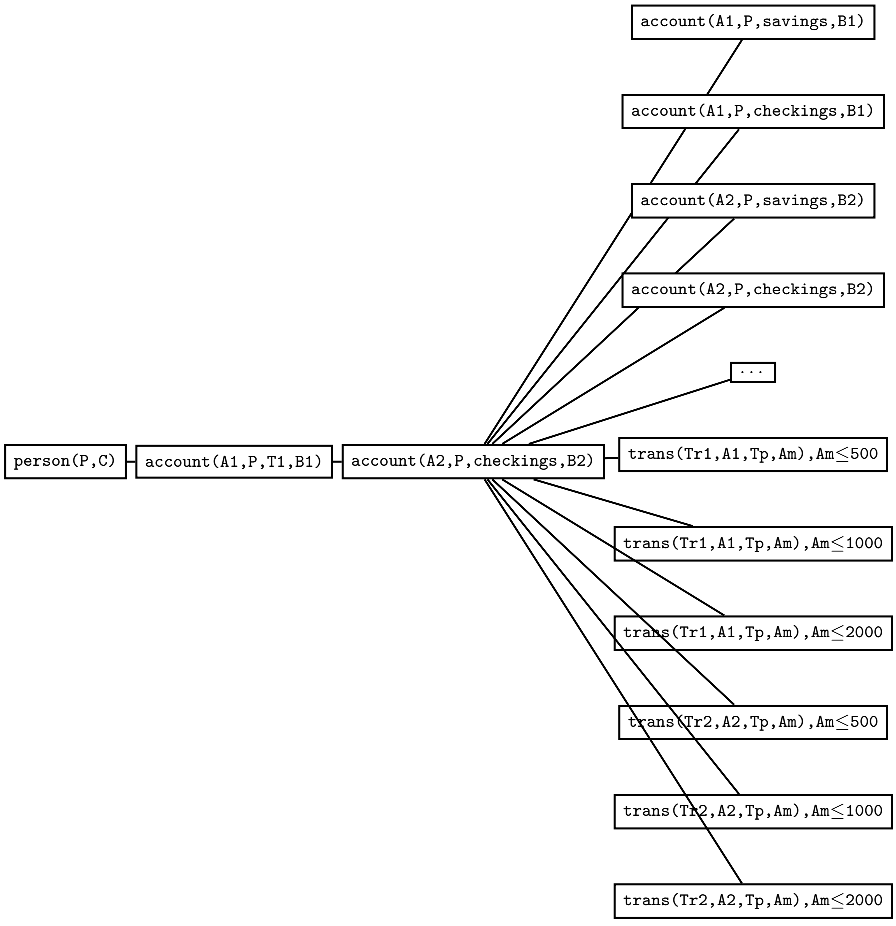

account(A1,P,savings,B1)
account(A1,P,checkings,B1)
account(A2,P,savings,B2)
account(A2,P,checkings,B2)
...
person(P,C)
account(A1,P,T1,B1)
account(A2,P,checkings,B2)
trans(Tr1,A1,Tp,Am),Am≤500
trans(Tr1,A1,Tp,Am),Am≤1000
trans(Tr1,A1,Tp,Am),Am≤2000
trans(Tr2,A2,Tp,Am),Am≤500
trans(Tr2,A2,Tp,Am),Am≤1000
trans(Tr2,A2,Tp,Am),Am≤2000

Figure 4.2: Query pack with refinements of the current query $person(P, C)$, $account(A_1, P, T_1, B_1), account(A_2, P, checkings, B_2)$. The lowest level contains all literals that can be added to the current query to form a refinement.

4.4.2.2 Re-implementing Tilde's query pack structure

We provide a new refinement operator ρ for Tilde that organizes the aggregate conditions in the query pack that it returns in a way that reflects the monotone paths in the refinement cubes. Pseudo-code for this refinement operator is given in Table 4.7. The current query and the candidates are now contained in a query pack. In the algorithm, *pack(X)* means the node in the query pack with conjunction X. Nodes can be added to the query pack by the *add* method. First, candidates originating from the rmodes and from the cubes' start conditions are generated and added to the pack. Then, for each start condition, we subsequently refine the aggregate query, the aggregate function, and the threshold to compare the result with. Refining along these dimensions consists of following the arrow in the corresponding cube along the direction pointed to. The fact that in the REFINE_FUNCTION and REFINE_VALUE methods, we do not consider the three dimensions anymore for further refinements, ensures that each aggregate condition is only generated once. Note that in Tilde a query can only be refined by specializing it, not by dropping literals. Thus, in the context of the refinement cubes, refinements along the Q-axis can only take place in one direction.

Example 4.3 *Figure 4.3 shows how the query pack for the example in Fig. 3.5 would be organized, both for the α-subsumption and μ-subsumption based refinement operators. If the test $sum(Bal, account(PersID, AccID, Type, Bal), S)$, $S \geq 10000$ fails for an example, with the optimized pack structure we could prune all its children in the search space, whereas in the original unstructured space, they would still all be tested.*

4.5 Dealing with empty sets

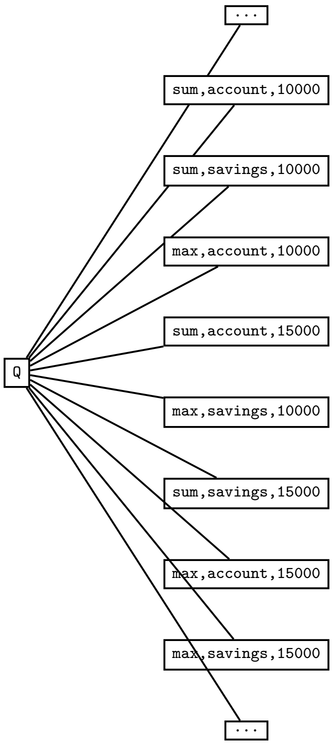
An issue that has not been mentioned before is what happens if an aggregate function is undefined for an example. For example, the maximum balance of a person's savings accounts is undefined if a person only has checkings accounts, or if he does not have any account. Aggregating over empty sets often occurs when using complex aggregates: the selection condition on the set to aggregate over can become so complex that the aggregate is defined only for a few examples. Only the function *count* is able to deal with this in a natural way. For the other functions, a number of possibilities exist. We first discuss some of them. Then we give a number of desirable properties for the approaches and report for each approach whether the property is fulfilled. Afterwards, we explain the approach adopted in Tilde.

Possible approaches to deal with undefined aggregate conditions include

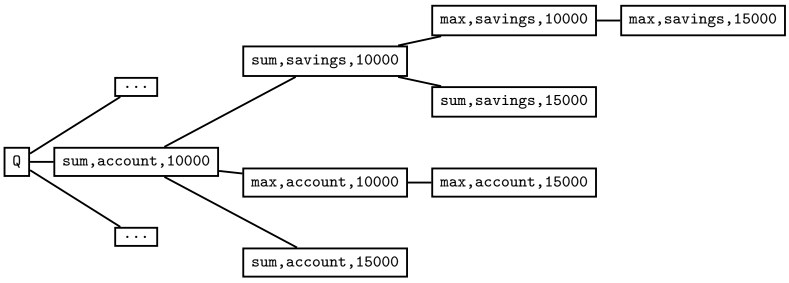
1. using a fixed value (e.g., 0) as result,

Table 4.7: Refinement operator based on μ-subsumption for adding complex aggregate conditions to TILDE's search space.

```
procedure $\rho$ ($pack(Q)$: query pack, MAX_SELECTIONS: integer):
    $rmode\_cand$ := EXTEND_WITH_RMODES($Q$)
    $aggcondition\_cand$ := EXTEND_WITH_STARTCONDITIONS($Q$)
    $conj$ := LAST_LITERAL($Q$)
    for each $(Q, Q_2) \in rmode\_cand$
        $pack(conj)$.add($Q_2$)
    for each $(Q, (F(V, Q_1, R), R\vartheta T)) \in aggcondition\_cand$
        $pack(conj)$.add($(F(V, Q_1, R), R\vartheta T)$)
        $\rho_\mu$($pack(F(V, Q_1, R), R\vartheta T)$,MAX_SELECTIONS)

procedure $\rho_\mu$ ($pack(F(V, Q, R), R\vartheta T)$: query pack,
                      MAX_SELECTIONS: integer):
    REFINE_QUERY($pack(F(V, Q, R), R\vartheta T)$,MAX_SELECTIONS)
    REFINE_FUNCTION($pack(F(V, Q, R), R\vartheta T)$)
    REFINE_VALUE($pack(F(V, Q, R), R\vartheta T)$)

procedure REFINE_QUERY ($pack(F(V, Q, R), R\vartheta T)$: query pack,
                      MAX_SELECTIONS: integer):
    if SELECTIONS($Q$) <MAX_SELECTIONS
    then $querycand$ := EXTEND_WITH_LANGUAGE_CONSTRUCTS($Q$)
    for all $(Q, Q_2) \in querycand$
        $pack(F(V, Q, R), R\vartheta T)$.add($F(V, (Q, Q_2), R), R\vartheta T$)
        REFINE_QUERY($pack(F(V, (Q, Q_2), R), R\vartheta T)$,MAX_SELECTIONS-1)
        REFINE_FUNCTION($pack(F(V, (Q, Q_2), R), R\vartheta T)$)
        REFINE_VALUE($pack(F(V, (Q, Q_2), R), R\vartheta T)$)

procedure REFINE_FUNCTION ($pack(F(V, Q, R), R\vartheta T)$: query pack):
    $F_2$ := GET_REFINED_FUNCTION($F$)
    $pack(F(V, Q, R), R\vartheta T)$.add($F_2(V, (Q), R), R\vartheta T$)
    REFINE_FUNCTION($pack(F_2(V, (Q), R), R\vartheta T)$)
    REFINE_VALUE($pack(F_2(V, (Q), R), R\vartheta T)$)

procedure REFINE_VALUE ($pack(F(V, Q, R), R\vartheta T)$: query pack):
    $T_2$ := GET_REFINED_VALUE($T$)
    $pack(F(V, Q, R), R\vartheta T)$.add($F(V, (Q), R), R\vartheta T_2$)
    REFINE_VALUE($pack(F(V, (Q), R), R\vartheta T_2)$)
```



(a)



(b)

Figure 4.3: Comparison between the query pack structures. In this figure, the current query Q is reduced to one node. (a) Original query pack structure (α-subsumption). (b) Query pack structure that reflects the monotone paths in the refinement cubes (μ-subsumption).

2. using a value in the range of the aggregate condition as result

    a) the value can be chosen as close as possible to the values of defined examples (e.g., the average of the values for the examples on which the aggregate condition is defined),

    b) the value can be chosen as far as possible of the values of defined examples as result (e.g., since the maximum decreases when we decrease the set, we can set the maximum of the empty set as $max(\emptyset) = min(S)$, and similarly, $min(\emptyset) = max(S)$ with S the set of values observed for the aggregate query for all training examples),

3. always failing the aggregate condition for examples for which it is undefined,

4. disqualifying an aggregate condition from being chosen by the OPTIMAL_SPLIT procedure if it is undefined for at least one example.

Desirable properties for the approaches to deal with empty sets include the following

- equivalent branches:
  When the comparison operator of an aggregate condition is switched (i.e., $\leq$ is changed into $\geq$ or vice versa), then it is desirable that the branches are switched too (i.e. all examples from the left branch go to the right branch, and vice versa). This allows to dispose of one of the comparison operators and still have the same expressivity.

- generality preservation:
  If an aggregate condition C_1 is more general than C_2, then this generality order should be preserved if C_1 and C_2 are undefined for some examples.

- domain preservation:
  If the aggregate condition returns a value for undefined cases (as in the first two approaches given above), then the comparison to a threshold value makes only sense if the value returned belongs to the range of the aggregate function.

- noise tolerance:
  We say that an approach is noise tolerant if a noisy example can only influence the branch the example itself sorts down to and does not influence the other examples.

- unambiguity:
  After the tree is built, it may be used to make predictions for new unseen examples. If an aggregate condition is undefined for a new example, the child node into which the example has to be sorted down should be unambiguously determined.

Having proposed a number of approaches to deal with undefined aggregate conditions, and a number of desirable properties for the approaches, Table 4.8 shows for each combination of approach and property, whether the property is fulfilled by the approach. The property of equivalent branches is fulfilled by all approaches except the failing approach, since in the latter, examples for which the condition is undefined always go to the right branch, no matter what comparison operator is used. Concerning the second property, only the last two approaches (failing and disqualifying) maintain the generality order. For example the aggregate condition

$$min(Bal, account(PersId, AccId, Type, Bal), M), M \leq 100$$

has a specialization

$$min(Bal, account(PersId, AccId, savings, Bal), M), M \leq 100.$$

Suppose a person has only checkings accounts, and the minimum balance is 500. Whereas the first aggregate condition fails for this person, the second one may succeed if the value returned by the aggregate condition is smaller than 100, which would result in a generalization. The domain preserving property is trivially fulfilled by the second approach and is not applicable to the third and fourth one. Noise tolerance is fulfilled by every approach but the last one. In the last approach one noisy example is able to discard the aggregate condition from being chosen by the OPTIMAL_SPLIT procedure. Also disambiguity is fulfilled by every approach but the last one.

Table 4.8: Approaches to deal with aggregate conditions that are undefined for some examples and desirable properties for them.

	fixed value	value in range	fail	disqualify
eq. branches	yes	yes	no	yes
generality pres.	no	no	yes	yes
domain pres.	no	yes	n/a	n/a
noise toler.	yes	yes	yes	no
disambiguity	yes	yes	yes	no

For inclusion in TILDE, probably the most important property is generality preservation. If we would use an approach that does not fulfill this property, then we would not be able to use a refinement operator based on μ-subsumption. Therefore, only the third and fourth approach are candidates that may be considered. Since the fourth approach is too much influenced by noisy examples and is not deterministic w.r.t. new examples, the approach taken in our system is the third one: an aggregate condition fails for examples on which it is undefined. This implies that the trees we learn are not equivalent up to switching branches, thus both the $\leq$ and $\geq$ comparison operators have to be used to obtain full expressivity (if this is desired).

4.6 Experiments

In this section we experimentally evaluate the use of complex aggregates in Tilde. The precise questions we want to answer in our experimental evaluation of complex aggregates are the following:

1. How does the use of aggregates (both simple and complex ones) influence the predictive performance, model size, induction times, and search space sizes of trees output by Tilde?
2. How do our two techniques to deal with the size of the feature space (sampling the aggregate search space by upgrading Tildeto a first order random forest induction system and structuring the aggregate search space with the μ-subsumption based refinement operator) compare w.r.t. induction times?
3. Is the predictive performance of Tildeincreased by upgrading it to a first order random forest induction system?
4. How does the performance of Tildewith complex aggregate conditions relate to other available relational learners?

To answer these questions, we will perform experiments on three datasets, they are introduced in Sect. 4.6.1. The setup used for our experiments is discussed in Sect. 4.6.2. Results are presented in Sect. 4.6.3.

4.6.1 Datasets

For our experiments we used three well-known real world data sets: *Mutagenesis* (Srinivasan et al. 1996), *Diterpenes* (Džeroski et al. 1998), and *Financial* (Berka 2000). The first two data sets contain complicated structures and have been widely used as ILP benchmarks. The latter is a business domain data set with high degree of non-determinacy.

4.6.1.1 Mutagenesis

The *Mutagenesis* dataset (Srinivasan et al. 1996) was already introduced in Ex. 2.3. The task is to predict mutagenicity of 230 nitro-aromatic compounds. We consider the classification version of this problem: a compound is positive if the logarithm of its mutagenicity level is positive and negative otherwise. 60% of the compounds is classified positive. The description of compounds consists of the atoms and the bonds that make up the molecule, i.e., the so called background B2 (Srinivasan et al. 1995) (including the partial charges of the atoms). Of the 230 compounds, 188 are called “regression-friendly”. Unless stated otherwise, we use the complete dataset in the experiments.

The aggregate functions included are

- *count* to count the number of atoms and bonds in the molecules,
- *max*, *min*, and *avg* of the partial charges of the atoms of the molecules,
- *mode* (most frequent value) of bond types and atom elements.

4.6.1.2 Diterpenes

The *Diterpenes* data set (Džeroski et al. 1998) contains information on 1503 diterpenes with known structure. The $red(Mol, Mult, Freq)$ relation stores the measured NMR-Spectra. It contains the multiplicity and frequency for each of the 20 carbon atoms in the skeleton of the diterpene. The $prop(Mol, Satoms, Datoms, Tatoms, Qatoms)$ relation counts the atoms that have multiplicity s, d, t, or q respectively. Additional unary predicates describe to which of the 23 classes a compound belongs. Several learning settings are defined on this data set: using *prop* only, using *red* only, and using both *prop* and *red*. In our experiments, only the *red*-relation was used, since we expect that by allowing aggregate functions the system should be able to construct the propositionalized *prop*-relation by itself if necessary.

The aggregate functions used are

- *min*, *max* and *avg* for the frequencies,
- *mode* for the multiplicities,
- *count* for the different values of multiplicities.

4.6.1.3 Financial

The *Financial* data set originates from the discovery challenge that was organised at PKDD'99 and PKDD'00 (Berka 2000). This data set involves learning to classify expired bank loans into good and bad ones. Since 86% of the examples is positive, the data distribution is quite skewed. The data set consists of 8 relations. For each of the 234 loans, customer information and account information is provided. The account information includes permanent orders and several hundreds of transactions per account. This problem is thus a typical business data set which is highly non-determinate.

The aggregate functions used apply to the orders and transactions and include

- *max*, *min*, *avg*, and *sum* for the amount of transactions and orders,
- *max*, *min*, and *avg* for the balance of transactions,
- *count* to count the number of transactions and orders,
- *mode* for transaction type, transaction operation, and characteristic of transactions and orders.

4.6.2 Experimental setup

To address the first question formulated above, we have investigated the performance of TILDEaccording to different levels of aggregation. In the first level, we did not use any aggregates (afterwards, this setting is denoted by NA). In the second level, simple aggregate conditions were introduced (SA). The third level includes refinement of aggregate queries already present in the tree (RA) and the fourth level allows lookahead up to depth 1 within the aggregate queries (LA). For this experiment we used a refinement operator based on α-subsumption (denoted by TILDE-α). We report accuracy and complexity of the trees, and induction times and maximal encountered search space size to build a tree.

For the second question, the LA-experiments were rerun using the structured search space strategy (denoted by TILDE-μ) and using the first order random forest induction algorithm (denoted FORF) with the original refinement operator. For FORF various settings were used w.r.t. the number of trees in the forest and the number of tests considered at each node. Since TILDE-μ searches the same search space as TILDE, the same tree is obtained in the end. Only induction times are influenced by this method. Combining the prediction of several trees, however, can improve predictive performance, thus for FORF we also evaluate the accuracy obtained, which addresses question 3.

For the last question, we compared the accuracies of TILDE and FORF to other systems for which results were found in the literature.

In all our experiments, the trees output by TILDE were pruned using C4.5's post-pruning method. In FORF no pruning was used, since pruning decreases the diversity among the trees in our random forest. The minimal number of examples a leaf has to cover is two (TILDE's default value). However, for the *Diterpenes* dataset, it was changed to 20, for efficiency reasons, except for answering the last question, where it was reset to two.

Concerning refinements, we do not allow aggregate queries to be refined with new aggregate conditions, since otherwise the search space becomes too large for some of our experimental settings. For *Mutagenesis* and *Diterpenes* the aggregates can be refined with selection conditions coming from any of the rmodes in the language bias. For *Financial* we have constructed a separate set of rmodes, such that refinements would relate to the transaction or order in the simple aggregate condition (and not to other transactions or orders or to the loan or account objects, which would result in a huge search space containing a lot of uninteresting aggregate conditions). For all datasets only one level of complexity was allowed in the RA and LA settings, i.e., starting from the simple aggregates, at most one selection condition was added.

4.6.3 Experimental results

For each of the four questions above, we now present the experimental results.

Table 4.9: Accuracy, model size, induction time, and maximal search space size of Tilde with different levels of aggregation on the *Mutagenesis* data set.

	Tilde			
	LA	RA	SA	NA
accuracy	73.4% (1.9)	73.6% (2.5)	73.1% (2.6)	70.9% (3.7)
nodes	11.7 (1.5)	10.5 (2.5)	9.9 (2.2)	13.7 (1.4)
time	6672.2 (5.8)	18.7 (0.0)	8.5 (0.0)	1.8 (0.0)
search space	38341	1299	570	911

Table 4.10: Accuracy, model size, induction time, and maximal search space size of Tilde with different levels of aggregation on the *Diterpenes* data set.

	Tilde			
	LA	RA	SA	NA
accuracy	84.4% (0.4)	82.7% (0.8)	82.7% (0.8)	70.0% (1.3)
nodes	9.8 (0.7)	9.7 (0.7)	9.4 (1.1)	14.1 (1.9)
time	26701.2 (34.8)	298.6 (0.8)	18.0 (0.1)	40.6 (0.2)
search space	76361	3788	1613	4545

4.6.3.1 Varying the level of aggregation

In this experiment, we run TildEwith different levels of aggregation (*NA*, *SA*, *RA*, and *LA*, as described above) and compare predictive performance, model size, induction times, and search space sizes. Predictive performance is obtained by averaging the accuracy of five full threefold cross-validations with different folds. We use threefold cross-validation since this error assessment approach is closest to out-of-bag estimation, which will be used to measure predictive performance of Forf in Sect. 4.6.3.3. Model size is measured as the average number of internal nodes of the trees learned with the cross validation procedure. Induction time is measured as the time (in seconds) of building a single tree on the whole dataset (averaged over 5 runs). Also the maximal size of the refinement space is reported. The experiments were run with the refinement operator based on α-subsumption.

The results are presented in Table 4.9 (*Mutagenesis*), Table 4.10 (*Diterpenes*), and Table 4.11 (*Financial*), respectively. We now discuss the main results for each performance measure.

Accuracy. The results show that including aggregates in the search space leads to an obvious improvement in predictive performance (the accuracy increases with 2.2% for *Mutagenesis*, 12.7% for *Diterpenes*, and 3.6% for *Financial*). However, the difference between simple (SA) and complex (RA and LA) aggre-

Table 4.11: Accuracy, model size, induction time, and maximal search space size of Tilde with different levels of aggregation on the *Financial* data set.

	Tilde			
	LA	RA	SA	NA
accuracy	98.4% (1.4)	98.2% (0.9)	97.7% (1.1)	94.1% (2.0)
nodes	3.8 (0.4)	4.3 (0.8)	4.3 (0.9)	6.1 (0.6)
time	535.3 (6.7)	23.3 (0.1)	23.4 (0.1)	4.2 (0.0)
search space	6972	305	305	145

`loan`$(LId, AccId, Date, Amount, Dur, Paym, Status), Amount > 156672$?
`+yes:` `min`$(Bal, trans(TrId, AccId, \ldots, Bal, Ch), R), R < 2.0$?
| `+yes:` [neg] (14 ex.)
| `+no:` [pos] (31 ex.)
`+no:` `loan`$(LId, AccId, Date, Amount, Dur, Paym, Status), Amount > 23184$?
        `+yes:` `min`$(Bal, trans(TrId, AccId, \ldots, Bal, Ch), R)), R < 2.0$?
        | `+yes:` [neg] (17 ex.)
        | `+no:` [pos] (151 ex.)
        `+no:` [pos] (21 ex.)

Figure 4.4: Tree for *Financial* with 100% accuracy.

gates is less clear.

Fig 4.4 shows the tree with simple aggregates for *Financial*, that is learned on the whole dataset. It yields a training accuracy of 100%. (Note that it is not the shortest tree possible.)

Model size. Next to an improvement in predictive accuracy, the use of aggregates also results in a decrease of model size. Again, complex aggregates do not seem to give further improvements. For *Mutagenesis* and *Diterpenes*, the model size even tends to slightly increase again.

Induction times and search space size. When comparing induction times and search space sizes, we see that going to the next more complex level of aggregation results in an increase of both measures. This increase is most explicit when moving from RA to LA. With the lookahead applied in aggregate conditions, the search space is significantly expanded, sometimes causing induction times to escalate. For example, for *Diterpenes*, where we already increased the minimal number of examples in the leaves, building one tree takes almost 7.5 hours.

Interestingly, in *Mutagenesis* and *Diterpenes*, the search space size decreases when going from NA to SA. For *Diterpenes* this also results in a decrease in induction time. An explanation for the decrease in search space size is that standard tests can introduce variables that may be reused lower in the tree, while aggregates can not. Thus, for the NA setting, nodes deeper in the tree will have a larger set of candidate refinements than in the SA setting (if aggregates are used higher in the tree). The fact that for *Diterpenes* this also results in a reduction of induction time for SA, may be attributed to the fact that most of the aggregates are *count* aggregates, which are faster to execute than *max*, *min*, and *avg*.

Artificial data. The experiment showed that the use of simple aggregates yields quite a large performance improvement on all data sets. Refinement of the aggregate conditions, on the other hand, increased accuracy only in a few cases, and in general the use of lookahead within the aggregate query also added only a slight performance improvement. The reason for these small improvements could be that the target concept simply does not require a combination of selection and aggregation. To test this conjecture and to know whether complex aggregates can add any improvements when the target concept does contain complex aggregates, we conducted experiments on artificially generated data where the target concept was defined to involve these complex aggregate functions. We used the Random Train Generator from Muggleton[3] for generating random "Michalski-style" train examples (Michalski 1980) according to a specified concept. We used the following concept:

$$\begin{aligned}
&eastbound(T) \leftarrow sum(W,(car(T,C),wheels(C,W)),SumW),SumW>7,\\
&\qquad\qquad count(C,(car(T,C),roof(C,none),NbC),\\
&\qquad\qquad NbC>1,!.\\
&westbound(T) \leftarrow sum(W,(car(T,C),wheels(C,W)),SumW),SumW>7,!.\\
&westbound(T) \leftarrow sum(L,(car(T,C),load(C,circle,L)),SumL),SumL>1,!.\\
&eastbound(T).
\end{aligned}$$

This concept states that if a train has more than 7 wheels and there is more than one car without a roof, the train goes east, else if only the first condition holds or if the train has at least 2 circular loads, it goes west. In all other cases it goes east. A dataset of 1000 trains was generated, half of them going to the west and the other half going to the east.

Table 4.12 gives an overview of predictive accuracy and model size for different levels of aggregation. The aggregate functions used include counting the number of cars, and taking the sum over the number of wheels and the number of loads. The table clearly shows an improvement of both accuracy and model size when going from simple aggregates (*SA*) to complex aggregates (*RA* and

[3]The train generator is available at
http://www-users-csyork. ac.uk/~stephen/progol.html.

Table 4.12: Accuracy and model size of Tilde with different levels of aggregation on the *Trains* data set.

	Tilde			
	LA	RA	SA	NA
accuracy	99.1% (0.6)	98.4% (0.5)	84.5% (1.7)	72.9% (1.4)
nodes	8.5 (1.9)	8.7 (0.2)	60.3 (4.3)	53.2 (6.6)

Table 4.13: Induction times for the LA experiment for Tilde-α and Tilde-μ on the *Mutagenesis* data set.

	Tilde-α	Tilde-μ
execution time	6641.3 (5.5)	1629.9 (66.4)
induction time	6672.2 (5.8)	1829.2 (73.4)

LA). This suggests that when the concept indeed involves complex aggregates, it is very useful to use higher levels of aggregation.

4.6.3.2 Comparing efficiency

For comparing efficiency results, we will concentrate on the LA setting, where efficiency really is a concern. We first compare the induction times for the two refinement operators and then check how this compares to the results for the first random forest induction algorithm.

Tilde-α versus Tilde-μ. The Tilde-LA experiment was rerun with the refinement operator based on μ-subsumption. The obtained results are indicated with Tilde-μ. Using this optimized refinement operator, the search space is structured differently, such that faster execution of the candidate tests on the examples is possible. Thus, the search space remains the same (which means that also the same tree is obtained), only the induction times are affected. Timings are reported in Tables 4.13, 4.14, and 4.15 for the three datasets. Next to induction time, also the execution time is reported, which is the sum of the time needed to execute all candidate tests on all examples for each node of the tree. A first observation when looking at the tables is that execution time corresponds to about 98% of the total induction time, thus, reducing this factor will be very rewarding w.r.t. the total induction time. Second, we indeed observe an efficiency gain: for the execution times we obtain speedup factors of 2.27 (*Financial*) to 4.07 (*Mutagenesis*), resulting in overall speedup ratios of 2.22 (*Financial*) to 3.65 (*Mutagenesis*).

Table 4.14: Induction times for the LA experiment for TILDE-α and TILDE-μ on the *Diterpenes* data set.

	Tilde-α	Tilde-μ
execution time	26153.9 (33.5)	7684.6 (12.9)
induction time	26701.2 (34.8)	8275.1 (6.9)

Table 4.15: Induction times for the LA experiment for TILDE-α and TILDE-μ on the *Financial* data set.

	Tilde-α	Tilde-μ
execution time	528.7 (1.3)	233.2 (0.7)
induction time	538.0 (1.4)	242.6 (0.7)

Tilde-α versus Forf. We have also experimented with the first order random forest induction algorithm (FORF). Note that in FORF also the refinement operator based on α-subsumption is used. We considered random subsets of 50%, 25%, 10%, and the square root of the number of tests at each node in the trees to test the influence of the size of the refinement sample. The effect of the size of the forest was tested by using forests of 3, 11, and 33 trees. For FORF the size of the search space decreases according to the sample ratio used at each node. Induction times for the LA setting are reported in Tables 4.16, 4.17, and 4.18, respectively. We see that, depending on the settings used, FORF may take more or less time than TILDE-α and TILDE-μ to induce a model.

4.6.3.3 Comparing predictive performance

We compare the predictive performance obtained for FORF to the performance of TILDE[4]. The accuracy as well as model size results for the different settings of FORF-LA are shown in Tables 4.19, 4.20, and 4.21, respectively. Accuracy is measured by averaging the out-of-bag accuracy of five independent runs of the

[4]Note that we do not compare the predictive performance of TILDE-α and TILDE-μ, since these are the same.

Table 4.16: Induction times for FORF-LA on the *Mutagenesis* data set.

	Forf-LA			
forest	50%	25%	10%	sqrt
3	2740 (116)	1303 (102)	602 (60)	37 (1)
11	10049 (428)	4780 (376)	2210 (220)	137 (3)
33	30148 (1285)	14342 (1130)	6630 (660)	412 (10)

Table 4.17: Induction times for FORF-LA on the *Diterpenes* data set.

	Forf-LA			
forest	50%	25%	10%	sqrt
3	44853 (121)	17911 (102)	6277 (90)	191 (8)
11	164463 (446)	65673 (376)	23017 (332)	701 (29)
33	493391 (1338)	197021 (1128)	69053 (997)	2104 (89)

Table 4.18: Induction times for FORF-LA on the *Financial* data set.

	Forf-LA			
forest	50%	25%	10%	sqrt
3	1045 (12)	540 (10)	153 (8)	28 (2)
11	3833 (44)	1982 (37)	561 (29)	105 (9)
33	11500 (132)	5946 (111)	1684 (89)	315 (28)

algorithm, and therefore is comparable to the accuracy results given for TILDE. Model size is measured as the sum of the number of internal nodes of all trees in the forest.

Looking at the predictive performances of the forests we see clear improvements w.r.t. TILDE-LA (shown in Tables 4.9 to 4.11). Thus, one may want to spend more time building the forest in order to achieve higher accuracies. We see that for *Mutagenesis* the accuracy increases from 73.4% for TILDE-LA to up to 80.0% for FORF-LA. For *Diterpenes* we observe a maximal improvement of 2.8% and for *Financial* a maximal improvement of 1.3% can be found. Also if a sampling ratio of, say 10%, is used, this increase in performance can be obtained if enough trees are used. Thus, with FORF one can obtain higher predictive accuracies compared to TILDE, while in general giving up on interpretability of the model and, depending on the settings used, giving up on efficiency.

4.6.3.4 Comparing to other systems

To compare TILDE and FORF with complex aggregates to other systems, we reran the algorithms (using one particular parameter instantiation for FORF) using tenfold cross-validation and did this five times with different folds to get more reliable estimates. We follow this approach since the previously published results of the other systems were also obtained by doing five times tenfold cross-validation.

Mutagenesis. Table 4.22 shows predictive accuracies of FORF-LA and TILDE-LA compared to other systems. All results in this table were obtained using

Table 4.19: Accuracy and model size results for FORF on the *Mutagenesis* data set.

		Forf-LA			
	forest	50%	25%	10%	sqrt
accuracy	3	73.4% (3.1)	74.4% (2.8)	74.7% (4.7)	74.1% (1.6)
	11	77.2% (1.9)	76.2% (1.5)	77.6% (1.7)	75.1% (1.8)
	33	78.7% (1.9)	80.0% (2.1)	79.1% (0.9)	79.6% (1.5)
nodes	3	49.4 (3.9)	58.0 (6.2)	55.2 (6.8)	61.6 (2.3)
	11	195.2 (8.4)	201.4 (6.2)	211.0 (17.6)	234.8 (8.1)
	33	579.0 (11.3)	606 (19.2)	636.5 (30.4)	692.8 (22.5)

Table 4.20: Accuracy and model size results for FORF on the *Diterpenes* data set.

		Forf-LA			
	forest	50%	25%	10%	sqrt
accuracy	3	84.1% (1.1)	83.9% (0.8)	83.8% (0.6)	77.8% (1.8)
	11	85.7% (0.9)	85.4% (0.2)	85.4% (0.3)	82.0% (1.0)
	33	87.2% (0.7)	86.4% (0.3)	86.6% (0.4)	85.4% (0.8)
nodes	3	27.1 (2.0)	27.2 (2.0)	30 (2.9)	44.4 (6.2)
	11	97.7 (3.1)	100.4 (5.4)	111.7 (7.8)	161.0 (3.6)
	33	301.5 (3.8)	319.6 (3.8)	330.2 (8.3)	469.2 (12.3)

Table 4.21: Accuracy and model size results for FORF on the *Financial* data set.

		Forf-LA			
	forest	50%	25%	10%	sqrt
accuracy	3	97.9% (1.5)	98.3% (0.7)	98.1% (1.5)	97.6% (1.1)
	11	99.3% (0.2)	98.9% (0.6)	98.7% (1.0)	98.9% (0.4)
	33	99.7% (0.2)	99.6% (0.4)	99.7% (0.2)	99.7% (0.4)
nodes	3	12.6 (4.6)	14.6 (3.1)	14 (1.2)	17.2 (5.9)
	11	51.4 (7.9)	53.2 (7.9)	51.8 (8.4)	64.2 (9.9)
	33	152.4 (11.4)	155.2 (15.5)	156.8 (7.6)	187.8 (19.9)

Table 4.22: Accuracy results on the "regression-friendly" *Mutagenesis* data set compared to other systems. The results for FOIL and PROGOL are obtained from (Srinivasan et al. 1995), and for ROLLUP from (Knobbe et al. 2001).

FORF-LA	TILDE-LA	TILDE	FOIL	PROGOL	ROLLUP
87.2% (1.3)	85.8% (1.5)	75.3% (1.5)	75% (3.0)	81% (3.0)	85%

tenfold cross-validation only on the "regression-friendly" part of the *Mutagenesis* data set, using background B2. FOIL (Quinlan 1990) induces concept definitions represented as function-free Horn clauses, from relational data. PROGOL (Muggleton 1995) is an ILP learner capable of learning in structurally very complex domains. ROLLUP (Knobbe et al. 2001) is a propositionalization approach that makes use of aggregates. For FORF-LA 33 trees and a sampling ratio of 25% were used. From Table 4.22, we see that TILDE-LA is at least competitive with the best of the other results and FORF-LA is able to lift the performance over the other systems.

Diterpenes. To allow a good comparison with previously published results for the *Diterpenes* dataset, the minimal leaf size of trees was reset to its default value of 2. Given the increased induction time, we report results for the SA setting. For FORF again 33 trees and a sampling ratio of 25% were used.

The result of these experiments is compared with published results for other first order systems in Table 4.23. RIBL (Emde and Wettschereck 1995) is a relational instance based learning algorithm. The ICL system (De Raedt and Van Laer 1995) uses exactly the same representation as TILDE, but induces rule sets instead of trees.

We only used the *red* relation for FORF-SA and TILDE-SA, since they should be able to construct the *prop* relation by themselves, but we compared them to the other systems both using only *red* and using *red* and *prop*. The difference between TILDE-SA with the *red* relation and TILDE with the *red* and *prop* relations is that the latter only counts atoms with a particular multiplicity whereas the former also uses the aggregate functions given in Sect. 4.6.1.2. It was already found that combining propositional (aggregate) features with relational information yielded the best results (Džeroski et al. 1998). Comparing with those best results, we see that FORF-SA outperforms the best of the other approaches.

Financial. Table 4.24 shows predictive accuracies for FORF-LA and TILDE-LA for *Financial* compared to other systems. The forest contained again 33 trees and a sampling ratio of 25%. DINUS-C (Lavrač and Džeroski 1994) is a propositionalization technique using only determinate features and using C4.5 rules as propositional learner. RELAGGS (Krogel and Wrobel 2001) is a propo-

Table 4.23: Accuracy results on the *Diterpenes* data set compared to other systems. The results for Foil, Ribl, Icl, and Tilde are obtained from (Džeroski et al. 1998) (no standard deviations were available for these results).

	Forf-SA	Tilde-SA	Tilde	Foil	Ribl	Icl
red	92.7% (0.4)	90.2% (0.7)	80.1% (0.7)	46.5%	86.5%	65.3%
red+prop			89.9% (1.0)	78.3%	91.2%	86.0%

Table 4.24: Accuracy results on the *Financial* data set compared to other systems. The results for Dinus-C, Relaggs, and Progol are obtained from (Krogel and Wrobel 2001).

Forf-LA	Tilde-LA	Tilde	Dinus-C	Relaggs	Progol
90.8% (1.7)	87.9% (1.5)	87.3% (1.1)	85.1% (10)	88.0% (7)	86.7% (7)

sitionalization approach where aggregates are used to summarize related individuals. Progol (Muggleton 1995) is an ILP learner capable of learning in structurally very complex domains.

The results for Dinus-C, Relaggs, and Progol are taken from (Krogel and Wrobel 2001), where they were obtained from experiments on a modified version of the *Financial* dataset. In particular, all transactions dated after the granting of the loan were removed. This causes the average number of transactions per example to drop from 388 to 75. The motivation for this data reduction is that, if the model is to be used to classify customers who apply for a loan, only data known at the time of granting the loan can be used. It turns out that this is a more difficult learning setting. For our comparison, the systems Tilde, Tilde-LA and Forf-LA were run on the same reduced dataset.

The table shows that Tilde-LA's predictive performance is very close to Relaggs's, but has a lower variance. Forf-LA adds another 3% to the accuracy, yielding the best result.

4.7 Conclusion

In this chapter we have discussed how to add complex aggregates to Tilde's hypothesis space. The basic components of aggregate conditions are provided by the user in the language bias specification. From these components (i.e., aggregate functions, aggregate queries, threshold values, and comparison operators) the system then constructs simple aggregate conditions and adds them to the search space traversed by Tilde. Complex aggregate conditions are obtained by refining an aggregate condition that occurs higher in the tree, or by including them directly by using a lookahead procedure inside the aggregate query. We have explained that invalid refinements are not a problem in TILDE, and therefore it is possible to use a simple refinement operator based on α-subsumption.

An issue to cope with when introducing complex aggregate conditions in TILDEis the significant increase in size of the search space, which is very explicit in the case of lookahead. We have described two techniques to deal with this problem. The first technique upgrades TILDEto a first order random forest induction system. Essentially, an ensemble of decision trees is constructed in which only a sample of all possible tests is tried at each node. In the second technique we retain the complete search space but traverse it in an efficient way. To this end, we implement the refinement operator based on μ-subsumption, which was introduced in the previous chapter, into TILDE.

Experimental evaluation showed that the capability of learning aggregates brings clear performance improvements. The largest improvements are due to the use of simple aggregates. The use of complex aggregates turns out to be useful if the target concept involves complex aggregates, as was shown by the use of an artificial dataset.

The use of the refinement operator based on μ-subsumption results in a significant efficiency gain. Upgrading the system to a random forest learner can increase or decrease induction times, depending on the number of trees and sampling ratio used, while improving the predictive performance.

Chapter 5

Constructing Predictions with Complex Aggregates

5.1 Introduction

In the previous chapters we have explained that one-to-many and many-to-many relationships in a relational database may lead to non-determinate attributes for the objects in the target table. These non-determinate attributes are typically summarized by testing for the existence of a specific value, or by aggregating them. In Chapt. 3, we have proposed complex aggregates, which are a combination of both approaches. Up till now, the complex aggregates that we used occurred in the *if-part* (antecedent) of the hypothesis. The *then-part* (consequent, conclusion) was always a fixed value (nominal for classification and numeric for regression). However, in some patterns, e.g., linear equations (see Sect. 2.2.1), predictive attributes are involved in the consequent of the hypothesis. In relational learning, these predictive attributes may be non-determinate, each possible value resulting in a different predictive function. In this chapter we consider the use of complex aggregates to summarize the predictive attributes in the conclusion part of a hypothesis.

We will focus on model trees, which are a combination of linear equations and regression trees and, hence, are another example of patterns that involve predictive attributes in the consequent. More precisely, model trees are regression trees that contain some non-trivial, usually linear, model in their leaves. In the propositional case, they have been shown to be able to increase predictive performance compared to regression trees that predict the same constant value for each example falling into the same leaf (Alexander and Grimshaw 1996; Karalic 1992; Malerba et al. 2004; Quinlan 1992; Torgo 1997). While classification and regression trees have been around in ILP for several years now (Blockeel and De Raedt 1998; Kramer 1996; Kramer and Widmer 2001), less can be said about model trees. Probably the main reason for this is the problem mentioned above, i.e., that the predictive attributes to be included in the regression functions may be non-determinate.

We distinguish a number of approaches to handle non-determinate predictive attributes in regression functions:

1. do not use non-determinate attributes (Kramer and Widmer 2001)
2. assume one of the instances is relevant
   a) and it can be specified with conditions (Karalič and Bratko 1997)
   b) and it can not be specified (Ray and Page 2001; Srinivasan 2003)
3. summarize the instances
   a) using simple aggregate functions defined in advance (Appice et al. 2003)
   b) using complex aggregate functions

Complex aggregates were introduced in Chapt. 3 and there we have studied how they can be efficiently learned to be included in the condition part of a hypothesis. We have proposed a refinement operator by which complex aggregate conditions are learned general-to-specific, subsequently reducing the coverage of the hypothesis. In Chapt. 4, we have extended an existing decision tree algorithm to learn complex aggregates in the internal nodes. The algorithm can output both classification and regression trees. In this chapter, we lift the algorithm to a model tree learner that constructs regression functions with complex aggregates in the leaves. These complex aggregates are not learned from scratch at the leaves, since this would be very time consuming. Instead, complex aggregates are included in the regression model of a leaf if they have shown a linear effect with the target during the tree building process. This means that the heuristic function employed needs to take into consideration the construction of linear models at the leaves. It has been shown that model tree learners produce good results if such a heuristic function is used (Malerba et al. 2004). However, most (propositional and relational) systems that use a heuristic that takes into account linear functions are quadratic (Alexander and Grimshaw 1996) or cubic (Karalic 1992; Malerba et al. 2004; Appice et al. 2003) in the number of numeric attributes. Given the fact that we want to include complex aggregates in the search, the number of numeric attributes can become very large, which renders existing systems infeasible to use. Therefore, an important requirement for our system is an efficient heuristic function.

In Sect. 5.2 we present some related work on learning regression functions and model trees in relational learning. Section 5.3 focusses on heuristics for learning model trees and proposes a new heuristic that meets our requirements. Section 5.4 then presents our relational model tree learner. Experimental results are presented in Sect. 5.5. Finally, in Sect. 5.6 we conclude.

5.2 Related work

The task of relational regression was formalized by Džeroski (1995) in the normal ILP framework. This work presents the transformation based system Di-

NUS, which is the first ILP system to address the task of relational regression. The induction is delegated to a propositional learner. Using a model tree learner as RETIS (Karalic 1992), linear regression is used in the model output by DINUS.

FORS (Karalič and Bratko 1997) is the first system able to predict numbers with non-determinate background knowledge. It is a sequential covering approach that learns rules that contain linear regression models. Non-determinacy among the predictive attributes is handled by testing for the existence of a specific instance giving a number of conditions. If the conditions succeed for more than one instance, the value of the first of these instances is taken.

TILDE (Blockeel and De Raedt 1998) and SRT (Kramer 1996) are first order regression tree learners. S-CART (Kramer and Widmer 2001), the successor of SRT, is capable of including linear models in the leaves. The use of non-determinate predictors in these linear models is not supported. The model trees induced by S-CART are built by first constructing a normal regression tree (using a standard variance reduction heuristic), and afterwards replacing the constant predictions by linear models. This heuristic has been shown to produce sub-optimal model trees in the sense that it tends to split the data set in the wrong places and results in trees that are larger than necessary (Malerba et al. 2004; Torgo 2002).

Appice et al. (2003) present a system called MR-SMOTI which is a relational upgrade of their propositional SMOTI model tree algorithm (Malerba et al. 2004). The SMOTI algorithm is different from most model tree inducers in the sense that the multiple linear model that is associated with the leaves is built incrementally from simple linear regression models. These models are introduced by so-called regression nodes occurring in the tree. Each regression node thus adds one term to the multiple regression model and requires updating the target value and other continuous attributes in order to remove the linear effect of the introduced term. To determine the coefficients of a simple linear regression model in a regression node of a tree induced by MR-SMOTI, the problem is locally transformed into a propositional problem by joining the tables from the underlying relational database structure, and normal least squares is applied on this flattened table. Note that this propositionalization step gives examples that have a higher number of related objects more weight in the least squares procedure. The predicted value for unseen examples is the average prediction for all instances in the propositional representation of the example. Contrary to the efficient methods as S-CART, the systems SMOTI and MR-SMOTI have a high computational complexity. This is due to the heuristic function, which takes into account the fact that linear models are built. It is discussed in more detail further in this chapter.

5.3 Finding a suitable heuristic for learning model trees

In this section, we concentrate on finding a suitable heuristic for our relational model tree learner. The most important requirement is that it is efficient to compute. This is necessary in order to deal with large search spaces that contain complex aggregates. In order to find a suitable heuristic, we first examine existing model tree learners. Since in propositional learning model tree systems abound, we concentrate on this setting for now.

The most efficient group of systems use what we call a variance based heuristic. This is the same heuristic as used by standard regression tree learners: reduction of variance (or variations of variance, such as standard deviation, or sum of squared errors). However, in Sect. 5.3.1, we show that model tree learners employing this heuristic may exhibit pathological behaviour in some quite simple cases. This is not visible in the predictive accuracy of the tree, but it reduces its explanatory power. Section 5.3.2 discusses more complex approaches. This discussion leads to the identification of four levels of complexity, where the second level represents a heuristic that overcomes the problems of the variance based approach, and this at little additional computational cost. In Sect. 5.3.3 we elaborate on this approach and propose a propositional model tree learner using such a heuristic. The resulting model tree induction algorithm is experimentally evaluated and compared with simpler and more complex approaches on a variety of synthetic and real world data sets in Sect. 5.3.4. In the remainder of this chapter we then show how the heuristic function and a number of other aspects of the algorithm are adopted by our relational model tree learner.

5.3.1 The variance based approach

Existing regression tree algorithms are instantiations of the the Top Down Induction of Decision Trees (TDIDT) algorithm, see Table 2.1 for a generic variant.

TDIDT algorithms vary mainly with respect to the heuristic they use to decide which partitioning is best and the model that is stored in each leaf. For instance, CART (Breiman et al. 1984), when building regression trees, uses variance reduction as a heuristic (that is, it tries to reduce the variance of the target variable within the subsets as much as possible), and the model stored in a leaf is the mean of the target values of all examples in that leaf. M5 (Quinlan 1992), a popular algorithm for building model trees with linear models in the leaves, and M5' (Wang and Witten 1997), a re-implementation that is included in the well-known Weka software (Witten and Frank 1999) also use variance[1] reduction as a heuristic. They fill in constants in the leaves, but in the pruning

[1]Or variations of variance, such as standard deviation (Wang and Witten 1997) or the 5th root of the variance, as in the Weka implementation.

Table 5.1: M5' instantiation of TDIDT.

```
procedure LOCAL_MODEL ($E$: examples):
    return linear regression model for $E$
    (based on variables occurring in the subtree)

procedure QUALITY ($S$: split, $E$: examples):
    $\mathcal{P}$ := partition induced on $E$ by $S$
    return $SD(E) - \sum_{E_j \in \mathcal{P}} |E_j|/|E| SD(E_j)$
    ($SD(T)$ = target variable's standard deviation in $T$)

procedure STOP_CRITERION ($E$: examples):
    return $|E| < 4$ or $SD(E) < 0.05 * SD(\text{all examples})$
```

phase they change internal nodes into leaves containing a linear model if that model performs at least equally good as the subtree rooted in that node. The linear model uses as predictor variables only attributes occurring in the subtree that it replaces. Hence, M5 and M5' can be described as an instantiation of the generic TDIDT algorithm. Table 5.1 shows the instantiation of the *local_model*, *quality*, and *stop_criterion* functions.

Keeping in mind that linear models will be built in the leaves, it seems reasonable to select tests that maximize the expected quality of these models. In that sense, variance reduction is not a very suitable heuristic. As also noted elsewhere (Karalic 1992; Malerba et al. 2004), the quality of a linear model constructed for a data set is quite independent of the variance of that data set. For instance, a set of points that form a perfectly linear but steep line has a greater variance than a set of points closer together but randomly distributed. Given this independence, there is no reason why variance would perform better than random splitting. In fact, it may even be worse, as the following simple example shows.

Consider the piecewise linear function $y = x, 0 \leq x \leq 1$; $y = 2 - x, 1 < x \leq 2$ (see Fig. 5.1(a)). Clearly the simplest model tree to represent this function is one that splits on the condition $x < 1$ and builds two linear models.

In the following, we consider a continuous uniform distribution of (x, y) couples. Note that for the optimal tree, the variance along y of both subsets is equal to the variance along y of the original set; thus, from the point of view of reducing variance, this tree would not be considered a good tree. The split $x < c$ that is found by M5' is the one that minimizes the sum of the weighted variance over both subsets. For a continuous uniform distribution, the optimal c can be computed by minimizing

$$h(c) = c \cdot Var(y|x \leq c) + (2 - c) \cdot Var(y|x > c),$$

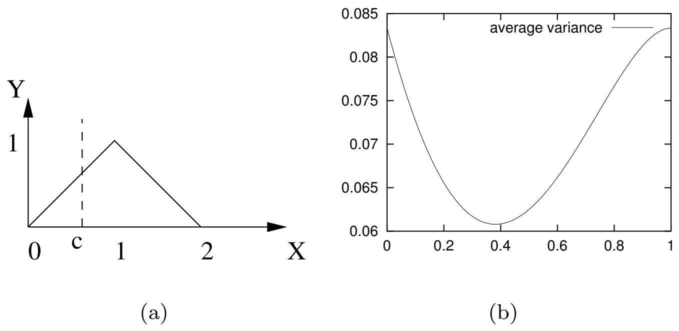


Figure 5.1: (a) A piecewise linear function and split point c. (b) The weighted mean of the variance in the subsets defined by $x \leq c$ and $x > c$ as a function of c.

where the factors c and $2 - c$ are weights referring to the respective sizes of the subsets and the *Var* factors are the variances of these subsets. The optimal c is then defined as

$$c^* = \arg\min_c h(c).$$

Due to symmetry, $2 - c^*$ is an optimum in the interval [1,2].

The function $h(c)$ is plotted in Fig. 5.1(b), and there it can clearly be seen that a minimum is obtained near 0.4; that is, quite far away from the optimal split point 1. In fact, even random splitting can be expected to yield slightly better results on average, since it has 60% probability of creating a split closer to 1.

Without loss of generality, given the symmetry, we assume $c^* < 1$. The left subset yields a perfect linear model. However, as the model is not actually constructed as part of the stopping criterion, M5' constructs a subtree for this subset, partitioning this linear area into many small areas with ever decreasing variance; only in the pruning phase does it remove this subtree to build the linear model. The right subset is a function similar to the original one, though less symmetric. Hence, similar behaviour can be expected. Note that in contrast to the left branch, the right branch cannot be pruned to a single leaf; and that also further down the tree, the splits will always be off the optimal point. Thus a relatively deep tree will still remain.

The above reasoning shows that variance-based heuristics tend to split the data set in the wrong places. This reduces the explanatory power of the tree: first, a larger tree is obtained than is necessary, and second, the splits in the tree are not really informative. Moreover, superfluous partitioning of the data space causes relatively small areas to be separated, with each area containing few examples. As local models built from few examples tend to be less accurate, the predictive performance of the induced trees may also be reduced.

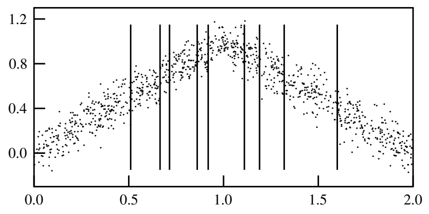
In our experiments section (Sect. 5.3.4) we further explore this behaviour of M5'. At this point, we just illustrate the concrete behaviour of M5' on the simple function mentioned above, for a random sample of data points with some noise added. Figure 5.2 shows the tree M5' builds for such a data set. It confirms our theoretical analysis: the model built by M5' is good in the sense that it has a reasonably good predictive performance, but its explanatory power is diminished.

Another system that uses variance reduction as a heuristic is Htl (Torgo 1997). The major difference with M5 is that it is a hybrid system that can integrate several alternative models into the tree leaves. However, since we are interested only in the case where linear models are used in the leaves, we will not elaborate on Htl.

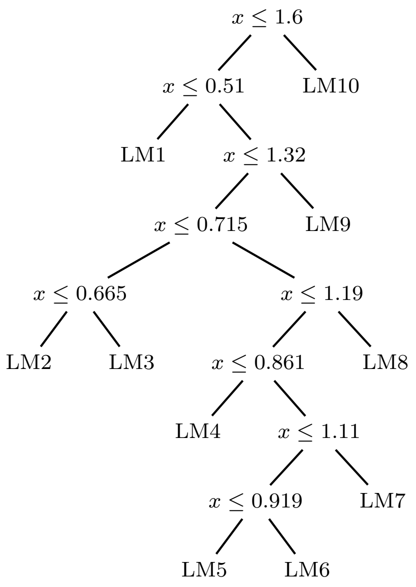
5.3.2 More complex approaches

Our simple case above not only shows that variance does not work very well as a heuristic, it also suggests that constructing linear models for the subsets and evaluating their goodness of fit should work fine. This is exactly the approach followed by Retis (Karalic 1992). Retis evaluates a split by building a multiple linear model for each subset and computing its residual variance. Also in the leaves multiple linear regression models are built. The Retis instantiation of TDIDT is shown in Table 5.2. Unfortunately, these multiple regressions result in a complexity that will not scale up to large problems. Torgo (2002) has presented a set of incremental formulae that allows to evaluate the set of candidate splits for a given variable with complexity quadratic in the number of attributes. Hence, the complexity for evaluating all splits in a node is cubic in the number of attributes, which is still too high for many practical applications. Besides its high computational cost, another drawback of Retis is that only full regression models (i.e. regression models in all variables) are considered in the leaves and in the quality function. Next to the obvious fact that understanding a smaller model is usually easier than understanding a larger one, it is well-known that if some of the predictors are related to each other, that is, they are (approximately) collinear, models based on subsets may give more precise results than will models based on more variables (Weisberg 1980). Therefore, variable selection is a desirable part of many regression analyses. In the case of Retis, applying variable selection techniques when choosing the best split may become too costly, since it may require the computation of extra multiple regression models (Malerba et al. 2004).

A number of systems have been proposed that were designed to address the computational concerns of Retis. They can be divided into two groups. The first group uses multiple linear regression models in the leaves, but at each node transforms the regression problem into a classification problem in order



(a)



LM1: y = 0.003 + 1.010x
LM2: y = 0.577
LM3: y = 0.713
LM4: y = 0.470 + 0.411x
LM5: y = 2.163 - 1.449x
LM6: y = 1.260 - 0.296x
LM7: y = 0.859
LM8: y = 0.730
LM9: y = 1.630 - 0.744x
LM10: y = 1.806 - 0.895x

(b)

Figure 5.2: (a) A simple data set. The vertical lines in the data plot indicate where the split points of a variance based model are. (b) The model induced for this dataset by a variance based model tree builder.

Table 5.2: RETIS instantiation of TDIDT.

```
procedure LOCAL_MODEL (E: examples):
    return linear regression model for E
    (based on all variables)

procedure QUALITY (S: split, E: examples):
    $\mathcal{P}$ := partition induced on E by S
    return $RVAR(E) - \sum_{E_j \in \mathcal{P}} |E_j|/|E| RVAR(E_j)$
    (RVAR(E) = residual variance of a multiple
    linear model built from E)
```

to use more efficient search strategies. Examples are SUPPORT (Chaudhuri et al. 1994), GUIDE (Loh 2002), and SECRET (Dobra and Gehrke 2002). The first two systems make use of statistical methods for variable selection. At each node, a (constant or linear) model is fit, then the signs of their residuals are used to separate the observations into two classes and a statistical test is used to pick the variable giving the largest separation between the two classes. The two systems differ in the test they use and in the way to determine the split point, once the variable is found. However, as Dobra and Gehrke (2002) point out, these approaches may also lead to redundant splits. SECRET uses the EM algorithm (Dempster et al. 1977) at each node to determine two Gaussian clusters in the data and to label the data based on the closeness to these clusters. Classification tree techniques are then used to select the split attribute and the split point.

The second group retains the regression problem at each node, but fits a simpler regression model to evaluate a split. In this work, we will focus on the second group. Obviously, M5 is an example of this approach, since it fits the most basic model at the nodes: the average of the sample data at that node.

The simple example described in the previous section deals with a single predictor variable, which is uncommon. Consequently, the variable used to define the split is the same as the variable used in the regression. When multiple predictor variables exist, a number of different approaches are possible, with RETIS and M5 at both ends of the spectrum. We characterize each of them by listing the variables for which it performs regression *in order to evaluate the split* (not to be confused with the construction of linear models in the leaves), given the variable it splits on. We consider only univariate splits here. All these options could be extended by considering also multivariate splits (Brodley and Utgoff 1995), but this extension is more or less orthogonal to the dimension discussed here, and hence out of scope for us.

The four options we distinguish are:

1. no regression (the M5 approach)

Table 5.3: TREED REGRESSION instantiation of TDIDT.

```
procedure LOCAL_MODEL (E: examples):
    return simple linear regression model for E

procedure QUALITY (S: split, E: examples):
    $\mathcal{P}$ := partition induced on E by S
    return $-\sum_{E_j \in \mathcal{P}} |E_j|/|E| RSSE^*(E_j)$
    (RSSE*(E) = smallest residual sum of squared errors of a
    simple linear model built from E)

procedure STOP_CRITERION (E: examples):
    return $|E| < 20$
```

2. simple regression on the split attribute if it is numerical

3. simple regression on all numerical attributes separately

4. multiple regression on all numerical attributes together (the RETIS approach)

Our reason for listing exactly these different options is that they differ significantly with respect to their computational complexity: options 1 and 2 are linear in the number of attributes, option 3 quadratic and option 4 cubic. All of these complexity factors are multiplied with at least N, with N the number of examples.

An example of a system that takes the third option is described by Alexander and Grimshaw (1996) and is called TREED REGRESSION. To evaluate a split, say, on attribute X_j, each independent attribute is evaluated as the regressor attribute for a simple regression. The best linear model is determined for each child node independently. The algorithm stores simple linear regression models at each leaf. The TDIDT instantiation is shown in Table 5.3.

Another example of an option 3 system is SMOTI (Malerba et al. 2004). It induces an alternative kind of decision trees, where, besides splitting nodes, regression nodes can be introduced. Whereas the former partition the sample space, the latter perform only simple linear regression. The regression is done in such a way that the multiple linear model associated to each leaf is obtained by composing the effect of regression nodes along the path from the root to the leaf. The purpose of this tree building mechanism is to discover global effects that some attributes may have in the underlying model function. In order to evaluate a split, simple regression is performed on all attributes separately and combined with the regressions associated to regression nodes along the path from the root to the node. In order to evaluate a regression node, a lookahead step is performed, which actually renders this method cubic in the number of attributes (Malerba et al. 2004).

5.3.3 A simple linear regression based heuristic

In this section we present an algorithm that uses a simple linear regression based heuristic, hence it follows option 2 from the previous section. We start by motivating our approach in Sect. 5.3.3.1 and discuss the algorithm in Sect. 5.3.3.2.

5.3.3.1 Motivation

To our knowledge, there exists no system that uses option 2, although such a system would have some interesting properties:

- It is expected to provide a solution for the undesirable behaviour that systems using option 1 exhibit. More specifically, it should work better in those cases where at least one variable has an influence on the target variable that is non-linear but can be approximated with a piecewise linear function.

- It has a computational complexity close to that of option 1, differing only with a constant factor. Variance, $\sigma^2 = \sum_i (y_i - \bar{y})^2/n$ with $\bar{y}$ the mean of the y_i, can be computed entirely from the sufficient statistics $\sum_i(1, y_i, y_i^2)$ with i varying over all elements of the data set. Similarly, a simple linear regression model as well as its residual variance can be computed from $\sum_i(1, x_i, x_i^2, y_i, y_i^2, x_i y_i)$. In our experience, this takes about three times longer. Computing the residual variance from these statistics is about four times as much work as computing the total variance from them. Hence, this heuristic is roughly three to four times as expensive to compute as the one in M5. In fact, since smaller trees can be expected, the slightly more complex computation might well be compensated.

Obviously, for each of the options 1-3 it is possible to point out situations where the next more complex option performs better. The point that we wish to make, is that the move from option 1 to 2 solves at least some problems, and costs almost nothing with respect to efficiency.

5.3.3.2 The algorithm

The model tree induction algorithm we propose follows option 2 and is a variant of M5' that we call MAUVE ("M5' Adapted to use Uni-VariatE regression"). The MAUVE instantiation of TDIDT is shown in Table 5.4. The main difference with M5' lies in the heuristic, which is based on simple regression. Also the stop criterion and pruning method are altered to be better tuned towards the heuristic. We discuss each of these, as well as other components of the algorithm

Table 5.4: Mauve instantiation of TDIDT.

```
procedure LOCAL_MODEL ($E$: examples):
    return linear regression model for $E$
    (based on all variables)

procedure QUALITY ($S$: split, $E$: examples):
    $\mathcal{P}$ := partition induced on $E$ by $S$
    return $RSD(T) - \sum_{T_j \in \mathcal{P}} |T_j|/|T| RSD(T_j)$
    ($RSD(T)$ = residual standard deviation of a
    simple linear model built from $T$)

procedure STOP_CRITERION ($E$: examples):
    return $|T| < 2 * nbattributes$ or
        $SD(T) < 0.05 * SD(all\ examples)$
```

(pruning and smoothing). The algorithm is implemented in the Weka software (Witten and Frank 1999), which also contains the M5' algorithm.

Quality function. A model tree is built by calling the GROW_TREE function (see Table 2.1) using all examples. For each input variable X_j, we pass through the possible split points. If X_j is numeric, each possible split takes the form $X_j < v$ with v an element from the domain of X_j. The splits are evaluated by calculating, for each subset of the partition induced by the split, the residual standard deviation of the target variable after fitting a simple linear regression model with X_j as regressor.

If X_j is a nominal variable, say it takes values in $V = \{X_{j1}, X_{j2}, ..., X_{jm},\}$, then splits take the form $X_j \in V'$ with $V' \subset V$, i.e. we consider only binary splits. Each of these splits can be seen as testing the value of a binary variable (a variable that has value 1 for those examples with $X_j \in V'$, and value 0 otherwise). Since the residual standard deviation is undefined when using binary variables, these splits are evaluated using the (normal) standard deviation of the target variable. If X_j can take m possible values, there are 2^{m-1} different binary partitions possible, so we need to consider 2^{m-1} binary variables. However, since the binary splits minimize the (normal) standard deviation, we can make use of a result by Breiman et al. (1984) to reduce the number of binary splits to be considered to $m - 1$, as in the implementation of M5'. To this purpose, the average target value corresponding to each possible value for X_j is computed and the values in the enumeration are sorted according to these averages. Then X_j is replaced by $m - 1$ binary attributes, the ith being 0 if the value is one of the first i in the ordering, and 1 otherwise.

For each input variable X_j, the calculations of the (residual) standard deviation are performed incrementally, by sorting the examples by their X_j-value, and by updating a set of statistics $\sum_i(1, x_i, x_i^2, y_i, y_i^2, x_i y_i)$ for each possible split point. From all possible splits, we choose the one that maximizes the reduction in (residual) standard deviation. The function is recursively applied to the subsets of the partition induced by the chosen split.

Local models. The occurrence of a certain split attribute X_j at a node N reflects the existence of a linear relation between the target and X_j in the child nodes of N. Hence, X_j should be included in the linear model of the leaves in the subtree of N. Since attributes not occurring on a particular path may also be relevant, the model stored in each leaf is a multiple linear regression model that considers all input attributes as predictor variables. As in M5', variable selection techniques are applied, in order to increase the reliability of the estimated regression coefficients, and hence to minimize the difference between predicted target and actual target.

Stop criterion. The stop criterion of the GROW_TREE function was slightly modified. M5' doesn't split a node if it contains less than four instances, because the minimum number of instances in a leaf is two. However, when building a linear model in k variables in the leaves, one needs at least $k + 1$ examples. Taking into account that we consider all predictor variables to be included in the model, we need at least a number of examples equal to the number of predictors plus one in each leaf. Hence, at least $(2j + 2)$ examples, with j the number of predictor variables, are needed to split a node. The second part of M5's stop criterion (see Table 5.1) was left unchanged.

Pruning. In the pruning phase, each interior node N is associated with two error measures. The first is called the model error $ME(N)$ and is the error that would be experienced if the node were a leaf (hence, a multiple regression model is built at the node). The second is the subtree error $SE(N)$ and measures the error if the node keeps its child nodes. The tree is pruned at N (i.e. its subtrees are discarded and the node becomes a leaf) if $ME(N) < SE(N)$. In order to compensate for the underestimation of these errors on unseen cases, they are multiplied by $(n + m * p)/(n - p)$ with n the number of examples reaching the node N, p the number of parameters, and m the pruning factor, a multiplier that controls the size of the tree (Wang and Witten 1997). The number of parameters is the sum of the number of nodes in the subtree starting at N and the number of variables occurring in the linear equations of the leaves in the same subtree. The pruning factor m has default value 2 in M5', which is taken over in MAUVE. Of course, for both systems, performance can be improved by using cross validation to estimate an optimal value (see Sect. 5.3.4.2).

Smoothing. Smoothing (Quinlan 1992) is a method for improving the predictions of a model tree. It can be used in the final stage of the tree building process. It compensates for the sharp discontinuities that may occur between adjacent linear models at the leaves of the pruned tree. A weighted average between models in the leaves and linear models higher up in the tree is taken to make predictions. Smoothing may substantially increase the accuracy of predictions, without altering tree size.

5.3.4 Experiments

This section serves to evaluate the performance of our propositional model tree learning system and to compare it with traditional variance based approaches and with more sophisticated approaches. In particular, three questions are addressed:

1. If there is a (piecewise) linear relation between the target and (at least) one of the input variables, does our approach perform better than a variance based approach? If yes, at which computational cost?
2. How often does such a linear relation occur in practice?
3. Does the use of more sophisticated approaches result in a performance improvement?

In order to answer the first question, experiments were performed on synthetic data sets, specifically constructed with (piecewise) linear relations. Results are presented in Sect. 5.3.4.1. The second question is tackled by conducting experiments on real world data sets (Sect. 5.3.4.2). The third question is dealt with by performing a more complete comparison of our approach to other existing approaches on the same real world data sets (Sect. 5.3.4.3).

All experiments were performed using tenfold cross validation and were run twenty times in order to account for the variance between the folds. The relevant criteria are predictive performance, tree size, and induction time. Predictive performance is estimated using the RRMSE[2], a measure output by Weka. Tree size is measured in number of leaves. Since the number of leaves alone does not give a good estimation of the complexity of the model (which is also influenced by the number of variables in the linear equation in the leaves), we also report the number of parameters. The number of parameters is the sum of the number of nodes in the tree and the number of variables occurring in the equation in each leaf.

For our experiments, standard settings in Weka were used, except for smoothing, which was initially switched off.

[2]RRMSE = root relative mean squared error, this is the root of the ratio of the mean squared error of the tree to the mean squared error of a trivial model always predicting the mean.

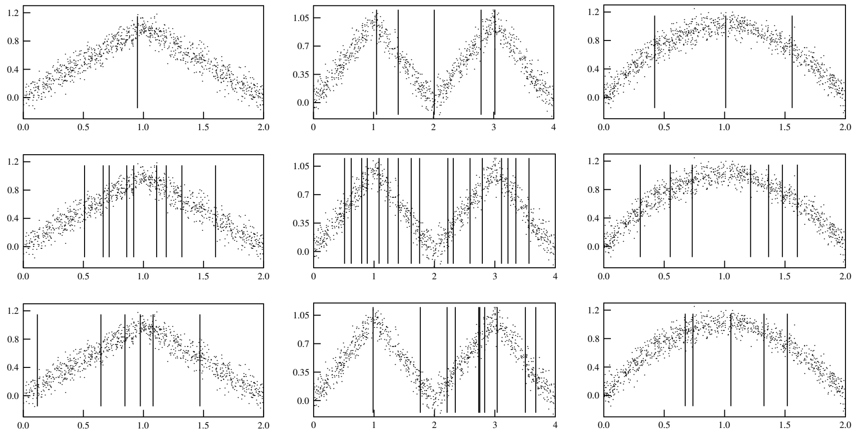


Figure 5.3: Approximating functions of one variable. Three different functions are shown, we call them (from left to right) *function1*, *function2* and *function3*; for each we indicate the thresholds that are created by (from top to bottom) Mauve; M5' with its original variance heuristic; M5' with random splits.

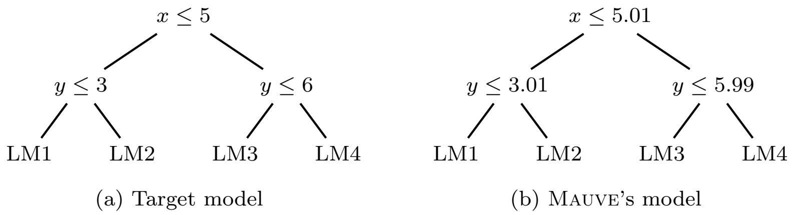
5.3.4.1 Synthetic data sets

Figure 5.3 shows some simple functions, in which there is a piecewise linear relation (or a relation that can be approximated by it) between the target and the input variable. For each function $f(x)$, a data set was constructed by drawing a random sample of 1000 values from a uniform distribution over x, and associating with each x a $y = f(x) + \epsilon$ with ϵ a normally distributed random variable with mean 0 and standard deviation σ that represents noise. Varying σ had no significant influence on the interpretation of our results.

For each function, Fig. 5.3 shows the partitioning created by model tree builders using simple regression, variance, and random splits. These results confirm our earlier findings: for a variety of functions, Mauve tends to find simpler models, with fewer split points that intuitively make more sense. Introducing random splits works approximately as well as using variance-based splits, or even slightly better.

We have also constructed piecewise linear functions in multiple predictor variables. Figure 5.4 presents a representative example problem. A dataset of 700 examples was constructed according to the target model in Fig. 5.4(a), this time no noise was added. From Fig. 5.4(b) and Fig. 5.4(c), we see again that Mauve produces a model that is much shorter than that of M5'.

Some statistics on the induced trees are presented in Tables 5.5 and 5.6. They confirm that the models induced by Mauve are always simpler without causing any loss of accuracy. In fact, for these datasets, predictive performance



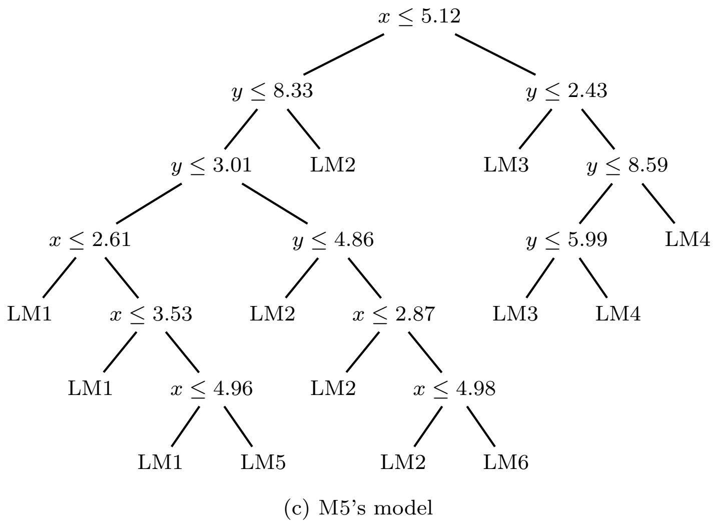


Figure 5.4: Learning a piecewise linear function of two variables (*function4*). (a) The original function. (b) The model tree built by MAUVE. (c) The model tree built by M5'.

Table 5.5: Predictive accuracy (RRMSE) for synthetic datasets when using simple regression (MAUVE), variance (M5'), and random heuristics (RND).

	MAUVE	M5'	RND
Data set	RRMSE (StdDev)	RRMSE (StdDev)	RRMSE (StdDev)
function1	**33.99%** (0.04)	34.35% (0.12)	34.39% (0.22)
function2	**32.96%** (0.13)	34.72% (0.21)	33.61% (0.37)
function3	**31.39%** (0.06)	31.70% (0.20)	31.63% (0.53)
function4	**16.40%** (3.05)	21.83% (1.85)	68.64% (2.97)

Table 5.6: Tree size and induction times for synthetic datasets when using simple regression (MAUVE), variance (M5'), and random heuristics (RND).

	Size						CPU time		
	MAUVE		M5'		RND		MAUVE	M5'	RND
Data set	Leaves	Pars	Leaves	Pars	Leaves	Pars	Secs	Secs	Secs
function1	**2**	**5**	10	25	5	13	3.75	3.41	3.87
function2	**6**	**17**	18	44	12	32	3.95	3.51	3.45
function3	**4**	**11**	8	20	6	16	3.11	3.95	2.89
function4	**4**	**15**	13	47	9	28	1.67	1.51	2.23

is improved. Moreover, the induction times for MAUVE are similar to those of M5', which suggests that the tree building procedure stops earlier and that this effect compensates for the more complex heuristic.

We conclude that, if we know that there is a simple piecewise linear relation (or a relation that can be approximated by it, such as *function3*) between the target attribute and (at least) one of the input attributes, MAUVE outperforms variance based methods. The question remains whether this linear behaviour also occurs in real life data sets.

5.3.4.2 Real world data sets

We have compared the performance of the different heuristics on a number of UCI data sets (Merz and Murphy 1996) and on some datasets taken from the collection of regression datasets by Luis Torgo[3]. Most of these datasets have been used as benchmarks in other studies on regression trees and model trees. For each dataset, we list the number of examples, the number of predictor attributes, the number of numeric predictor attributes, and the origin in Table 5.7.

Standard experiments. In this section, we compare the basic versions of M5' and MAUVE, i.e., with a fixed pruning factor (having M5's default value of 2) and without using the smoothing operator. Alternative parameter settings will be discussed in the following sections.

Table 5.8 shows predictive accuracies. Due to the high variation in the RRMSE, we also compared the error for MAUVE and M5' for each random seed of the cross validation and report the number of times each system won. As noted by Dietterich (1998), the use of significance tests based on the means and standard deviations from repeated cross validations is not valid, since null hypotheses will nearly always be rejected given enough repetitions. Therefore, we don't make any statements regarding significance. Instead, we emphasize the results in Table 5.8 if one method outperforms the other one in at least 90%

[3] http://www.liacc.up.pt/~ltorgo/Regression/DataSets.html

Table 5.7: Description of real world data sets.

Data set	Examples	Attributes	Origin
Housing	506	13 (12 num.)	UCI
Machines	209	7 (6 num.)	UCI (without unique model name and linear regression prediction of target)
Auto-Mpg	398	7 (4 num.)	Torgo
Auto-Price	159	15 (14 num.)	Torgo
Abalone	4177	8 (7 num.)	UCI
Servo	167	4 (0 num.)	UCI
Wisconsin	198	34 (33 num.)	UCI (prognostic dataset)
Kin8NM	8192	8 (8 num.)	Torgo
Puma8NH	8192	8 (8 num.)	Torgo
Puma32H	8192	32 (32 num.)	Torgo
Bank8FM	8192	8 (8 num.)	Torgo
Bank32NH	8192	32 (32 num.)	Torgo

of the cases. Model complexity and timings are shown in Table 5.9. In that table, sizes are emphasized if one method results in a model that is at least one third shorter than the other method's model.

The results can be summarized as follows (see also Table 5.10). Overall, the predictive performance of MAUVE is better than, or equally good as M5'. In the one case where M5' performs clearly better (*Auto-Price*), the model output by MAUVE is shorter. Concerning model complexity, in almost all cases, the tree found by MAUVE is smaller than, or equally large as the tree output by M5'. The only exception is *Auto-Mpg*, for which MAUVE outputs a slightly larger tree than M5'. In Table 5.10 we see that, taking into account both accuracy and number of leaves, MAUVE has seven clear winners, against one for M5'.

If we look at the number of parameters, then the results are not that clear (not visible in Table 5.10). That MAUVE is not longer outperforming M5' on this measure is probably due to the fact that it considers all predictor variables in the leaf models.

We can conclude that, from the point of view of minimizing the number of subregions in the input space where the same linear relation holds, the use of a simple regression based heuristic is in many cases useful for real world data. As can be seen from the induction times in Table 5.9, this heuristic comes at no additional computational cost.

In the following sections we examine whether this conclusion still holds in a different setting, e.g. when the pruning parameter is tuned, or when the smoothing operator is used.

Tuning the pruning factor. As discussed in Sect. 5.3.3.2 the two error estimates to be compared in the pruning phase are multiplied with a compensation

Table 5.8: Accuracy for real world data sets.

	Mauve		M5'	
Data set	RRMSE (StdDev)	Wins	RRMSE (StdDev)	Wins
Housing	47.98% (8.78)	8	46.56% (3.73)	12
Machines	**32.06%** (3.03)	19	37.34% (4.85)	1
Auto-Mpg	36.73% (0.87)	8	36.91% (0.95)	12
Auto-Price	56.08% (7.72)	1	**42.16%** (3.62)	19
Abalone	68.17% (2.10)	9	67.25% (0.51)	11
Servo	40.14% (3.31)	17	42.81% (3.03)	3
Wisconsin	103.69% (4.47)	12	104.42% (2.77)	8
Kin8NM	**60.02%** (1.19)	20	66.47% (0.66)	0
Puma8NH	**56.94%** (0.09)	20	57.88% (0.15)	0
Puma32H	28.31% (0.21)	11	28.28% (0.19)	9
Bank8FM	**20.21%** (0.09)	19	20.42% (0.08)	1
Bank32NH	**67.62%** (0.26)	19	69.34% (0.88)	1

Table 5.9: Size and induction times for real world data sets.

	Size				CPU times	
	Mauve		M5'		Mauve	M5'
Data set	Leaves	Pars	Leaves	Pars	Secs	Secs
Housing	**6**	46	13	52	0.93	1.14
Machines	**2**	38	3	**15**	0.46	0.46
Auto-Mpg	3	32	**2**	**17**	0.75	0.69
Auto-Price	**1**	**11**	10	29	0.30	0.35
Abalone	12	87	13	79	4.92	4.97
Servo	5	37	5	**21**	0.24	0.27
Wisconsin	3	29	4	**14**	0.49	0.47
Kin8NM	**62**	473	173	688	94.10	89.41
Puma8NH	**17**	**74**	46	153	93.58	104.45
Puma32H	**26**	**238**	169	618	141.68	132.79
Bank8FM	23	170	24	145	110.76	86.18
Bank32NH	3	56	3	54	116.16	150.19

Table 5.10: Summary of the comparison between Mauve and M5' on real world datasets. One method is said to be winning in accuracy if its RRMSE is smaller in 90% of the cases. One method is said to be winning in size if it results in a tree with one third less leaves.

		Accuracy		
		Mauve wins	Equality	M5' wins
Size	Mauve wins	3	2	1
	Equality	2	3	0
	M5' wins	0	1	0

Table 5.11: Accuracy for real world data sets with optimal pruning factor.

	Mauve		M5'	
Data set	RRMSE (StdDev)	Wins	RRMSE (StdDev)	Wins
Housing	48.42% (3.98)	6	46.38% (4.34)	14
Machines	**32.15%** (2.96)	19	37.25% (4.83)	1
Auto-Mpg	37.23% (0.84)	5	36.63% (0.74)	15
Auto-Price	53.79% (5.75)	1	**42.43%** (3.35)	19
Abalone	67.32% (1.15)	14	67.43% (0.38)	6
Servo	39.92% (3.40)	14	42.50% (2.54)	6
Wisconsin	106.17% (5.02)	6	101.80% (3.54)	14
Kin8NM	**60.16%** (0.70)	20	66.35% (0.60)	0
Puma8NH	**56.89%** (0.07)	20	57.52% (0.15)	0
Puma32H	28.27% (0.18)	0	**27.46%** (0.13)	20
Bank8FM	**20.26%** (0.09)	18	20.43% (0.08)	2
Bank32NH	**67.54%** (0.17)	19	68.02% (0.26)	1

factor that takes into account the number of parameters. In the calculation of the compensation factor a multiplier is used that was called the pruning factor. This parameter controls the size of the tree. The previous experiments were performed using M5's default value of 2 for this parameter. Since in Mauve the leaves contain linear models that may include all predictor variables (while M5' only includes the variables occurring in the pruned subtree), another value might be more appropriate, which might influence our experimental results. Therefore, the experiments were rerun, both for M5' and for Mauve, while estimating an optimal value for the pruning factor using cross validation. We proceeded as follows. For each of the training sets in the tenfold cross validation, we performed an inner threefold cross validation to estimate the optimal value for that training set. This optimal value was then used to test the model on the remaining test data. This procedure was again performed 20 times. The values for the pruning factor that were tested are 1, 2, 3, 4, and 5.

Table 5.11 shows error estimates. Model complexity is shown in Table 5.12. The number of leaves (parameters) is now the mean value of the number of leaves (parameters) of the model for each fold in the 20 tenfold cross validations. The tables show that, for both methods, the RRMSE value is not that much affected, but its variance tends to decrease. The size of the trees and the number of parameters also tend to decrease.

From Tables 5.11, 5.12, and 5.13 we learn that the relative performance of both systems is not much affected by the changed experimental setting. The only changes concern the *Puma32H* dataset, for which M5' becomes superior in accuracy, and the *Auto-Mpg* data, for which the size of both models becomes equal.

Table 5.12: Model size for real world data sets with optimal pruning factor.

	Mauve		M5’	
Data set	Leaves	Pars	Leaves	Pars
Housing	**4.86**	40.51	10.81	43.56
Machines	**2.66**	35.79	4.50	**17.33**
Auto-Mpg	2.56	26.05	2.38	18.30
Auto-Price	**2.14**	19.60	5.46	17.65
Abalone	8.43	57.92	6.19	39.15
Servo	4.03	28.47	4.28	19.59
Wisconsin	**1.71**	23.21	3.38	**12.66**
Kin8NM	**54.09**	384.80	103.54	442.22
Puma8NH	**14.31**	**61.39**	26.02	102.88
Puma32H	**17.57**	**142.55**	57.20	296.13
Bank8FM	23.58	173.26	27.19	157.13
Bank32NH	3.53	59.84	3.72	49.91

Table 5.13: Summary of the comparison between Mauve and M5’ on real world datasets with optimal pruning factor. One method is said to be winning in accuracy if its RRMSE is smaller in 90% of the cases. One method is said to be winning in size if it results in a tree with one third less leaves.

		Accuracy		
		Mauve wins	Equality	M5’ wins
Size	Mauve wins	3	2	2
	Equality	2	3	0
	M5’ wins	0	0	0

Table 5.14: Accuracy for real world data sets when smoothing is used.

	Mauve		M5'	
Data set	RRMSE (StdDev)	Wins	RRMSE (StdDev)	Wins
Housing	43.55% (6.72)	10	44.13% (2.66)	10
Machines	**30.86%** (3.02)	19	33.74% (4.05)	1
Auto-Mpg	36.08% (0.63)	10	36.25% (0.51)	10
Auto-Price	48.97% (3.70)	1	**41.07%** (2.16)	19
Abalone	66.44% (0.62)	14	66.74% (0.41)	6
Servo	39.37% (2.47)	11	40.30% (1.83)	9
Wisconsin	100.91% (3.22)	4	97.86% (1.57)	16
Kin8NM	**58.87%** (0.84)	20	63.68% (0.49)	0
Puma8NH	**56.81%** (0.06)	20	57.26% (0.11)	0
Puma32H	28.17% (0.17)	0	**27.07%** (0.09)	20
Bank8FM	**20.10%** (0.08)	18	20.25% (0.05)	2
Bank32NH	**67.52%** (0.19)	20	68.33% (0.33)	0

Table 5.15: Summary of the comparison between Mauve and M5' on real world datasets, when the smoothing operator is used. One method is said to be winning in accuracy if its RRMSE is smaller in 90% of the cases. One method is said to be winning in size if it results in a tree with one third less leaves.

		Accuracy		
		Mauve wins	Equality	M5' wins
Size	Mauve wins	3	1	2
	Equality	2	3	0
	M5' wins	0	1	0

Using the smoothing operator. A last experiment investigates the validity of our results in a setting where smoothing is used. At the same time, we examine whether smoothing improves the predictive performance of Mauve, as it does for standard M5' (Wang and Witten 1997). The pruning factor was reset to its default value in this experiment. Since smoothing does not change the model, only the way it is interpreted during prediction, it has no effect on the size of the tree. Therefore, we only report predictive performances (Table 5.14). For both systems, we see high improvements in RRMSE, up to an improvement of 7.11% for the *Auto-Price* dataset for Mauve.

Concerning the relative performance of both systems, we see no important changes in the summary table (Table 5.15). The only difference w.r.t. not using the smoothing operator again concerns the *Puma32H* dataset, for which M5' now wins in accuracy.

5.3.4.3 An extended comparison

In this last experiments section, we want to investigate where Mauve is positioned with respect to other model tree builders. For each of the four options discussed in section 5.3.2 we choose a representative system:

1. M5'
2. Mauve
3. Treed Regression (Alexander and Grimshaw 1996)
4. Retis (Karalic 1992)

In order to be able to compare the results (including induction times) in an objective manner, we also implemented these last two systems in Weka. In fact, we used Mauve's implementation and only adapted its quality function (the quality functions of Treed Regression and Retis are shown in Figures 5.3 and 5.2, respectively). This allows for a better comparison with the other methods. Consequently, there is a difference between Treed Regression and our implementation: in the latter, multiple linear models are used in the leaves, while Treed Regression uses simple linear models). We denote our Weka-implementation of Treed Regression and Retis by TR' and Retis' respectively. As in Mauve (and in M5') the calculations of the residual standard deviations are performed incrementally in TR', in order to evaluate the set of candidate splits in an efficient way. In the implementation of Retis', we used the incremental formulae proposed by Torgo (2002).

For completeness, we also included a standard regression tree inducer in our comparison. For this purpose we used M5' with the option to induce a normal regression tree instead of a model tree. This method is called RegTree in the comparison. Remark that RegTree is also an option 1 system, since it employs the same heuristic function as M5'.

For these experiments, both the pruning factor tuning and the smoothing operator were used. This gives a realistic experimental setup (optimized for high accuracy, which is what the user would normally do). The experiments were conducted on the same real world data sets of the previous section. In the tables we highlight the best results for each data set, without adding any statistical significance. Again, each experiment was performed 20 times and the results are averaged.

Table 5.16 shows the RRMSE for the different systems. We see that, except for the *Wisconsin* data set, RegTree always performs worst. Hence, the benefit of using model trees instead of standard regression trees is clearly shown. For the model tree systems, however, differences are smaller and, although Mauve and Retis' have the most winners, no clear conclusions can be drawn.

Tables 5.17 and 5.18 show the average number of leaves and parameters, respectively. We see in Table 5.17 that RegTree and M5' build overly large

Table 5.16: Accuracy for the different systems.

	RegTree	M5'	Mauve	TR'	Retis'
Data set	RRMSE (StD)	RRMSE (StD)	RRMSE (StD)	RRMSE (StD)	RRMSE (StD)
Housing	52.22% (0.93)	44.02% (2.80)	44.15% (1.80)	**38.97%** (1.41)	40.10% (1.95)
Machines	71.80% (1.55)	33.65% (4.09)	**30.92%** (2.97)	40.15% (3.84)	36.92% (4.73)
Auto-Mpg	46.63% (0.75)	36.34% (0.51)	35.99% (0.74)	**35.41%** (0.99)	37.95% (1.73)
Auto-Price	56.90% (1.26)	**40.80%** (2.35)	49.11% (2.77)	46.84% (4.92)	53.05% (8.71)
Abalone	69.77% (0.32)	67.06% (0.50)	**66.36%** (0.57)	67.04% (0.21)	66.80% (0.62)
Servo	59.57% (1.18)	39.93% (1.53)	**38.82%** (2.37)	38.89% (2.43)	39.27% (2.13)
Wisconsin	85.69% (1.57)	**83.42%** (2.05)	89.05% (4.49)	87.51% (4.36)	90.29% (2.43)
Kin8NM	69.91% (0.31)	62.92% (0.44)	57.76% (0.57)	53.47% (0.33)	**49.23%** (0.55)
Puma8NH	58.63% (0.13)	57.29% (0.11)	**56.82%** (0.06)	56.88% (0.06)	56.93% (0.07)
Puma32H	28.48% (0.10)	27.09% (0.09)	28.19% (0.16)	24.88% (0.07)	**22.09%** (0.08)
Bank8FM	25.64% (0.09)	20.15% (0.07)	20.04% (0.07)	20.62% (0.15)	**19.64%** (0.06)
Bank32NH	75.10% (0.24)	67.93% (0.17)	67.49% (0.15)	67.47% (0.16)	**67.41%** (0.29)

trees, as discussed in previous sections. The other systems are better matched. The differences in model size are reduced as we look at the number of parameters (Table 5.18). Note that for the standard regression trees, if the number of leaves is n, then the number of parameters is $2n - 1$.

If we compare Tables 5.17 and 5.18 to Table 5.12, we see large discrepancies in model size. The only difference in experimental setup is that smoothing is used in this last experiment. This suggests that the smoothing operator favours large trees. Remark that smoothing itself does not alter the tree size. When tuning the pruning factor, however, smoothed trees are compared in each fold of the tenfold cross validation. For each fold, the smoothed tree with the lowest error (determined by an inner threefold cross validation) is chosen. Since smoothing can be seen as somewhat reducing the effect of a split, larger trees may be chosen.

Induction times for building a single tree are shown in Table 5.19. A first observation is that the induction times of Retis' may be too high for many practical applications. For instance, for the *Puma32H* dataset, building one tree took us over six hours (on an Intel Pentium4 system with a 2.0GHz CPU). If one wants to tune the pruning factor using our method, in total 160 trees are built (although most of them on smaller parts of the data). A second observation is that, although we would expect RegTree to be the fastest (since it does not build regression models), in three cases Mauve is faster. This fact, together with the sizes of the pruned trees in Table 5.17 suggests that Mauve's tree building procedure in general stops earlier. A last observation concerns the *Bank8FM* dataset, where TR' has a shorter induction time than Mauve, although its pruned tree is twice as large. Given the fact that TR's tree building procedure is more complex, we conclude that Mauve's pruning procedure for this dataset takes much more time. Because of these two last observations we also measured the CPU time needed for the grow_tree function for each system, i.e. the time needed to grow the initial tree, before pruning takes place

Table 5.17: Number of leaves for the different systems.

	RegTree	M5'	Mauve	TR'	Retis'
	Leaves	Leaves	Leaves	Leaves	Leaves
Housing	36.29	18.77	7.39	10.44	**6.18**
Machines	15.08	5.02	**2.66**	2.85	3.49
Auto-Mpg	26.74	3.32	4.15	6.17	**3.31**
Auto-Price	10.59	7.50	**2.34**	2.95	2.99
Abalone	105.87	20.48	12.84	**5.29**	5.65
Servo	10.34	4.66	4.39	**4.15**	4.52
Wisconsin	16.33	11.06	2.88	2.60	**1.48**
Kin8NM	576.03	400.80	**135.60**	272.49	201.59
Puma8NH	205.95	38.83	16.12	19.09	**14.16**
Puma32H	759.25	147.83	**18.81**	38.83	36.67
Bank8FM	451.78	64.71	**31.71**	64.16	33.38
Bank32NH	267.33	3.82	**3.66**	3.69	4.21

Table 5.18: Number of parameters for the different systems.

	RegTree	M5'	Mauve	TR'	Retis'
	Pars	Pars	Pars	Pars	Pars
Housing	71.58	63.70	**56.48**	73.70	56.49
Machines	29.17	**18.74**	35.83	33.34	37.80
Auto-Mpg	52.49	**20.87**	36.95	46.94	36.13
Auto-Price	**20.18**	22.18	21.52	28.73	34.30
Abalone	210.74	83.38	80.08	**34.72**	41.24
Servo	**19.68**	20.53	30.85	28.40	29.90
Wisconsin	31.67	30.31	38.58	34.83	**27.07**
Kin8NM	1151.07	1184.45	**843.68**	1464.77	1523.98
Puma8NH	410.91	135.31	68.56	77.51	**63.63**
Puma32H	1517.51	531.18	**153.31**	313.21	380.91
Bank8FM	902.56	284.49	**225.16**	375.70	250.79
Bank32NH	533.67	**50.54**	61.80	60.09	73.63

Table 5.19: Total induction time used for the different systems.

	RegTree	M5'	Mauve	TR'	Retis'
	Secs	Secs	Secs	Secs	Secs
Housing	0.91	1.05	**0.86**	1.39	19.47
Machines	**0.35**	0.37	0.38	0.92	36.40
Auto-Mpg	**0.57**	0.61	0.67	1.12	39.59
Auto-Price	**0.25**	0.31	0.29	0.35	4.87
Abalone	**3.84**	4.88	4.78	7.35	605.04
Servo	0.20	0.23	**0.16**	0.27	3.23
Wisconsin	**0.35**	0.40	0.42	0.70	34.53
Kin8NM	**84.38**	86.90	89.13	91.51	1976.87
Puma8NH	95.75	101.14	**91.57**	102.06	2641.36
Puma32H	**88.63**	130.16	139.43	224.21	24175.54
Bank8FM	83.61	84.34	107.62	**81.34**	2835.64
Bank32NH	**89.52**	147.55	114.69	177.73	23561.74

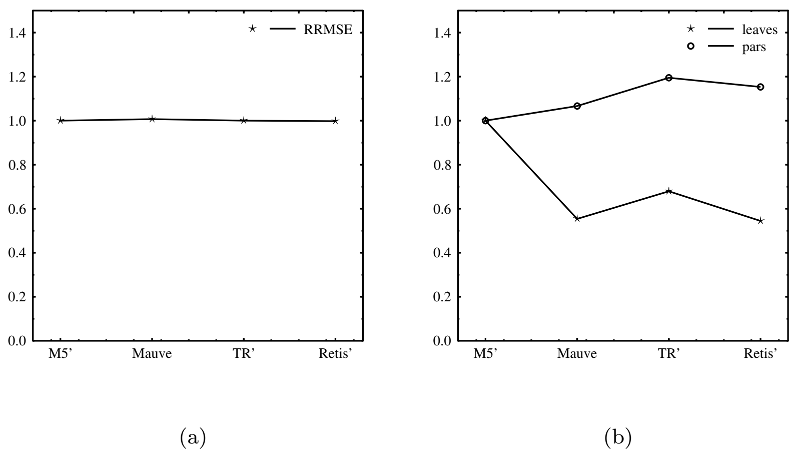
and before any regression model is built in the leaves. Table 5.20 shows the results. There we see that Mauve always has the shortest CPU time (except for one case). Since the time needed for pruning is relatively stable compared to the time needed for growing the tree, it follows that the relative time spent in the pruning phase is much smaller for Retis' (3.4% on average) than for, e.g., Mauve (92.4% on average).

This extensive comparison of different model tree systems is summarized in Fig. 5.5. We left out the results for RegTree and take M5' as a base reference. The results of the other model tree inducers are plotted against this base reference. We conclude that going from option 1 (represented by M5') to option 2 (represented by Mauve) leads to a large drop in the number of leaves while costing nothing w.r.t. accuracy or induction time. Going to more sophisticated systems in general does not add any noticeable improvements in accuracy or model size, while giving up on efficiency.

5.4 Upgrading Mauve to relational learning

In the previous section we have proposed a propositional model tree learner, called Mauve, that uses a heuristic that overcomes the problems of variance based methods without giving up on efficiency. In this section we discuss how Mauve is upgraded to a relational model tree learner. Again, we start from the system Tilde, and make changes to the algorithm where needed. We call the resulting algorithm ReMauve (RElational Mauve).

The system Tilde was introduced in Sect. 4.2 and its procedure to grow a tree is given in Table 4.1. Remember that Tilde learns both classification and



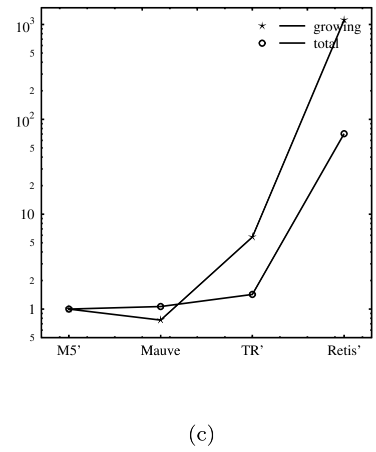


Figure 5.5: Summary of the comparison of different model tree inducers. (a) RRMSE (b) Model Size (c) Induction times (logarithmic scale).

Table 5.20: Induction time for growing the tree for the different systems.

	RegTree	M5'	Mauve	TR'	Retis'
	Secs	Secs	Secs	Secs	Secs
Housing	0.10	0.10	**0.06**	0.53	18.56
Machines	0.09	0.09	**0.05**	0.53	36.01
Auto-Mpg	0.10	0.10	**0.07**	0.66	39.03
Auto-Price	0.05	0.05	**0.03**	0.15	4.67
Abalone	0.70	0.72	**0.69**	3.75	600.42
Servo	0.03	0.03	**0.02**	0.12	3.05
Wisconsin	0.08	0.08	**0.04**	0.36	32.07
Kin8NM	1.53	1.54	**1.47**	7.08	1866.99
Puma8NH	1.69	1.69	**1.49**	7.78	2486.98
Puma32H	6.61	6.63	**6.00**	80.09	24021.03
Bank8FM	**1.51**	1.53	1.83	6.80	2742.07
Bank32NH	8.24	8.26	**5.50**	71.60	23418.22

regression trees. The regression tree subsystem is usually denoted by Tilde-RT.

We briefly repeat the main steps in Tilde's procedure to grow a tree. It takes as input the training examples E and a query Q that corresponds to the empty query. In the recursive calls of the algorithm, Q will represent the conjunction of all succeeding tests from the root of the tree to the node being split. This query was referred to as the *current query* in Chapt. 4. The procedure to grow a node T is as follows. First, a refinement operator generates the set of candidate splits. This set is determined by the language bias given by the user, and by the variables occurring in the current query at T. The refinement operator typically operates under θ-subsumption (Plotkin 1969) and generates candidates by extending the current query with a number of new literals. Next, the optimal_split procedure executes all candidates on the set of examples E, estimating the quality of each candidate, and returns the best candidate Q_b. The candidate Q_b is chosen to split the examples. The conjunction put in the node T consists of $Q_b - Q$, i.e., the literals that have been added to Q in order to produce Q_b. In the left branch, Q_b will be further refined, while in the right branch Q is to be refined. When the stop criterion holds (typically, this is when a predefined minimum number of examples is reached), a leaf is built. The predict procedure returns the mean target value of the examples E.

In Tilde-RT the quality of a candidate is calculated using a simple variance based heuristic function. More precisely, the sum of squared errors in both child nodes is minimized, and an F-test is used to decide whether an improvement is obtained w.r.t. the parent node.

In the following sections, we discuss several aspects of the algorithm that were changed in order to upgrade Tilde to a model tree algorithm.

5.4.1 Adapting Tilde-RT's heuristic function.

We changed TILDE-RT's heuristic function in order to use residual standard deviation (*RSD*) instead of sum of squared errors. For nominal splits the normal standard deviation is used. In other words, we have implemented the same heuristic function as MAUVE's.

For numeric splits, an important issue to deal with concerns the multi-valuedness of the numeric split attribute to be introduced in the regression functions for the *RSD* calculations. The formula for residual standard deviation (*residual* w.r.t. a simple linear regression line) is as follows:

$$RSD(Y,X) = \sqrt{\frac{\sum_i (y_i - (\alpha x_i + \beta))^2}{n-2}},$$

and can be computed entirely from the statistics $\sum_i(1, x_i, x_i^2, y_i, y_i^2, x_i y_i)$, where Y is the predicted attribute and X is the predictor. If X is non-determinate, different values can be filled in, each one resulting in an other *RSD* value.

In Chapt. 3, we have seen that, in general, two approaches exist to deal with multi-valuedness. ILP systems usually test for the existence of a specific element, thus, a split condition $account(PersId, AccId, Type, Bal), Bal > 1000$ corresponds to testing the existence of an account with balance larger than 1000. Other approaches (Koller 1999; Neville et al. 2003; Krogel and Wrobel 2003) use aggregate functions (such as *max*, *min*, *avg*, *sum*,...) to summarize the set of values. The following lemma shows that ILP tests are semantically equivalent to an aggregate function (Knobbe and Ho 2005):

Lemma 5.1 *Let B be a bag of real numbers, and t some real value, then*
$\exists v \in B : v \geq t$ *iff* $max(B) \geq t$, *and*
$\exists v \in B : v \leq t$ *iff* $min(B) \leq t$.

Using this lemma, every numeric attribute results for each example in one deterministic value to feed to the *RSD* formula. The previous non-determinate numeric attribute would become $max(Bal, account(PersId, AccId, Tp, Bal), M)$.

5.4.2 Adapting Tilde-RT's predictive function.

In MAUVE the leaves contain a multiple linear regression function using *all* numeric attributes as predictors. Adopting this strategy in REMAUVE is not feasible: in relational learning the number of numeric attributes becomes very high, especially when complex aggregate conditions are taken into account. Therefore, we include a numeric attribute in the predictive model of a leaf only if it was chosen at a node on the path[4] from the root to the leaf. The underlying idea is that an attribute would not have been chosen to split the dataset if it did not result in a linear relation with the target in the child nodes.

[4]Note that we use the numeric attributes on the complete path from the root to the leaf, not only those from the current query which correspond only to the succeeding tests.

5.4.3 Dealing with global effects.

Consider an attribute that has a global linear effect on the target. Sooner or later in the tree building process this attribute will give rise to a best split, with the same linear effect in both child nodes and will thus generate a superfluous split in the model tree. While the split is redundant, we do want to take into account this attribute in the predictive models at the leaves. Therefore, when the best test for a node N is determined and is found to be a numeric split, the RSD is also calculated for all examples at node N. If this RSD is equal to the heuristic value of the split, then we know that the linear effect between the split attribute and the target holds in the complete set of examples at N, thus it should be introduced in the predictive models in the leaves under N without splitting the data at N. To deal with such global linear effects, we introduce unary regression nodes that do not split the data, but only serve to introduce an extra predictor in the linear regression function. The regression nodes contain numeric attributes (without the "$>$" or "$<$" equation) and pass all examples down to their unique child node. As for split nodes, variables occurring in the attribute of a regression node can be used further down the tree.

In relational learning, especially when aggregates are used, correlation between attributes often comes into play, either true or apparent (Jensen et al. 2003). For example, in the task of predicting a person's income, the income may increase with the number of children. However, the number of children is correlated with the number of daughters or with the sum of the ages of the children. In our system, if the number of children is an attribute occurring in a regression node, the probability of having an other regression node with the number of daughters is high. To avoid this, the linear effect of numeric attributes occurring in the tree needs to be accounted for. Therefore, after introducing a regression node or a numeric split node, we remove the linear effect of the involved attribute A from the target, i.e., we pass on the residuals $y_i - \hat{y}_i$ with $\hat{y}_i = \alpha * A + \beta$ to the child node(s). In fact, the linear effect should also be removed from all other numeric attributes that can still be used in the model. Given the large number of such attributes, this is not feasible, and instead, when building a regression node N we check whether the involved numeric attribute A has a significant correlation with an attribute in a split or regression node on the path from the root to N. If this is the case a leaf is built.

By introducing regression nodes, the analogy with Mr-Smoti increases. A comparison between the two systems is given later in this section.

5.4.4 Stop criterion.

We implemented several stop criteria. The first one concerns the minimal number of examples a leaf has to cover. Building a linear model in k attributes in the leaves requires at least $k + 1$ examples.

Therefore, after refining a node T, we check whether each child node of T contains at least $m+1$ examples, where m is the number of numeric attributes occurring on the path from the root to T. If this is not the case, T is made a leaf. The second stop criterion calculates the SD of the target values, before they are updated to reflect the linear effect of the best test. If this falls below a certain percentage (default 5%) of the original SD at the root node, a leaf is constructed.

As a last stop criterion, if the best test turns out to be nominal, its SD value is compared to the SD value of the parent node. If they are the same, a leaf node is built. As stated before, for numeric tests, a regression node is built in that case.

The pseudo code of the most important procedures of the algorithm is presented in Table 5.21. Pruning and smoothing are not supported in the current version of ReMauve.

5.4.5 Undefined attributes.

An issue that has not been mentioned in the description of the algorithm is what happens if an attribute is undefined for an example. This problem and several possible solutions for it were presented in Sect. 4.5. We argued that for our system the best option was to fail whenever aggregating over empty sets. In ReMauve the same approach is taken, in the sense that examples for which a split condition is undefined go to the right (failing) branch of the tree. However, this is not sufficient: the heuristic function needs to have a numeric value for each example in the node to be split (also for those going to the right branch) and the linear equations in the leaves need to be able to provide a prediction for each example. Therefore, whenever an explicit value for an undefined attribute is needed (i.e., to calculate the heuristics or to make predictions in the leaves), we make use of a default value. There are several possibilities for choosing a default value. We decided to use a value that reduces as much as possible the influence of examples for which the attribute is undefined. This corresponds to option 2(a) in Sect. 4.5. The exact values are:

- $max(\emptyset) = avg(max(S_1), max(S_2), ..., max(S_n))$
- $min(\emptyset) = avg(min(S_1), min(S_2), ..., min(S_n))$
- $avg(\emptyset) = avg(avg(S_1), avg(S_2), ..., avg(S_n))$
- $sum(\emptyset) = avg(sum(S_1), sum(S_2), ..., sum(S_n))$
- $mode(\emptyset) = mode(mode(S_1), mode(S_2), ..., mode(S_n))$

where S_i is the set of values observed for the attribute for the i-th example and n is the number of training examples at the node under consideration for which

Table 5.21: ReMauve algorithm for first order logical model tree induction.

```
procedure GROW_TREE ($E$: examples, $T$: targets, $Q$: query, $P$: path):
    $candidates := \rho(Q)$
    $Q_b :=$ OPTIMAL_REFINEMENT$(candidates, E, T)$
    $conj := Q_b - Q$
    $P_{new} := P + conj$
    if STOP_CRIT $(conj, P_{new}, E)$
    then
        $K :=$ PREDICT$(E, P)$
        return leaf$(K)$
    else
        if SPLIT_COND $(conj)$
        then
            $E_l := \{e \in E | Q_b$ succeeds in $e \wedge Background\}$
            $E_r := \{e \in E | Q_b$ fails in $e \wedge Background\}$
            $T_l :=$REMOVE_LINEAR_EFFECT $(E_l, T, conj)$
            $T_r :=$REMOVE_LINEAR_EFFECT $(E_r, T, conj)$
            $left :=$ GROW_TREE $(E_l, T_l, Q_b, P_{new})$
            $right :=$ GROW_TREE $(E_r, T_r, Q, P_{new})$
            return split node$(conj, left, right)$
        else
            $T_{ch} :=$REMOVE_LINEAR_EFFECT $(E, T, conj)$
            $child :=$ GROW_TREE $(E, T_{ch}, Q_b, P_{new})$
            return regression node$(conj, child)$

procedure OPTIMAL_REFINEMENT ($Q$: queries, $E$: examples, $T$: targets):
    for all $Q_i \in Qs$
        EXECUTE$(Q_i, E)$
        if (NOMINAL $(Q)$)
        then $Heur(Q_i) := \frac{|E_l|}{|E|} SD(E_l) + \frac{|E_r|}{|E|} SD(E_r)$
        else $Heur(Q_i) := \frac{|E_l|}{|E|} RSD(E_l, Q) + \frac{|E_r|}{|E|} RSD(E_r, Q)$
    $Q_b := arg\ min_{Q_i} Heur(Q_i)$
    if (NOMINAL $(Q_b)$)
    then return $Q_b$
    else
        $Heur_p(Q_b) := RSD(E, Q_b)$
        if $(Heur_p(Q_b) \leq Heur(Q_b))$
        then return EXTRACT_NUMERIC_ATTR$(Q_b)$
        else return $Q_b$
```

the attribute is defined. Note that this may not be the best solution. An other possibility would be to use the normal standard deviation for numeric splits where the split attribute is undefined for some examples.

5.4.6 Comparison with Mr-Smoti.

By introducing regression nodes into our system, the resemblance with Mr-Smoti increases. In the remainder of this section, we discuss the most important differences between both systems.

Complexity of finding the best split node. In ReMauve the evaluation of a numeric split requires the calculation of two simple linear regression functions: one for each child node. In Mr-Smoti a similar, but more complex heuristic function is used: in each child node simple linear regression models are constructed, each with a different numeric attribute used as the predictive attribute. The best regression is chosen independently for the two children and the heuristic value associated with the split under consideration is the weighted average of the RSD of the best regression lines of left and right child. Finding the best numeric split amongst all predictors therefore has complexity $O(m)$ for ReMauve and $O(m^2)$ for Mr-Smoti, with m the number of numeric predictors.

Complexity of introducing regression nodes. In our system, introducing a regression node requires almost no computation: after the best split condition is obtained and is found to be numeric, the global linear effect of the attribute in the split is tested. This requires only one extra RSD to be computed. In Mr-Smoti the best regression node is searched for independently of the best split node and requires a lookahead step, in the sense that the best split is searched after the new attribute is included in the multiple model. This renders the whole node selection procedure for Mr-Smoti cubic in the number of predictors.

Removing the linear effect of attributes. In Mr-Smoti regression nodes were introduced in order to incrementally build the multiple regression models in the leaves of the model tree. To achieve that, next to updating the target values, the linear effect of an introduced numeric attribute also has to be removed from all other numeric predictors that may be used later in the tree. In ReMauve it is not possible to update all numeric attributes in the dataset, because these attributes are generated on-the-fly at each node. It would not be feasible to do this updating during refinement generation (requiring another RSD calculation for each refinement and each numeric attribute on the path from the root to the node) given the huge search spaces that may be dealt with by introducing complex aggregates. Therefore, in ReMauve, the final multiple regression model in the leaves is built from scratch.

Overall complexity. The observations above lead to the following overall complexity results. For MR-SMOTI, the inner node refinement procedure has complexity $O(m^3)$, with m the number of numeric attributes. In a leaf, however, the predictive regression model is obtained by composing the models on the path from the root to the leaf, and thus, can be performed in constant time. A model tree with k inner nodes contains at most $k+1$ leaves, thus the overall complexity for building a model tree with MR-SMOTI is $k \times O(m^3) + (k+1) \times O(1)$.

For REMAUVE the node refinement process has complexity $O(m)$. Constructing a leaf requires $O(p^3)$, where p is the number of numeric attributes on the path from the root to the leaf. This results in an overall complexity of $k \times O(m) + (k+1) \times O(p^3)$. Given the fact that $p << m$, especially when using complex aggregates, the REMAUVE system is more efficient for the applications we target.

Representational formalism. A last important difference between both systems concerns their representational formalism. Whereas REMAUVE is an ILP system, MR-SMOTI operates on a relational database, using selection graphs (Knobbe et al. 1999) to represent nodes of the model tree.

5.5 Experiments

In this experiments section, we address two questions:

1. How do model trees that predict functions with complex aggregates perform compared to model trees that do not predict aggregates?

2. How does REMAUVE compare to other systems as TILDE-RT or MR-SMOTI?

The datasets used in our experiments are described in Sect. 5.5.1. The experimental setup is discussed in Sect. 5.5.2 and the obtained results are reported in Sect. 5.5.3.

5.5.1 Datasets

We have performed experiments on two biological datasets: *Mutagenesis* (Srinivasan et al. 1996) and *MassSpectrogram*[5]. Given the scarceness of publicly available relational regression datasets with numeric attributes, we also constructed two synthetic datasets.

[5]SDBS, National Institute of Advanced Industrial Science and Technology, Japan, http://www.aist.go.jp

5.5.1.1 Mutagenesis

The *Mutagenesis* dataset was already discussed in Ex. 2.3 and in Sect. 4.6.1.1. Recall that the task is to predict the mutagenicity of nitro-aromatic compounds. In these experiments we use the numeric target value, which corresponds to the logarithm of the mutagenicity level of the molecules. The dataset contains 230 compounds, of which 188 are known to be well predicted by linear regression methods. In our experiments we use both the regression friendly subset and the full dataset. Several descriptions of the compounds have been proposed (Srinivasan et al. 1995). In these experiments, we use the backgrounds B2 (atoms and bonds, including partial charge of atoms) and B3 (B2 extended with the ϵ_{Lumo} (energy of the compounds lowest unoccupied molecular orbital) and *LogP* (logarithm of the compound's octanol/water partition coefficient) properties).

5.5.1.2 MassSpectrogram

The task in the *MassSpectrogram* dataset is to predict the weight of a molecule based on its mass spectrogram. In mass spectroscopy, molecules of a compound are bombarded with electrons. Some break up to give a variety of charged fragments. A mass spectrogram is a graph of the mass-to-charge ratio of the different fragments versus the frequency. The dataset contains 873 molecules.

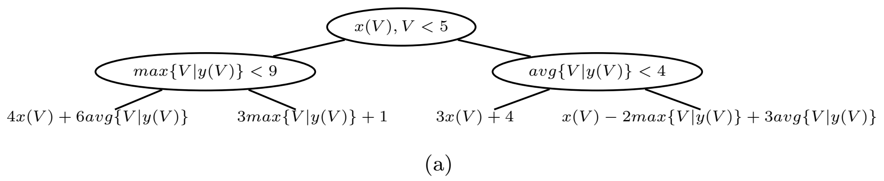
5.5.1.3 Synthetic datasets

For the synthetic datasets the true target function is a model tree that contains aggregates. They both contain 1000 examples. The first dataset (*Artificial1*) contains two predictive attributes: $x(X)$ (determinate) and $y(Y)$ (non-determinate). Each example contains 8 y literals, for which the value can be aggregated. All numeric values are random values, uniformly distributed between 0 and 10. The target function for this dataset is shown in Fig. 5.6(a). It requires two regression nodes in ReMauve.

The second dataset (*Artificial2*) includes three predictive attributes: $x(X)$, $y(C, Y)$, and $z(Z)$, of which y is non-determinate and has 15 values for each example. Again the numeric values for x, y, and z are uniformly distributed between 0 and 10. The C variable in the y literal is a boolean value. The target function is shown in Fig. 5.6(b). This dataset also requires two regression nodes, one of which involves a complex aggregate using the boolean condition. For the two datasets, we added Gaussian distributed noise to the target value.

5.5.2 Experimental setup

As explained before, ReMauve is able to learn complex aggregate conditions. In order to address the first question defined above and to allow for a comparison with Mr-Smoti, we performed the experiments with and without the ability



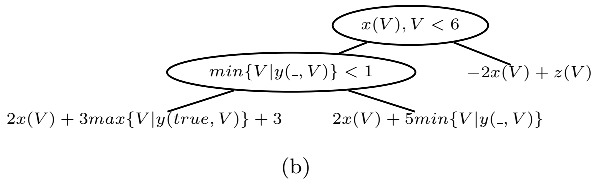

$x(V), V < 6$
$min\{V|y(\_, V)\} < 1$
$-2x(V) + z(V)$
$2x(V) + 3max\{V|y(true, V)\} + 3$
$2x(V) + 5min\{V|y(\_, V)\}$
(b)

Figure 5.6: Target function for two synthetic datasets. (a) The *Artificial1* dataset. (b) The *Artificial2* dataset.

to learn aggregates. ReMauve is compared to Tilde-RT and Mr-Smoti w.r.t. predictive performance and model complexity. Predictive performance is obtained by taking the average MSE (mean squared error) of five tenfold cross validations. Model size is measured as the number of leaves and the number of regression nodes (the latter only for ReMauve and Mr-Smoti). Induction times are difficult to compare, since Tilde-RT and Remauve were run on a different platform than Mr-Smoti.

5.5.3 Experimental results

The results are presented in Tables 5.22 and 5.23.

The first question is dealt with by comparing ReMauve's predictive performance when learning complex aggregates to when not learning them. For *MassSpectrogram* and the artificial datasets, a clear predictive performance improvement is obtained when complex aggregates are considered. Moreover, the improvement holds for both ReMauve and Tilde-RT. Part of the resulting tree for *MassSpectrogram* is shown in Fig. 5.7. For *Mutagenesis*, the result is less obvious. Both for ReMauve and Tilde-RT the error tends to increase when learning aggregates. Whereas in the classification setting complex aggregates turned out to be beneficial for this task, to our knowledge, complex aggregates have not been used before to predict the numeric mutagenicity level of molecules, thus we can not compare this result to other results in the literature.

The second question is answered by comparing ReMauve to Tilde-RT and Mr-Smoti w.r.t. predictive performance and model complexity. When com-

```
max{Ratio|ms(Mol, Ratio, Freq)} < 199.0 ?
+yes:  avg{Ratio|ms(Mol, Ratio, Freq), Ratio < 83.0} < 51.0 ?
|      +yes:  max{Ratio|ms(Mol, Ratio, Freq), Freq < 2.3} < 119.0 ?
|      |      +yes:  0.84 * max{Ratio|ms(Mol, Ratio, Freq)}+
|      |             0.85 * avg{Ratio|ms(Mol, Ratio, Freq), Ratio < 83.0}+
|      |             0.06 * max{Ratio|ms(Mol, Ratio, Freq), Freq < 2.3} − 19.08
       ...
+no:   avg{Freq|ms(Mol, Ratio, Freq)} < 8.0 ?
       +yes:  avg{Freq|ms(Mol, Ratio, Freq), Ratio < 60.0}
       |      +--:  0.95 * max{Ratio|ms(Mol, Ratio, Freq)}+
       |             −13.30 * avg{Freq|ms(Mol, Ratio, Freq)}+
       |             5.50 * avg{Freq|ms(Mol, Ratio, Freq), Ratio < 60.0} + 97.91
       ...
```

Figure 5.7: Resulting tree for the *MassSpectrogram* dataset.

paring ReMauve to Tilde-RT, we see that in the aggregate settings (i.e., in the context of many numeric attributes), an improvement in both predictive accuracy and model complexity is obtained. Also, for the artificial datasets, where the target concept involves linear regressions, a clear improvement is obtained, both with and without aggregates. In the other settings, while generally resulting in smaller models, the comparison in predictive performance is less clear. When comparing ReMauve to Mr-Smoti, a first observation is that ReMauve tends to build shorter trees. Only on the *Artificial2* dataset is the model built by Mr-Smoti simpler. Regarding predictive performance, we see clear winners for ReMauve on the artificial datasets. On the *MassSpectrogram* dataset, Mr-Smoti outperforms ReMauve. However, when learning complex aggregates, ReMauve reduces Mr-Smoti's MSE with a factor 3.6. On the *Mutagenesis* datasets, the results are divided: two winners for each system. (The high MSE of 32.68 for Mr-Smoti on the full dataset with background B2 is due to two particular test examples. Removing them from the test sets yields an average MSE of 4.79 (0.12).)

5.6 Conclusion

In the previous chapters, we have introduced the concept of complex aggregates. We have constructed hypotheses of the form "if A then B" where A is a conjunction of conditions that may contain complex aggregates and B is the predicted value. Some patterns, however, do not just predict a single value, but may involve predictive functions that contain predictive attributes. Examples are linear equations or model trees, which are regression trees that construct linear equations in their leaves. When these patterns are extrapolated to a relational setting, they are exposed to the same difficulty as explained in Chapt. 2: the predictive attributes may be non-determinate due to one-to-many and many-to-many relations and, thus, a technique is needed to summarize them.

In this chapter, we have developed a relational model tree learner that may

Table 5.22: Comparing ReMauve's predictive performance and tree size to Tilde-RT and Mr-Smoti for the *Mutagenesis* dataset.

	Mutagenesis					
	Regression friendly subset			Full dataset		
	B2 no agg.	B3 no agg.	B3 agg.	B2 no agg.	B3 no agg.	B3 agg.
Avg. MSE						
ReMauve	1.98 (0.1)	1.45 (0.5)	1.43 (0.4)	4.01 (0.2)	3.50 (0.6)	3.70 (0.6)
Tilde-RT	1.96 (0.1)	1.57 (0.1)	1.85 (0.2)	3.67 (0.2)	3.44 (0.2)	3.94 (0.4)
Mr-Smoti	3.02 (0.1)	1.14 (0.2)	-	32.68 (28.1)	3.32 (0.2)	-
Regr. nodes						
ReMauve	1	2	8	2	1	6
Mr-Smoti	8	5	-	8	15	-
Leaves						
ReMauve	7	3	5	11	8	5
Tilde-RT	14	16	28	11	23	28
Mr-Smoti	10	7	-	9	15	-

involve complex aggregate conditions in the linear equations at their leaves of the learned trees. Complex aggregates can be included in the model of a leaf if they occur on the path from the root of the tree to the leaf.

The main problem is to find a suitable heuristic function for the model trees. Next to taking into account linear models, an important requirement is efficiency, given the large feature spaces at the nodes of the trees. First, we have studied the behaviour of variance as a heuristic. While the inappropriateness of variance for this task, in itself, was pointed out in earlier work (Karalic 1992; Malerba et al. 2004), no investigation was performed on exactly how this influences the quality of the induced trees. Our results show that it mainly influences the explanatory power of the tree, rather than its predictive power. Since explanatory power is often mentioned as an important advantage of trees over black-box models, it is worthwhile to try to improve it. Therefore, we have presented a system, called Mauve, that does indeed induce model trees with better explanatory power, in the sense that shorter trees are built. This approach has a complexity that is linear in the number of attributes, and as such differs from the complexity of the most efficient approaches only with a constant factor. Currently existing alternative approaches towards building better trees (such as Retis or Smoti) have a complexity that is at least quadratic. Our experimental validation of Mauve confirms that the heuristic we propose yields simpler trees with equal predictive accuracy in cases where there is a piecewise linear relation between the target and the input variables. Moreover, the results suggest that this kind of behaviour often occurs in real world data and that using more sophisticated approaches on average does not yield improvements in accuracy or model size.

Table 5.23: Comparing ReMauve's predictive performance and tree size to Tilde-RT and Mr-Smoti for the *MassSpectrogram* and artificial datasets.

	MassSpectrogram		Artificial1		Artificial2	
	no agg.	agg.	no agg.	agg.	no agg.	agg.
Avg. MSE						
ReMauve	8144 (65)	1289 (101)	30.84 (0.2)	1.08 (0.0)	1.64 (0.1)	0.97 (0.0)
Tilde-RT	8132 (24)	2401 (146)	35.18 (0.3)	3.94 (0.1)	2.55 (0.0)	2.06 (0.1)
Mr-Smoti	4583 (221)	-	60.58 (2.0)	-	12.08 (0.7)	-
Regr. nodes						
ReMauve	1	9	1	2	2	3
Mr-Smoti	6	-	10	-	0	-
Leaves						
ReMauve	3	10	5	4	3	3
Tilde-RT	3	222	30	58	44	64
Mr-Smoti	8	-	14	-	3	-

Afterwards, we have upgraded Mauve to a relational model tree learner, correspondingly called ReMauve. The system was implemented in Tilde and, hence, making use of the results of Chapt. 4, it is able to learn complex aggregate conditions in the internal nodes of the tree. The linear equations at the leaves contain complex aggregates if these aggregates have shown a linear relation with the target during the tree building process.

Experimental results demonstrate that, if many numeric attributes occur in the dataset (e.g., in the context of learning aggregates), our system outperforms normal regression tree learners. When comparing to a model tree learner that uses a more complex heuristic function, the comparison in predictive performance is less obvious, while our system in general produces shorter trees.

Chapter 6

Applications

6.1 Introduction

In this chapter we present two applications where complex aggregate conditions are used in real life problems.

In the first application, we investigate an agricultural dataset about genetically modified crops. The possibility of genetically modified crops mixing with conventional or organic crops (e.g., by pollen being blown by wind) has become a delicate issue and the detection of modified crops in conventional fields presents a challenge. In Sect 6.2, we use TILDE with complex aggregates to investigate the influence of surrounding fields on the level of contamination with genetically modified variants in a particular target field.

The second application was introduced at the Inductive Logic Programming 2005 Challenge. The challenge consists of analysing a real world biological dataset containing the genes in the genome of baker's or brewer's yeast. The task is to predict the function of the genes. The complex aggregates used in this study are learned with a frequent pattern miner, which illustrates an other application of the aggregates. This application is presented in Sect. 6.3.

Section 6.4 concludes and gives a brief summary of the obtained results.

6.2 An agricultural application

In this first application we predict adventitious presence of genetically modified varieties in conventional oilseed rape crops. The problem is described in Sect. 6.2.1. The dataset and corresponding ILP representation are described in Sect. 6.2.2 and 6.2.3, respectively. Experiments are discussed in Sect. 6.2.4 and Sect. 6.2.5 concludes.

6.2.1 Problem description

Genetically modified (GM) crops were first planted world-wide to a significant extent in 1996 and since then the planted area has increased rapidly. The biggest share of area planted with GM crops (99%) is in USA, Canada, Argentina and China. The main purpose of engineering genetically modified crops is to create crops that are able to survive being sprayed with harmful chemicals like pesticides and herbicides.

However, for as long as GM crops have been planted, there has been controversy about it. Concerns about GM food include the following:

- commercial considerations: unintended presence of GM variants in conventional crop production affecting its competitiveness on the market place,
- ecological influences: creation of new weeds, pests may develop resistance to GM crops that have been designed to kill them,...
- health hazard: a lot of debate is going on about the possible unwanted influence of consuming GM crops on the human health, which has not been proven yet.

The main concern today is the co-existence issue, i.e., the possibility of GM crops mixing with conventional or organic crops. GM crops can contaminate other crops simply by pollen being blown by wind from one field to another. EU regulations allow 0.9% of adventitious presence of GM material in conventional yield and the co-existence is concerned with achieving the prescribed level of adventitious presence in regions with both conventional and organic crops. Therefore, there is a need to find appropriate cropping practices (sowing date, soil tillage,...) to minimize adventitious presence of GM crops.

To estimate levels of adventitious presence of GM varieties in non-GM varieties and to compare the effects of changing farming practices, computer models have been developed, such as GeneSys (Colbach et al. 2001). GeneSys's purpose is to rank cropping systems according to their probability of gene flow between genetically modified and non-genetically modified organisms, in this case oilseed rape (*Brassica napus*), which exists in a genetically modified herbicide-tolerant variant and a non-GM, conventional, variant. GeneSys predicts the level of harvest contamination of conventional oilseed rape crops by genetically modified rape seeds.

In this section we analyze a dataset produced by GeneSys. The dataset contains simulations of field patterns with information on the cropping practices for each field and for several years. The task considered is to predict the level of harvest contamination for one specific target field. The dataset was previously propositionalized and analyzed using propositional learning techniques (Ivanovska et al. 2006). However, intuitively, one would expect that the contamination of the target field depends a lot on the cropping techniques and crops grown on the surrounding fields (e.g., the level of contamination of a field may be influenced by the crop grown at or the level of contamination of its neighbouring fields). So it seems worthwhile to exploit neighbourhood relations in the predictive model and create a relational representation of the problem. Also, the probability of contamination might increase if the field plan contains a lot of contaminated fields in the neighbourhood of the target field. Therefore it would be useful to investigate properties at the regional level, which can be obtained by using complex aggregates. For this study we use the first order decision tree learner Tilde, extended with the capability of learning complex aggregates. More precisely, Tilde-μ was used.

6.2.2 Dataset

The dataset used in our study is the output from GeneSys simulations (Colbach et al. 2001). GeneSys was developed by INRA (French National Institute for Agricultural Research) to rank cropping systems according to their probability of gene flow from herbicide-tolerant oilseed rape to conventional oilseed rape both in time, via seeds, and in space, via pollen and seeds. GeneSys has the following input variables (Fig. 6.1):

- The field plan of the region, comprising cultivated fields as well as uncultivated field- and road-margins consisting of spontaneous vegetation (hence "borders"). Borders consist of strips of spontaneous vegetation where rape volunteers can appear, producing pollen and seeds that disperse to fields and other borders;
- The crop rotation of each field;
- The cultivation techniques applied to each crop (summer tillage, tillage for seed bed preparation, sowing date and density, herbicide applications, cutting dates and seed loss at rape harvest), and
- The type of the simulated gene, as well as the genotype of the rape seed varieties.

For the purpose of our research, a large-risk field plan is used with a small and rectangular central field surrounded by large neighbour fields (see Fig. 6.2), a combination which maximises pollen and seed input into the central field. Each simulation starts with an empty soil seed bank and simulates a period of 25 years. For each year, the crops and the cultivation techniques are chosen randomly, as well as the genetic variables. The only exception is the crop grown during the 25th year in the central field which is always a non-GM oilseed rape. The major output variables of GeneSys are, for each field and year, the number of rape seed plants, the proportions of these plants with and without GM seeds, the amount of seeds produced, and the seed bank containing the viable rape seeds in soil. Of the 25 simulated years of each simulation, full details on input and output variables were kept only for the last 4 years, resulting in a total of 1899 attributes. This was done because of the intended propositional analysis. In total 100,000 simulations were performed with GeneSys, thus the dataset consists of 100,000 examples, each example described by 1899 variables as mentioned above. In our study we are interested in the contamination with

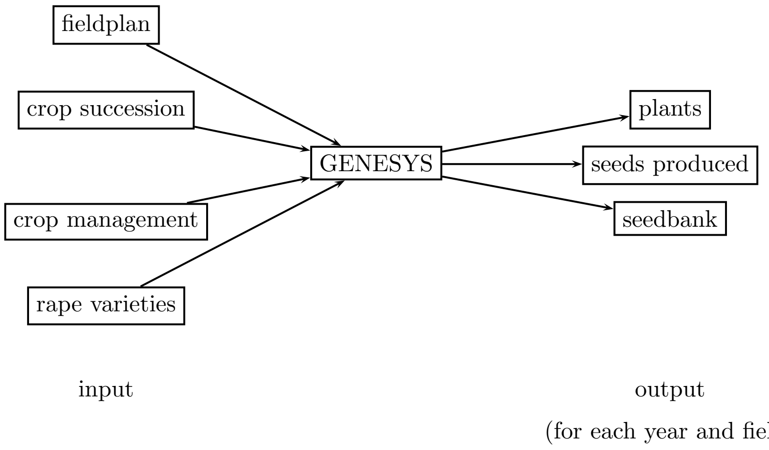


Figure 6.1: GENESYS: input and output variables.

GM seeds of the central field of the field plan in the 25th year (this is denoted as year 0).

6.2.3 The ILP representation

This analysis is intended to be a first step in a wider research for developing predictive models on gene flow. In this initial study, a simplified representation of the dataset is used. Next to the level of contamination, further information for each simulation includes

- for each field the number of years since the last GM oilseed rape crop was grown (denoted with *yearsSinceGM/2*),
- for each field and for each of the last four years (i.e., years 0 to 3)
  - the crop grown (GM oilseed rape, non-GM oilseed rape, winter crops, spring crops, autumn-sown set-aside, spring-sown set-aside, unsown set-aside, permanent set-aside),
  - the sowing date (measured in number of days since 1st January),

  denoted with *fieldDataYear/4*.

As an example, the information for the first simulation is given in Table 6.1. The background information further includes the following information for each field in the field plan (note that a fixed field plan is used, thus this information is valid for all simulations):

- the area of the field,

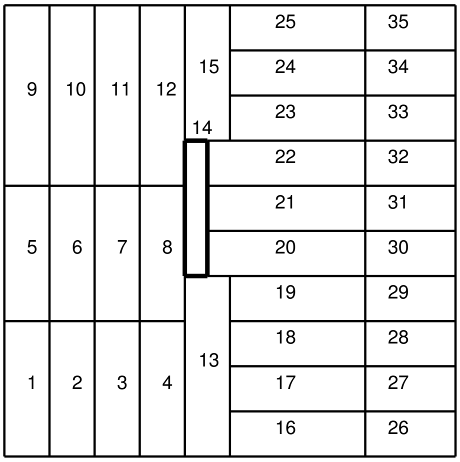


Figure 6.2: Large-risk field plan. The fields are numbered from 1 to 35. Field 14 is the central target field.

- whether the field is a neighbour of the central field,
- the neighbouring fields, including the neighbour type (common corner or common edge),
- the length of the common edge between the field and its neighbours of the corresponding type,
- the distance of the field to the central field (the distance is taken between the midpoints of the fields).

The general background knowledge for the application is summarized in Table 6.2.

6.2.4 Experiments

As mentioned before, we use the first order decision tree learner Tilde for analysing the dataset. We have three different sets of (non-overlapping) features:

- A first set of features are the central features: the information for the central field. This does not include any of the general background predicates in Table 6.2. It corresponds to a subset of the 1899 variables output by GeneSys that relates to the central field.
- A second set relates to the fields in the surroundings of the central field. These are the fields that are reachable from the central field using the

Table 6.1: Representation of the first example in the GeneSys dataset.

contamination(4.815339e-03).
yearsSinceGM(1,7).
yearsSinceGM(2,2).
...
yearsSinceGM(35,7).
fieldDataYear(1, 3, autumn-sown-set-aside, 301).
fieldDataYear(2, 3, spring-sown set-aside, 97).
...
fieldDataYear(35, 3, winter-crops, 272).
fieldDataYear(1, 2, spring-crops, 127).
...
fieldDataYear(35, 2, spring-sown-set-aside, 56).
...
...
fieldDataYear(35, 0, unsown-set-aside, 213).

Table 6.2: General background predicates for the GeneSys dataset.

area(1, 3.00).
area(2, 3.00).
...
targetneighbour(8).
targetneighbour(12).
...
neighbour(1,2,edge).
neighbour(1,5,edge).
...
lengthOfCommonEdge(1,2,300.00).
lengthOfCommonEdge(1,5,100.00).
...
distance(1, 542.63).
distance(2, 480.51).
...

neighbour, distance, or common edge relations (we will call these the field-by-field features).

- A last set of features is constructed through the use of complex aggregates. The aggregate literals that are used to start from are the following:

  - *count(_,fieldDataYear(Field,Year,gm_oilseed_rape,SowDate),R)*
  - *mode(Crop,fieldDataYear(Field,Year,Crop,SowDate),R)*
  - *max/min/avg(SowDate,fieldDataYear(Field,Yr,Crop,SowDate),R)*
  - *max/min/avg(Year,fieldDataYear(Field,Year,Crop,SowDate),R)*
  - *max/min/avg(Dist,distance(Field,Dist),R)*

  While the first three aggregates may be used as simple aggregates conditions, the last two aggregate conditions are only meaningful when conditions are applied to the aggregate query. These conditions include statements about the distance, area or contaminated fields. Thus, we could obtain complex aggregates that, for example, calculate the smallest distance to a contaminated field, or that compute the last sowing date of a contaminated field.

In our experiments, we consider increasingly richer language biases by first running experiments on the central features alone, then the central features complemented with the field-by-field features or the aggregate features, and finally the three feature sets together.

The dataset is split in a training set (60,000 examples) and test set (40,000 examples). Given the size of the dataset we used a sampling strategy to build the tree: at each node only 10,000 examples are used to search for the best test[1]. Afterwards, the whole dataset is split according to this best test found. The minimum number of examples a leaf has to cover was set to 2000 in order to get manageable trees, and 20% of the examples is set aside as validation set for pruning. For the settings with aggregates, the refinement operator based on μ-subsumption is used (see Sect. 4.4.2).

For each of the experiments, we report the predictive performance (measured as RRSE, the root of the relative mean squared error) on the test set, the tree size (number of internal nodes), and the number of these nodes that contain a test referring to the target, neighbour fields, and aggregates. The results are given in Table 6.3.

A first observation when looking at Table 6.3 is that, in general, the predictive models are not of good quality (the RRMSE is quite high in all settings). This may be due to the strong language bias that is imposed by only including the crop and sowing dates for each field and year.

[1]Note that this sampling strategy differs from the sampling strategy used in the random forest algorithm: in the latter case queries are sampled, whereas here we sample examples.

Table 6.3: Experimental results on the GeneSys dataset.

	central	centr.+f.-by-f.	centr.+agg.	centr.+f.-by-f.+agg.
RRMSE	72.52%	72.36%	72.51%	72.36%
nb. nodes	20	21	22	21
centr. nodes	20	16	16	15
f.-by-f. nodes	0	5	0	5
agg. nodes	0	0	6	1

```
targetfield(B),fieldDataYear(B,0,C,D),D < 252 ?
+yes: yearsSinceGM(B,E),E > 1 ? [0.192] (25907 ex.)
|      +yes: ... [0.131] (22504 ex.)
|      +no:  [0.598] (3403ex.)
+no: targetfield(U),yearsSinceGM(U,V),V > 2 ? [0.023] (34093 ex.)
|      +yes: ... [0.011] (25347 ex.)
|      +no: ... [0.059] (8746 ex.)
```

Figure 6.3: Part of the tree with target information.

When we compare the predictive error between the settings, we hardly see any difference. Moreover, all resulting trees have the same structure. The top tests, together with their average contamination level, are shown in Fig. 6.3. The test in the root checks whether the sowing date in the central field in the current year was before day 252. If yes, we check whether the last GM oilseed rape in the central field was grown more than one year ago. The tree contains 20 internal nodes and is of depth 6. If relational information is included, this only affects the tree from depth 4 on. Examples of relational field-by-field tests include

$distance(F, D), D < 258, fieldDataYear(F, 1, gm\_oilseed\_rape, SD)$, and
$targetNeighbour(F), fieldDataYear(F, 2, C, SD), SD < 233$.

The first example checks whether there is a field that had GM oilseed rape planted one year ago and is situated within distance 258 of the central field. Examples of complex aggregate conditions include

$avg(SD, (fieldDataYear(F, 0, C, SD), targetNeighbour(F)), Avg),$
$\quad Avg < 213$, and
$count(\_, (fieldDataYear(F, Y, gm\_oilseed\_rape, SD), distance(F, D),$
$\quad D < 301), Cnt), Cnt > 4$.

The first aggregate condition checks whether the average sowing date of the immediate neighbours of the central field in the current year is before day 213. The second aggregate counts the number of *fieldDataYear* predicates with a field within a distance of 301 meters from the central field that had GM oilseed rape in the last 4 years. This last aggregate is the only one that remains when adding relational information both on a field-by-field basis and with aggregates.

In order to have a better view on the importance of the relational data, we ran another experiment in which we discard information about the central field and used only relational information. The question was thus: can we obtain the same level of performance in predicting the contamination of the target field when only considering information about the surrounding fields (using both field-by-field and aggregated features)? The tree we obtained in this experiment has a RRMSE of 85.35%, and has 21 internal nodes, of which 4 contain complex aggregate conditions. Interestingly, the top node contains an aggregate condition:

$$max(SD, (fieldDataYear(F, 0, C, SD), distance(F, D), D < 200), Max),\\ \quad Avg < 252.$$

This condition tests whether the latest sowing date on the current year for the fields within distance 200 of the central field was before day 252. Next to the central field, only 2 fields are situated within this distance. Note that this test is very similar to the test in the top node of the trees that do include target information. When learning a tree using only aggregation features, a poor RRMSE value of 93.59% is obtained.

To summarize the experiments, we have learned that, for this particular dataset and with this particular language bias, the best prediction models are obtained when information about the target field is included in the language bias. Furthermore, the models with both central field related and relational features do not provide a noticeable improvement over the model with only central field features. However, it has been shown that the surrounding fields do indeed contain some information about the contamination of the central field.

6.2.5 Summary

In this work we used the first order decision tree learning system Tilde to predict adventitious presence of genetically modified varieties in conventional oilseed rape crops. This kind of study is important, since the possibility of genetically modified crops mixing with conventional crops is a concern today.

In our analysis, we used a dataset with 100,000 simulations of the cultivation practices on a fixed field pattern. The main question to be answered was whether surrounding fields have any influence on the contamination with genetically modified varieties in conventional oilseed rape crops. Intuitively, the answer would be positive given that crops can be contaminated by other crops simply by pollen being blown by wind from one field to another.

We have constructed three sets of features: central features only involving the field for which contamination needs to be predicted, and relational features

```
1 METABOLISM
1/1 amino acid metabolism
1/2 nitrogen and sulfur metabolism
...
2 ENERGY
2/1 glycolysis and gluconeogenesis
...
```

Figure 6.4: A part of the hierarchical FunCat classification scheme.

regarding the surrounding fields, either on a field-by-field basis or aggregated by using complex aggregate conditions. Experiments showed that, although the surrounding fields do have an influence on the contamination of the target field, this does not yield better predictive performances compared to only using features concerning the central field. However, this study was only an initial step in this analysis. Further work includes to consider other language biases and to move from the fixed field plan to variable plans.

6.3 A biological application

In this section we present a biological application, introduced at the Inductive Logic Programming 2005 Challenge, where the task is to build accurate models predicting gene function in the genome of the yeast *Saccharomyces cerevisiae*, based on relational homology and secondary structure data. The prediction is non-trivial as each gene is labeled with a set of functional classes instead of just one class and the classes are organized in a class hierarchy.

The problem is further described in Sect. 6.3.1, where we also discuss the method that will be used to analyse the dataset. This method consists of two steps: first, a relational frequent pattern mining system is used to obtain the most frequently occurring patterns (including complex aggregates) in the data (Sect. 6.3.2), and afterwards, a propositional predictive clustering tree (Sect. 6.3.3) is applied that uses the constructed features. Experimental results are presented in Sect. 6.3.4, and conclusions are formulated in Sect. 6.3.5.

6.3.1 Problem description

The goal of this application is to build accurate models predicting gene function in the genome of the yeast *Saccharomyces cerevisiae* (baker's or brewer's yeast). The data set provided for the challenge records information about 3,894 genes, and each gene is annotated with a set of functional classes assigned according to the FunCat classification scheme of the Munich Information Center for Protein Sequences (MIPS). FunCat is a hierarchical classification scheme; a small part is shown in Figure 6.4.

The classification setting of the data (each instance is labeled with one or more classes, each class selected from a class hierarchy) is known as hierarchical multilabel classification (HMC) (Blockeel et al. 2002). A simple approach to solve a HMC problem is to ignore the hierarchy and to learn separate models for each individual class (indicating whether a single instance belongs to the class or not). In this work, we instead follow the approach proposed by Struyf et al. (2005), which is to use so-called predictive clustering trees (PCTs) (Blockeel et al. 1998) in combination with a distance metric designed for HMC. This has two main advantages:

- a single PCT can be used to predict most of the classes, which reduces the total size of the predictive model, and
- the hierarchical structure defined over the classes is taken into account.

This last property is important, because the hierarchy conveys relevant information about the similarity and differences between classes and also expresses the constraint that an instance belonging to a class also belongs to the parent class. Predictive clustering trees have been implemented in Tilde (Blockeel et al. 1998), but application of it to HMC has been worked out in Clus[2]. This is a propositional version of Tilde, designed for building predictive clustering trees, and where a distance metric that is specific to HMC has been implemented and successfully applied in the context of functional genomics (Struyf et al. 2005).

For our specific dataset, the input data that is available for each gene consists of two parts: homology data and predicted secondary structure data. The homology data is relational, and may be interesting to learn complex aggregates on, and the secondary structure data stores sequences. Since we are dealing with relational data, there are two approaches to analyse the dataset:

- we can implement the distance metric designed for HMC into Tilde, or
- we can use the Clus system and transform the relational data into the appropriate input format in a propositionalization step.

In this study, we have chosen for the second approach, which consists of two steps. In the first step, the structured input data (relational data and sequences) is propositionalized by constructing features with the relational frequent pattern mining system Warmr (Dehaspe and Toivonen 1999). The constructed features are relational features including complex aggregates for the homology data, and subsequences for the secondary structure data. This approach illustrates the use of complex aggregates in another context than decision learning, or even predictive learning. In the second step, the propositional data is used to build the required models.

[2]http://www.cs.kuleuven.be/~dtai/clus/

6.3.2 Propositionalization

The process of converting structured input data into a format suitable for propositional learning algorithms (a table where each row describes an instance with a fixed number of features or attributes) is known as propositionalization, this was introduced in Sect. 2.3.2. Since constructing the propositional table through joins often results in an overly large table, we decided to use an other approach. We chose to use the relational frequent pattern mining algorithm WARMR (Dehaspe and Toivonen 1999) to propositionalize the data. This algorithm computes the set F of all queries expressed in a given hypothesis language that succeed for at least a proportion *minfreq* of the available instances (throughout this analysis, *minfreq* = 5%). The WARMR system is included in the ACE-ilProlog data mining system (ACE 2004; Blockeel et al. 2006), which also contains TILDE. It makes use of the same refinement operator as TILDE, and, hence, the extensions of the refinement operator (based on α- or μ-subsumption) that produce complex aggregate conditions are also available in WARMR.

Each query in the set F is used as a binary feature in the propositional representation, which takes the value true (false) for the instances for which the query succeeds (fails). In the following, we discuss the features that were constructed for the challenge data.

6.3.2.1 Homology data

The homology data stores for each yeast gene a number of similar SwissProt genes, together with their properties.

A first set of features that we construct based on this data are of the form $\texttt{eval}(G_j, S), S < s, \text{Tests}(G_j)$, and indicate that there exists a SwissProt gene G_j that is sufficiently similar and that fulfills the tests in $\text{Tests}(G_j)$. By restricting the search depth of WARMR to queries with $|\text{Tests}(G_j)| \leq 3$ we obtain 105,456 features. Some examples, together with their frequencies, are shown below.

$\texttt{eval}(G_j, S), S < 3 \cdot 10^{-5}, \texttt{organismclass}(G_j, \texttt{bacteria})$	72.5%
$\texttt{eval}(G_j, S), S < \infty, \texttt{keyword}(G_j, \texttt{inner\_membrane})$	57.7%
$\texttt{eval}(G_j, S), S < 0.14, (\texttt{molweight}(G_j, W), W < 35317^3)$	87.3%
$\texttt{eval}(G_j, S), S < 0.14, \texttt{dbref}(G_j, \texttt{maizedb}), \texttt{dbref}(G_j, \texttt{mendel})$	24.6%
$\texttt{eval}(G_j, S), S < 0.14, \texttt{dbref}(G_j, \texttt{embl}), (\texttt{seqlength}(G_j, L), L > 316)$	85.7%

Feature 1 tests for example if the given gene is homologous ($S < 3 \cdot 10^{-5}$) to a gene G_j originating from a bacterium. Note that organisms are classified according to a hierarchy with bacteria, viruses, eukaryota, and archaea the top-level classes.

Each feature in a second set of features is a test on the outcome of a complex aggregate. In particular, we use aggregates that compute the most frequent value (mode) of the database references, keywords, and top-level classification (of the originating organism) over the similar SwissProt genes G_j. Each feature is of the form $\texttt{mode}(V, (\text{Pred}(G_j, V), \text{Tests}(G_j)), M), M = m$, with V the variable to be aggregated and M the resulting mode. We used $|\text{Tests}(G_j)| \leq 1$ because with *mode* it is not possible to use the efficient refinement operator based on μ-subsumption (neither the aggregate function, nor the aggregate query, nor the value to compare with can be refined in a monotone way). We obtain 361 of these complex aggregates with WARMR. Some examples are shown below.

$\texttt{mode}(D, (\texttt{dbref}(G_j, D), \texttt{eval}(G_j, S), S < 0.54), M), M = \texttt{embl}$	91.4%
$\texttt{mode}(K, (\texttt{keyword}(G_j, K), \texttt{seq\_length}(G_j, L), L > 1056), M), M = \texttt{repeat}$	56.7%
$\texttt{mode}(K, \texttt{keyword}(G_j, K), M), M = \texttt{transmembrane}$	34.6%
$\texttt{mode}(C, (\texttt{topclass}(G_j, C), \texttt{dbref}(G_j, \texttt{embl})), M), M = \texttt{bacteria}$	18.7%

The homology data also stores similarity scores with particular other yeast genes. In this case, no extra information is available for the G_j. Therefore, we decided to construct features of the form $\texttt{yeast\_to\_yeast}(g_j, S), S < s$, with g_j a particular yeast gene identifier. We obtain 139 features; two of them are shown below.

$\texttt{yeast\_to\_yeast}(\texttt{ytq0070}, S), S < 1.2$	14.0%
$\texttt{yeast\_to\_yeast}(\texttt{ytq0105}, S), S < 0.005$	5.4%

6.3.2.2 Predicted secondary structure data

A second part of the data available for the genes is the predicted secondary structure data. The secondary structure is a sequence of stretches that are classified as alpha helix (a), beta sheet (b) or random coil (c). The class and length of each stretch is recorded in the challenge data. The features that we construct with WARMR for this part are frequent subsequences. The number of stretches in the subsequences varies from 1 to 8 (no frequent subsequences with 9 stretches were found). In total, we obtain 6259 features; some are shown below.

a_{1-1}	98.7%
$\mathrm{a}_{8-15}, \mathrm{c}_{16-31}$	45.1%
$\mathrm{a}_{1-1}, \mathrm{c}_{4-7}, \mathrm{b}_{4-7}, \mathrm{c}_{4-7}, \mathrm{a}_{8-15}, \mathrm{c}_{4-7}, \mathrm{b}_{4-7}, \mathrm{c}_{1-1}$	6.6%

Feature 2 tests for example if the secondary structure contains a subsequence consisting of a alpha helix with a length between 8 and 15 and a random coil with a length between 16 and 31. Note that the stretch length in the subsequences is discretized into exponentially increasing intervals of the form 2^i to $2^{i+1} - 1$.

The secondary structure data also includes for each gene the proportion of alpha helices, beta sheets and random coils. We also include these three numeric attributes in the propositional representation.

6.3.3 Predictive clustering trees

In this section, we briefly discuss predictive clustering trees (PCTs) (Blockeel et al. 1998) and their application to hierarchical multilabel classification (HMC) (Blockeel et al. 2002; Struyf et al. 2005).

PCTs generalize decision trees by viewing them as a hierarchy of clusters: the top-node corresponds to one cluster containing all data, which is recursively partitioned into smaller clusters while moving down the tree. PCTs are constructed with a standard top-down induction algorithm similar to that of C4.5 (Quinlan 1993), but use instead of information gain, intra-cluster variance summed over the subsets induced by the test. Minimizing intra-cluster variance results in homogeneous leaves, which in turn results in accurate predictions. PCTs can be used for any prediction task as long as a suitable distance metric on the target variable(s) can be defined (the distance metric is used to compute the intra-cluster variance heuristic).

PCTs can be used for HMC by defining a suitable distance metric. We use the metric defined by Struyf et al. (2005). To compute this metric, the set of classes S associated with each example e is represented as a vector v with one component for each class in the hierarchy; the components corresponding to the classes in S are set to one, the others to zero. The distance metric is the weighted Euclidean distance on the vector representation and the weights are used to take into account the hierarchical structure (i.e., the weight of a component v_k is set to $w_k = 0.75^{\text{level}(v_k)}$, with $\text{level}(v_k)$ the level on which the corresponding class occurs in the hierarchy[4]).

In the multilabel classification setting, a tree predicts a vector of binary class attributes, predicting positive for a component (a class) if at least 50% of the examples in the leaf belongs to that class.

6.3.4 Experiments

We apply the Clus system, which implements PCTs for HMC, to the propositional data set described in Section 6.3.2. The dataset contains 3,894 examples and 112,271 input attributes. The experiments are based on a three-way split of this data set: a training set, a validation set and a test set. The test set contains 33% of the data. The remaining 66% are split again using a 66%/33% split to create the training and validation set. The validation set is used to remove insignificant predictions from the PCT, by means of a significance test based on the hypergeometric distribution with a significance level of 0.05. Using this test has been shown to increase precision; details can be found in Clare (2003).

The Clus system uses an F-test as stopping criterion. Smaller values for the F significance level result in smaller trees. Results for different values

[4]The value 0.75 was selected ad-hoc. By using a value smaller than 1.0, errors at a deeper level receive a smaller weight.

Table 6.4: Results for different F values.

F-Test:	0.001	0.005	0.01	0.05	0.1	1.0
Average precision:	66.5	68.9	66.4	63.4	58.4	59.7
Coverage:	50.9	51.1	35.7	48.5	47.6	40.0
#Classes:	12	22	27	66	93	106
#Leaves:	7	12	16	64	133	286

```
$\texttt{mode}(C, (\texttt{topclass}(G, C), \texttt{dbref}(G, \texttt{embl})), M), M = \texttt{bacteria}$
+yes:$\texttt{eval}(G, S), S < 3 \cdot 10^{-5}, \texttt{orgcls}(G, \texttt{bacteria}), \texttt{sqlen}(G) > 316$
|       +yes:  1 (260ex.)
|       +no:   none (67ex.)
+no:    $\texttt{eval}(G, S), S < 3 \cdot 10^{-5}, \texttt{dbref}(G, \texttt{prints}), \texttt{dbref}(G, \texttt{prosite}), \texttt{keyw}(G, \texttt{in.memb})$
        +yes:  $\texttt{eval}(G, S), S < 3 \cdot 10^{-5}, \texttt{dbref}(G, \texttt{pir}), \texttt{molwt}(G) < 48391, \texttt{sqlen}(G) > 432$
        |       +yes:  20, 20/1, 20/9 (98ex.)
        |       +no:   none (47ex.)
        +no:   $\texttt{eval}(G, S), S < 0.14, \texttt{orgcls}(G, \texttt{diptera}), \texttt{sqlen}(G) < 316, \texttt{molwt}(G) > 35317$
                +yes:  1, 1/4, 1/4/1, 10, 10/3, 14, 14/7, 14/7/3 (59ex.)
                +no:   ...
```

Figure 6.5: Part of the PCT that was used to generate the submitted predictions.

are shown in Table 6.4. In accordance to Clare (2003), we report the tree's performance in terms of precision rather than accuracy. (Component-wise) precision is defined as the fraction of correct predictions among all *positive* predictions. The table shows average precision (precision averaged over the predicted classes) and coverage (proportion of examples for which at least one class is predicted) computed over the test set together with the total number of classes predicted by the PCT and its number of leaves.

We select the tree with $F = 0.005$ as final model (this tree was used to submit our predictions to the challenge) because it yields the highest average precision. A part of the tree is shown in Figure 6.5. The leaf at the bottom of the figure for example represents a cluster in which the genes are predicted to have three functions: 1/4/1, 10/3, and 14/7/3. Note that the tests in the nodes above the leaf provide a description of the cluster. The component-wise precisions obtained with this tree are listed in Table 6.5.

The tree uses almost exclusively homology based features. It includes one test on the distribution of random coils (not visible in the figure), but in general it appears that the features based on the predicted secondary structure data are less important. They are used more often in trees for larger F values, but never close to the top. Aggregates are also not used often, but interestingly, the top-test includes a complex aggregation condition.

Table 6.5: Prior probabilities and component-wise precision obtained by the selected PCT ($F = 0.005$).

Class	Prior	Clus
1	34.2	81.2
1/4	9.3	81.2
1/4/1	8.9	81.2
10	21.3	51.3
10/3	14.5	51.3
11	22.8	47.3
12	10.7	77.5
12/1	6.7	77.5
12/1/1	5.6	77.5
14	24.5	88.6
14/7	12.9	86.4
14/7/3	4.1	87.2
14/7/11	1.9	60.0
14/13	5.5	60.0
14/13/1	4.1	60.0
14/13/1/1	2.7	60.0
16	22.0	77.8
16/19	5.2	77.8
16/19/3	4.4	77.8
20	22.8	94.8
20/1	13.4	84.5
20/9	15.5	43.1

6.3.5 Summary

The goal of this application was to build accurate models for predicting gene function in the genome of the yeast *Saccharomyces cerevisiae.* The approach followed in this study is a propositionalization approach. In a first step, the structured input was converted into a propositional format by means of feature generation with Warmr. Since Warmr uses the same refinement operator as Tilde, it is able to learn complex aggregate conditions. In the second step, the propositional data was used to build models with Clus, a system that builds predictive clustering trees. Our resulting model has a complex aggregate condition in the top node, yields an average precision of 68.9%, covers 51.1% of the examples, and predicts a reasonable number of classes. Of the four submissions to the challenge, we ended second, with a score that was very close to the score of the winning team.

6.4 Conclusion

This chapter has presented two applications in which complex aggregates are used.

The first application dealt with an agricultural dataset about genetically modified crops. The task was to predict adventitious presence of genetically modified varieties in a field where conventional oilseed rape crops are planted and to investigate the influence of surrounding fields. It turned out that properties of neighbouring fields did occur in the models (either aggregated or not), but that predictive performance was not improved by it.

In the second application we analysed a biological dataset containing genes in the genome of baker's or brewer's yeast. The task was to predict the function of the genes, making use of (possibly aggregated) homology data and secondary structures. The data was first propositionalized using a relational frequent pattern miner and afterwards analysed with a propositional predictive clustering tree learner. The resulting tree contained almost exclusively homology based features, with a complex aggregate condition in the root node.

Chapter 7

Conclusions

In this chapter we summarize the most important results and provide some directions for further work.

7.1 Summary

The work presented in this dissertation is situated in the fields of machine learning and data mining, where one is concerned with extracting a set of useful patterns from data. More precisely, we concentrate on predictive learning, where the patterns predict one specific property (attribute) of the examples in the dataset and we assume that the data is contained in a relational database, i.e., information relevant for making a single prediction is spread over a set of tuples from different relations. We have argued that current relational learners handle such a set by implicitly using aggregations over a condition-defined subset of it, where either the aggregation function itself or the conditions are trivial, and that the combination of more general aggregation functions (i.e., other functions than "exists") with non-trivial conditions is worth exploring. This thesis analyses the problems that such a combination brings along, and proposes solutions for them.

7.1.1 Combining aggregate functions and selection conditions

In Chapt. 3 we have laid the basis for this combination. We have introduced so called complex aggregate conditions in the context of inductive logic programming (ILP), where most of the work in relational learning is situated. Complex aggregate conditions are aggregate conditions where the (first order) query representing the set to be aggregated is extended with a number of literals imposing selection conditions on that set. In ILP the patterns are predicate definitions and they are learned one clause at a time. For learning one clause, one usually starts with the most general clause and subsequently specializes it by applying a refinement operator based on θ-subsumption. We have extended the θ-subsumption relation in order to take into account refinements of the aggregate conditions, the extension is called α-subsumption.

Using a refinement operator based on α-subsumption to form a combination of aggregates and selections presents several difficulties. First, the space of refinements considered by the ILP learner is significantly expanded. Second, the property that a clause c_1 θ-subsumes a clause c_2 implies that c_2 is more specific than c_1 is lost. Refinement under α-subsumption may yield a specialization or a generalization, or even none of both. This behaviour can be related to two issues:

- In the first order logic context, an aggregate function can be interpreted in several ways when applied to a complex aggregate query (roughly, as being applied to *sets* or to *bags*).
- This behaviour can also be related to the monotonicity properties of the aggregate conditions.

To deal with the first issue, we propose a semantics for aggregate functions, such that, when refining the set to be aggregated, the function is always applied to a smaller set. The second issue is dealt with by investigating in more detail the monotonicity properties of aggregate conditions, and this along three dimensions: the aggregate functions, the set to aggregate over, and the threshold values to compare the result with. This study leads to the development of a new subsumption relation, called μ-subsumption, that guarantees specializations.

7.1.2 Learning trees with complex aggregate conditions

In Chapt. 4 we have applied the complex aggregate conditions in a learning system. We have chosen Tilde (Blockeel and De Raedt 1998), a first order decision tree learner, for investigating the practical use of complex aggregates. First, the system was modified to incorporate simple aggregate conditions (i.e., aggregate conditions without selections on the set to be aggregated) in the candidate refinements it considers at each node of the tree being built. Next, we have presented two approaches to introduce complex aggregate conditions in the learning algorithm: refining an aggregate condition that occurs higher in the tree, and directly adding a complex aggregate condition by extending the lookahead mechanism (Blockeel and De Raedt 1997) to the level of aggregates. Both approaches are implemented by providing a refinement operator based on α-subsumption, both because it was the first refinement operator developed in this thesis and because generalizations are not a problem in Tilde, which makes it possible to use this simple refinement operator. A problem that does occur, especially when including aggregates with the lookahead mechanism, is the substantial expansion of the feature space.

To improve efficiency, two techniques were developed. The first technique is to sample the refinement space. To that aim, Tilde is upgraded to a first order random forest learner. The second technique is to structure the refinement space in a general to specific way by applying a refinement operator based on

μ-subsumption. When an aggregate condition fails for an example, there is no need to test the aggregate conditions that are more specific, and as such, parts of the refinement space can be pruned.

An extensive experimental evaluation of all presented methods has been performed. The main results can be summarized as follows.

- The use of simple aggregate conditions in Tilde always results in a significant predictive performance improvement. The difference between simple and complex aggregate conditions is more subtle, but becomes very apparent when testing on synthetic datasets where the target function has been designed to involve complex aggregates.
- The use of a refinement operator based on μ-subsumption gives an efficiency gain of up to factor 3.6 on the tested datasets. The efficiency gain yielded by first order random forests may be more or less, depending on the sampling ratio used and the number of trees in the forest.
- Although random forests may or may not improve efficiency, they always result in a predictive performance improvement over Tilde.
- When comparing predictive performance results of Tilde or the first order random forest induction algorithm with complex aggregates to other learning systems, there is a noticeable improvement, even when compared to systems that use aggregates.

7.1.3 Using complex aggregates in prediction functions

While in Chapt. 3 and 4 complex aggregates were applied in the condition part of the hypothesis, in Chapt. 5, we investigate their use in the conclusion part. We do this in the context of model trees, which are regression trees that construct linear equations in the leaves. In particular, we incorporate complex aggregates in these linear equations. For that aim, we upgrade Tilde to a first order model tree learner. We do not consider all possible complex aggregates in the linear equation at a leaf, but only include those (or other numeric predictors) that have been chosen as a test in the nodes on the path from the root to the leaf.

The main difficulty in this work is to design a suitable heuristic function for choosing the best test at a node: it needs to take into account the fact that linear models are built in the leaves, while being efficient to compute. This second property is important, since existing heuristic functions that assume linear models in the leaves are at least quadratic in the number of attributes and this number becomes very high when learning complex aggregates. A considerable part of the chapter has been devoted to finding such a heuristic function. We have studied different heuristic functions used in propositional model tree learners, and have proposed a new function that differs from the complexity of the most efficient approaches only with a constant factor, while being better targeted towards learning linear models in the leaves. We have then applied this function to our first order model tree learner.

Experimental results demonstrate that, if many numeric attributes occur in the dataset (e.g., in the context of learning aggregates), our system outperforms normal regression tree learners. When comparing to a model tree learner that uses a more complex heuristic function, the comparison in predictive performance is less obvious, while our system in general produces shorter trees.

7.1.4 Applications of complex aggregates

In Chapt. 6 we present two applications where complex aggregate conditions are used in real life applications. In the first application an agricultural dataset is analyzed where the task is to predict adventitious presence of genetically modified varieties in conventional oilseed rape crops. The idea is that fields in the neighbourhood of a target field may have an influence on the contamination of this target field. Complex aggregates are applied to see whether this influence is present on the regional level, rather than on a field-by-field level. For this task we used TILDE with the μ-subsumption based refinement operator. Results showed that, although the aggregates (or the field-by-field relational properties) do contain some information regarding the contamination of the target field, for this particular simulated dataset, most of the information is contained in the target field itself.

The second application is of biological nature. The task is to build accurate models for predicting gene function in the genome of baker's or brewer's yeast, based on relational homology and secondary structure data. The prediction task is non-trivial as each gene is labelled with a set of functional classes instead of just one class and the classes are organized in a class hierarchy. To tackle this application, we use a propositional predictive clustering tree, which is specifically targeted to this kind of learning task. In order to apply this algorithm, the relational data is first propositionalized using the frequent pattern mining algorithm WARMR (Dehaspe and Toivonen 1999). Since WARMR is contained in the same software package as TILDE it makes use of the same refinement operator, and thus also incorporates complex aggregates. The resulting tree that we obtained had a complex aggregate condition in its root node.

7.2 Further work

7.2.1 Combining aggregates and selections

In this work we have combined the concepts of aggregation and selection by introducing complex aggregate conditions. These conditions aggregate over a subset of elements, for which a selection condition is fulfilled. For example, in the context of Mutagenesis, this results in tests as "the maximal charge for the carbon atoms is larger than 0.6". However, one could also combine aggregates and selections by doing in a sense the opposite: imposing selection conditions on the outcome of an aggregate. For example, this could result in tests as "the element of the atom with the maximal charge is carbon" or "the number of atoms that are bounded to the atom with the maximal charge is larger than 4". Whereas the outcome of an aggregate now is a (numeric) value, it would have to include a key (or a set of keys) in the proposed approach, such that properties of the object(s) associated with the key(s) could be tested.

Whereas the use of this approach would probably be limited to the aggregate functions *max* and *min*, several examples of patterns involving this combination of aggregates and selections occur in real life:

- In a chess play, each opponent has to think about what piece he will move next. To that aim, players (or computer programs) calculate an evaluation score for each possible move. The next piece to be moved will be the piece corresponding to the move with the highest score.
- In football, the trophy for top scorer of the year goes to the player with the highest number of goals.

In the first example, we are not interested in the value of the highest score, but in the piece corresponding to the move with the highest score. Similarly, in the second example, the value of interest is not the number of goals, but the name of the person associated with these goals.

7.2.2 Finding suitable threshold values

In this thesis, the basic language components for aggregate conditions need to be specified in the language bias: the aggregate functions, the aggregate queries, the comparison operator, and a number of threshold values. These threshold values may be given by the user or produced by a discretization procedure. A discretization algorithm typically finds a number of intervals in the domain of a numeric attribute. In TILDE these threshold values may be chosen in two different ways: such that an equal number of examples falls into each interval, or such that the class entropy is minimized in each interval (Blockeel and De Raedt 1997). The cut points of the intervals are then used as thresholds in the inequality tests.

However, discretization is not sufficient for dealing with aggregates. There are three main problems:

- The domain of the aggregate function may differ from the domain of the attribute to be aggregated. For example, *count* returns natural numbers, but is usually applied to key attributes. Hence, discretization procedures can not be used to produce a set of threshold values for *count*.

- The aggregate function may result in values in the same domain, but in a different range than the attribute to be aggregated. For example, *sum* can result in values that do not fall in the range of the numeric attribute.
- Even if in the same domain and range of the numeric attribute, different aggregate functions may have different sets of optimal threshold values. For example, *max* may prefer threshold values that are larger than the optimal threshold values for *min*.

We can conclude that techniques are needed to automatically find good values to compare the result of an aggregate function with.

References

ACE (2004). The ACE data mining system. Katholieke Universiteit Leuven, Dept. of Computer Science, Declarative Languages and Artificial Intelligence research lab. `http://www.cs.kuleuven.ac.be/∼dtai/ACE/`.

Agrawal, R., H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo (1996). Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. The MIT Press.

Alexander, W. and S. Grimshaw (1996). Treed regression. *Journal of Computational and Graphical Statistics 5*, 156–175.

Appice, A., M. Ceci, and D. Malerba (2002). Kdb2000: An integrated knowledge discovery tool. In A. Zanasi, C. Brebbia, N. Ebecken, and P. Melli (Eds.), *Data Mining III, Series Management Information Systems*, Volume 6, pp. 531–540. Southampton, UK: WIT Press.

Appice, A., M. Ceci, and D. Malerba (2003). Mining model trees: a multi-relational approach. In T. Horváth and A. Yamamoto (Eds.), *Proceedings of the 13th International Conference on Inductive Logic Programming*, Volume 2835 of *Lecture Notes in Artificial Intelligence*, pp. 4–21. Springer-Verlag.

Berka, P. (2000). Guide to the financial data set. In A. Siebes and P. Berka (Eds.), *The ECML/PKDD 2000 Discovery Challenge.*

Blockeel, H. (1998). *Top-Down Induction of First Order Logical Decision Trees.* Ph. D. thesis, Department of Computer Science, Katholieke Universiteit Leuven. `http://www.cs.kuleuven.ac.be/∼ml/PS/blockeel98:phd.ps.gz`.

Blockeel, H. and M. Bruynooghe (2003). Aggregation versus selection bias, and relational neural networks. In *IJCAI-2003 Workshop on Learning Statistical Models from Relational Data, SRL-2003, Acapulco, Mexico, August 11, 2003.*

Blockeel, H., M. Bruynooghe, S. Džeroski, J. Ramon, and J. Struyf (2002). Hierarchical multi-classification. In *Proceedings of the ACM SIGKDD 2002 Workshop on Multi-Relational Data Mining (MRDM 2002)*, pp. 21–35.

Blockeel, H. and L. De Raedt (1997). Lookahead and discretization in ILP. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, Volume 1297 of *Lecture Notes in Artificial Intelligence*, pp. 77–85. Springer-Verlag.

Blockeel, H. and L. De Raedt (1998, June). Top-down induction of first order logical decision trees. *Artificial Intelligence 101*(1-2), 285–297.

Blockeel, H., L. De Raedt, and J. Ramon (1998). Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning*, pp. 55–63.

Blockeel, H., L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele (2002). Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research 16*, 135–166.

Blockeel, H., L. Dehaspe, J. Ramon, J. Struyf, A. Van Assche, C. Vens, and D. Fierens (2006). The ace data mining system: User's manual. `http://www.cs.kuleuven.be/∼dtai/ACE`.

Blockeel, H. and W. Uwents (2004). Using neural networks for relational learning. In *Proceedings of SRL-2004, the ICML-2004 Workshop on Statistical Relational Learning and its connections to other fields, SRL-2004, Banff, Canada, July 8, 2004*.

Bratko, I. (2001). *Prolog Programming for Artificial Intelligence.* Addison-Wesley. 3rd Edition.

Breiman, L. (1996). Out-of-bag estimation.
ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation.ps.

Breiman, L. (2001). Random forests. *Machine Learning 45*(1), 5–32.

Breiman, L., J. Friedman, R. Olshen, and C. Stone (1984). *Classification and Regression Trees.* Belmont: Wadsworth.

Brodley, C. and P. Utgoff (1995). Multivariate decision trees. *Machine Learning 19*, 45–77.

Chapman, P., J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer, and R. Wirth (2000). CRISP-DM 1.0: Step-by-step data mining guide. Technical report, CRISP-DM Consortium.

Chaudhuri, P., M. Huang, W. Loh, and R. Yao (1994). Piecewise-polynomial regression trees. *Statistica Sinica 4*, 143–167.

Clare, A. (2003). *Machine learning and data mining for yeast functional genomics.* Ph. D. thesis, University of Wales, Aberystwyth.

Colbach, N., C. Clermont-Dauphin, and J. Meynard (2001). Genesys: A model of the influence of cropping system on gene escape from herbicide tolerant rapeseed crops to rape volunteers. *Agriculture, Ecosystems and Environment 83*, 235–270.

Das, S. K. (1992). *Deductive Databases and Logic Programming.* Addison-Wesley.

De Raedt, L. (1997). Logical settings for concept learning. *Artificial Intelligence 95*, 187–201.

De Raedt, L. (1998). Attribute-value learning versus inductive logic programming: the missing links (extended abstract). In D. Page (Ed.), *Proceedings of the Eighth International Conference on Inductive Logic Programming*, Volume 1446 of *Lecture Notes in Artificial Intelligence*, pp. 1–8. Springer-Verlag.

De Raedt, L. and W. Van Laer (1995). Inductive constraint logic. In K. P. Jantke, T. Shinohara, and T. Zeugmann (Eds.), *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, Volume 997 of *Lecture Notes in Artificial Intelligence*, pp. 80–94. Springer-Verlag.

Dehaspe, L. and H. Toivonen (1999). Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery 3*(1), 7–36.

Dehaspe, L. and H. Toivonen (2001). Discovery of relational association rules. In S. Džeroski and N. Lavrač (Eds.), *Relational Data Mining*, pp. 189–212. Springer-Verlag.

Dempster, N., A. Laird, and D. Rubin (1977). Maximum likelihood from incomplete data via the EM algorithm. *J. R. Statist. Soc. B 39*, 185–197.

Demsar, J. and B. Zupan (2004). Orange: From experimental machine learning to interactive data mining. `http://www.ailab.si/orange`.

Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation 10*(7), 1895–1924.

Dobra, A. and J. Gehrke (2002). Secret: A scalable linear regression tree algorithm. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

Džeroski, S., S. Schulze-Kremer, K. R. Heidtke, K. Siems, D. Wettschereck, and H. Blockeel (1998, July-August). Diterpene structure elucidation from ^{13}C NMR spectra with inductive logic programming. *Applied Artificial Intelligence 12*(5), 363–384.

Džeroski, S. (1995). *Numerical constraints and learnability in inductive logic programming.* Ph. D. thesis, University of Ljubljana (Slovenia).

Džeroski, S. (2001a). Data mining in a nutshell. In S. Džeroski and N. Lavrač (Eds.), *Relational Data Mining*, pp. 3–27. Springer-Verlag.

Džeroski, S. (2001b). Relational data mining applications: An overview. In S. Džeroski and N. Lavrač (Eds.), *Relational Data Mining*, pp. 339–364. Springer-Verlag.

Džeroski, S. and N. Lavrač (2001a). An introduction to inductive logic programming. In S. Džeroski and N. Lavrač (Eds.), *Relational Data Mining*, pp. 48–74. Springer-Verlag.

Džeroski, S. and N. Lavrač (Eds.) (2001b). *Relational Data Mining*. Springer-Verlag.

Elmasri, R. and S. Navathe (2004). *Fundamentals of Database Systems* (4nd ed.). Addison-Wesley.

Emde, W. and D. Wettschereck (1995). Relational instance based learning. In *Proceedings of the 1995 Workshop of the GI Special Interest Group on Machine Learning*.

Emde, W. and D. Wettschereck (1996). Relational instance-based learning. In L. Saitta (Ed.), *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 122–130. Morgan Kaufmann.

Fayyad, U., G. Piatetsky-Shapiro, and P. Smyth (1996). From data mining to knowledge discovery: An overview. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 495–515. The MIT Press.

Frawley, W., G. Piatetsky-Shapiro, and C. Matheus (1991). Knowledge discovery in databases: an overview. In G. Piatetsky-Shapiro and W. Frawley (Eds.), *Knowledge Discovery in Databases*, pp. 1–27. Cambridge, MA: MIT Press.

Getoor, L., N. Friedman, D. Koller, and A. Pfeffer (2001). Learning Probabilistic Relational Models. In S. Dzeroski and N. Lavrac (Eds.), *Relational Data Mining*, pp. 307–334. Springer-Verlag.

Hansen, L. and P. Salamon (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 993–1001.

Hastie, T., R. Tibshirani, and J. Friedman (2001). *The Elements of Statistical Learning*. Springer-Verlag.

Horvath, T. and S. Wrobel (2001). Towards discovery of deep and wide first-order structures: A case study in the domain of mutagenicity. In *Discovery Science 2001*, Volume 2226 of *Lecture Notes in Artificial Intelligence*, pp. 100–112.

Ivanovska, A., P. Panov, N. Colbach, M. Debeljak, S. Džeroski, and A. Messean (2006). Using simulation models and data mining to study co-existence of gm/non-gm crops at regional level. In *Proceedings of 20th International Conference on Informatics for Environmental Protection*, pp. 489–500.

Jacobs, N. (2004). *Relational Sequence Learning and User Modelling*. Ph. D. thesis, Department of Computer Science, Katholieke Universiteit Leuven.

Jaeger, M. (1997). Relational Bayesian networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pp. 266–273. Morgan Kaufmann Publishers.

Jensen, D., J. Neville, and M. Hay (2003). Avoiding bias when aggregating relational data with degree disparity. In *Proceedings of the 20th International Conference on Machine Learning.*

Karalic, A. (1992). Employing linear regression in regression tree leaves. In *European Conference on Artificial Intelligence*, pp. 440–441.

Karalič, A. and I. Bratko (1997). First order regression. *Machine Learning 26*, 147–176.

Knobbe, A., H. Blockeel, A. Siebes, and D. van der Wallen (1999). Multi-relational data mining. In H. Blockeel and L. Dehaspe (Eds.), *Proceedings of the 9th Belgian-Dutch Conference on Machine Learning*, Maastricht, The Netherlands, pp. 61–74.

Knobbe, A., M. de Haas, and A. Siebes (2001). Propositionalisation and aggregates. In L. De Raedt and A. Siebes (Eds.), *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery*, Volume 2168 of *Lecture Notes in Artificial Intelligence*, pp. 277–288. Springer.

Knobbe, A. and E. Ho (2005, October). Numbers in multi-relational data mining. In *Principles of Data Mining and Knowledge Discovery, Proceedings of the 9th European Conference*, pp. 544–551. Springer-Verlag.

Knobbe, A., A. Siebes, and B. Marseille (2002, August). Involving aggregate functions in multi-relational search. In *Principles of Data Mining and Knowledge Discovery, Proceedings of the 6th European Conference*, pp. 287–298. Springer-Verlag.

Knobbe, A., A. Siebes, and D. van der Wallen (1999). Multi-relational decision tree induction. In *Proceedings of PKDD-1999 - The Third European Conference on Principles and Practice of Knowledge Discovery in Databases*, Volume 1704 of *Lecture Notes in Computer Science*, Prague, Czech Republic, pp. 378–383. Springer.

Kohavi, R. and G. John. (1997). Wrappers for feature subset selection. *Artificial Intelligence 97*, 273–324.

Koller, D. (1999). Probabilistic relational models. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, Volume 1634 of *Lecture Notes in Artificial Intelligence*, pp. 3–13. Springer-Verlag.

Kramer, S. (1996). Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Cambridge/Menlo Park, pp. 812–819. AAAI Press/MIT Press.

Kramer, S., N. Lavrač, and P. Flach (2001). Propositionalization approaches to relational data mining. In S. Džeroski and N. Lavrač (Eds.), *Relational Data Mining*, pp. 262–291. Springer-Verlag.

Kramer, S. and G. Widmer (2001). Inducing classification and regression trees in first order logic. In S. Džeroski and N. Lavrač (Eds.), *Relational Data Mining*, pp. 140–159. Springer-Verlag.

Krogel, M.-A., S. Rawles, F. Železný, P. Flach, N. Lavrač, and S. Wrobel (2003). Comparative evaluation of approaches to propositionalization. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, Volume 2835 of *Lecture Notes in Artificial Intelligence*, pp. 194–217. Springer-Verlag.

Krogel, M.-A. and S. Wrobel (2001). Transformation-based learning using multi-relational aggregation. In *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, pp. 142–155.

Krogel, M.-A. and S. Wrobel (2003). Facets of aggregation approaches to propositionalization. In T. Horváth and A. Yamamoto (Eds.), *Proceedings of the Work-in-Progress Track at the 13th International Conference on Inductive Logic Programming*, pp. 30–39.

Langley, P. (1996). *Elements of Machine Learning.* Morgan Kaufmann.

Lavrač, N. and S. Džeroski (1994). *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood.

Lavrač, N., S. Džeroski, and M. Grobelnik (1991). Learning nonrecursive definitions of relations with LINUS. In Y. Kodratoff (Ed.), *Proceedings of the Fifth European Working Session on Learning*, Volume 482 of *Lecture Notes in Artificial Intelligence*, pp. 265–281. Springer-Verlag.

Lloyd, J. (1987). *Foundations of Logic Programming* (2nd ed.). Springer-Verlag.

Loh, W.-Y. (2002). Regression trees with unbiased variable selection and interaction detection. *Statistica Sinica 12*, 361–386.

Malerba, D., F. Esposito, M. Ceci, and A. Appice (2004). Top-down induction of model trees with regression and splitting nodes. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26*(5), 612–625.

McCarthy, J., M. Minsky, N. Rochester, and C. Shannon (1955). A proposal for the dartmouth summer research project on artificial intelligence.

McLachlan, G. (1992). *Discriminant analysis and statistical pattern recognition.* Addison-Wesley.

Merz, C. and P. Murphy (1996). UCI repository of machine learning databases `http://www.ics.uci.edu/~mlearn/mlrepository.html`. Irvine, CA: University of California, Department of Information and Computer Science.

Michalski, R. (1980). Pattern Recognition as Rule-Guided Inductive Inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence 2*, 349–361.

Michalski, R. (1987). Clustering. In S. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence*. Wiley.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill.

Muggleton, S. (Ed.) (1992). *Inductive Logic Programming*. Academic Press.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming 13*(3-4), 245–286.

Muggleton, S. and L. De Raedt (1994). Inductive logic programming : Theory and methods. *Journal of Logic Programming 19,20*, 629–679.

Nédellec, C., H. Adé, F. Bergadano, and B. Tausend (1996). Declarative bias in ILP. In L. De Raedt (Ed.), *Advances in Inductive Logic Programming*, Volume 32 of *Frontiers in Artificial Intelligence and Applications*, pp. 82–103. IOS Press.

Neville, J., D. Jensen, L. Friedland, and M. Hay (2003). Learning relational probability trees. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM.

Ng, R. T., L. V. S. Lakshmanan, J. Han, and A. Pang (1998). Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD International Conference on Management of Data*, pp. 13–24.

Pelov, N. (2004, apr). *Semantics of Logic Programs with Aggregates*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium. 152 + x pages.

Perlich, C. and F. Provost (2003). Aggregation-based feature invention and relational concept classes. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 167–176. ACM Press.

Plotkin, G. (1969). A note on inductive generalization. *Machine Intelligence 5*, 153–163.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning 5*, 239–266.

Quinlan, J. (1992). Learning with continuous classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pp. 343–348. World Scientific, Singapore.

Quinlan, J. R. (1988). Simplifying decision trees. In B. Gaines and J. Boose (Eds.), *Knowledge Acquisition for Knowledge-Based Systems*, pp. 239–252. London: Academic Press.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann series in Machine Learning. Morgan Kaufmann.

Ramon, J. (2002). *Clustering and Instance Based Learning in First Order Logic.* Ph. D. thesis, Department of Computer Science, K.U.Leuven.

Ray, S. and D. Page (2001). Multiple instance regression. In *Proceedings of the 18th International Conference on Machine Learning*, pp. 425–432. Morgan Kaufmann.

Srinivasan, A. (2003). Note on the use of statistical procedures as background predicates in ilp. Technical Report PRG-RR-03-09, Oxford University Computing Laboratory.

Srinivasan, A., S. Muggleton, and R. King (1995). Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt (Ed.), *Proceedings of the Fifth International Workshop on Inductive Logic Programming*, pp. 199–230. Department of Computer Science, Katholieke Universiteit Leuven.

Srinivasan, A., S. Muggleton, M. Sternberg, and R. King (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence 85*(1,2), 277–299.

Struyf, J., S. Džeroski, H. Blockeel, and A. Clare (2005). Hierarchical multi-classification with predictive clustering trees in functional genomics. In *Progress in Artificial Intelligence: 12th Portugese Conference on Artificial Intelligence*, Volume 3808 of *Lecture Notes in Computer Science*, pp. 272–283. Springer.

Torgo, L. (1997). Functional models for regression tree leaves. In *Proceedings of the 14th International Conference on Machine Learning*, pp. 385–393. Morgan Kaufmann.

Torgo, L. (2002). Computationally efficient linear regression trees. In *Proceedings of the 8th Conference of the International Federation of Classification Societies.* Springer.

Uwents, W. and H. Blockeel (2005). Classifying relational data with neural networks. In *Proceedings of 15th International Conference on Inductive Logic Programming, Bonn, Germany*, Volume 3625 of *Lecture Notes in Artificial Intelligence*, pp. 384–396. Springer.

Van Laer, W. and L. De Raedt (2001). How to upgrade propositional learners to first order logic: A case study. In S. Džeroski and N. Lavrač (Eds.), *Relational Data Mining*, pp. 235–261. Springer-Verlag.

Wang, Y. and I. Witten (1997). Inducing model trees for continuous classes. In *Proc. of the 9th European Conf. on Machine Learning Poster Papers*, pp. 128–137.

Weisberg, S. (1980). *Applied linear regression.* John Wiley & Sons.

Witten, I. and E. Frank (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann.

Witten, I. and E. Frank (2005). *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann. 2nd Edition.

Zantema, H. and H. Bodlaender (2000). Finding small equivalent decision trees is hard. *International Journal of Foundations of Computer Science 11*(2), 343–354.

Zucker, J.-D. and J.-G. Ganascia (1998). Learning structurally indeterminate clauses. In *Proceedings of the International Conference on Inductive Logic Programming*, pp. 235–244.

Index

Publication List

Articles in international reviewed journals

- C. Vens, and H. Blockeel, *A simple regression based heuristic for learning model trees*, Intelligent Data Analysis 10 (3), pp. 215-236, 2006.
  http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41951

- A. Van Assche, C. Vens, H. Blockeel, and S. Džeroski, *First order random forests: Learning relational classifiers with complex aggregates*, Machine Learning 64, pp. 149-182, 2006.
  http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42153

Contributions at international conferences

Published in proceedings

- C. Vens, J. Ramon, and H. Blockeel, *ReMauve, a relational model tree learner*, ILP'06, 16th International Conference on Inductive Logic Programming, Post Conference Proceedings (Muggleton, S. and Otero, R., eds.), 2006, accepted.

- C. Vens, J. Ramon, and H. Blockeel, *Refining aggregate conditions in relational learning*, Knowledge Discovery in Databases: PKDD 2006, 10th European Conference on Principles and Practice of Knowledge Discovery in Databases, Proceedings (Fürnkranz, J. and Scheffer, T. and Spiliopoulou, M., eds.), vol 4213, Lecture Notes in Artificial Intelligence, pp. 383-394, 2006.
  http://www.springerlink.com/content/433582r410g77426/

- C. Vens, J. Ramon, and H. Blockeel, *Re-Mauve, a relational model tree learner*, ILP'06, 16th International Conference on Inductive Logic Programming, Short Papers (Muggleton, S. and Otero, R., eds.), pp. 216-218, 2006.
  http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42376

- A. Ivanovska, C. Vens, and S. Džeroski, *Using ILP to study the presence of genetically modified variants in organic oil seed rape*, ILP'06, 16th

International Conference on Inductive Logic Programming, Short Papers (Muggleton, S. and Otero, R., eds.), pp. 107-109, 2006.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42495

- W. Uwents, C. Vens, A. Van Assche, and H. Blockeel, *Learning aggregations and selections with relational neural networks*, Proceedings of the Workshop on Sub-Symbolic Paradigms for Learning in Structured Domains (Gori, M. and Avesani, P., eds.), pp. 112-121, 2005.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41894

- J. Struyf, C. Vens, T. Croonenborghs, S. Džeroski, and H. Blockeel, *Applying predictive clustering trees to the inductive logic programming 2005 challenge data*, Inductive Logic Programming, 15th International Conference, ILP 2005, Late-Breaking Papers (Kramer, S. and Pfahringer, B., eds.), pp. 111-116, 2005.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41826

- C. Vens, A. Van Assche, H. Blockeel, and S. Džeroski, *First order random forests with complex aggregates*, Inductive Logic Programming, 14th International Conference, ILP 2004, Proceedings (Camacho, R. and King, R. and Srinivasan, A., eds.), vol 3194, Lecture Notes in Computer Science, pp. 323-340, 2004.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41318

- A. Van Assche, C. Vens, H. Blockeel, and S. Džeroski, *A random forest approach to relational learning*, ICML 2004 Workshop on Statistical Relational Learning and its Connections to Other Fields (Dietterich, T. and Getoor, L. and Murphy, K., eds.), pp. 110-116, 2004.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41239

- C. Vens, and H. Blockeel, *On heuristics for learning model trees*, Proceedings of the 15th Belgian-Dutch Conference on Artificial Intelligence (Heskes, T. and Lucas, P. and Vuurpijl, L. and Wiegerinck, W., eds.), pp. 307-314, 2003.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=40883

Not published or only as abstract

- C. Vens, J. Ramon, and H. Blockeel, *Refining aggregate conditions in relational learning*, 18th Belgium-Netherlands Conference on Artificial Intelligence, BNAIC 2006, Namur, Belgium, October 5-6, 2006.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42428

- C. Vens, and H. Blockeel, *Refining aggregate conditions in relational learning*, Joint APrIL/IQ Workshop, Titisee, Germany, March 15-18, 2006.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42181

- C. Vens, and H. Blockeel, *Refining aggregate conditions in relational learning*, 7th "Freiburg, Leuven and Friends" Workshop on Machine Learning, FLF-06, Titisee, Germany, March 13-14, 2006.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42167

- C. Vens, A. Van Assche, H. Blockeel, and S. Džeroski, *Aggregation and selection in relational data mining*, Dutch Belgian Database Day 2004, DBDBD 2004, Antwerpen, Belgium, December 3, 2004.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41531

- A. Van Assche, C. Vens, H. Blockeel, and S. Džeroski, *A random forest approach to relational learning*, 16h Belgian-Dutch Conference on Artificial Intelligence, BNAIC 2004, Groningen, The Netherlands, October 21-22, 2004.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41415

- C. Vens, A. Van Assche, H. Blockeel, and S. Džeroski, *Random forests for combining selection and aggregation in relational data mining*, 5th "Freiburg, Leuven and Friends" Workshop on Machine Learning, FLF-04, Hinterzarten, Germany, March 8-10, 2004.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41194

- H. Blockeel, and C. Vens, *On heuristics for learning model trees*, 4th "Freiburg, Leuven and Friends" Workshop on Machine Learning, FLF-03, Leuven/Dourbes, Belgium, March 19-21, 2003.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=40763

Contributions at other conferences

Published in proceedings

- J. Struyf, C. Vens, T. Croonenborghs, S. Džeroski, and H. Blockeel, *Napovedovanje funkcij genov z induktivnim logičnim programiranjem in drevesi za napovedno razvrščanje*, Prvo Srečanje Slovenskih Bioinformatikov (BIOINFO2005) (Anderluh, G. and Zupan, B. and Stare, J., eds.), pp. 27-30, 2005.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42015

Not published or only as abstract

- C. Vens, and H. Blockeel, *On heuristics for building model trees*, Belgian Data Mining Day, BDM 2005, Antwerpen, Belgium, May 27, 2005.
http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41721

Courses, with text

- C. Vens, and A. Van Assche, *Prediction*, The HIV Data Management and Data Mining Workshop, South African Medical Research Council, 491 Ridge Road, Durban, South-Africa, December 15, 2004, Molecular Virology and Bioinformatics Unit at Africa Centre for Health and Population Studies, 6h.

Biography

Celine Vens was born on July 12, 1980 in Menen (Belgium). After finishing high school at the Lyceum Onze-Lieve-Vrouw van Vlaanderen in Kortrijk in 1998, she started studying computer science at the Katholieke Universiteit Leuven, Campus Kortrijk. After two years, she came to Leuven to graduate cum laude as a Master in Informatics in July 2002. Her Master's thesis "Visualization of Genetic Algorithms" was supervised by Hendrik Blockeel.

In October 2002, she joined the Declarative Languages and Artificial Intelligence research group of the Katholieke Universiteit Leuven, starting a Ph.D. funded by a doctoral scholarship of the university. Her Ph.D. research, entitled "Complex Aggregates in Relational Learning", was supervised by Hendrik Blockeel and was defended on March 30, 2007.

Complexe aggregaten in relationeel leren

Nederlandse samenvatting

1 Inleiding

Situering van het onderzoek

Dit werk situeert zich in het vakgebied van **kunstmatige intelligentie** (McCarthy et al. 1955), de wetenschap die zich bezighoudt met het maken van intelligente machines, of meer specifiek, intelligente computerprogramma's. Het begrip *intelligent* is moeilijk te definiëren, wat zich uit in verschillende beschikbare definities. Deze lijken het toch eens te zijn dat één van de belangrijkste kenmerken van intelligent gedrag het vermogen tot leren is. Dit brengt ons bij het **automatisch leren**, een belangrijk deeldomein van kunstmatige intelligentie, dat algoritmen en technieken ontwikkelt die een automatisch systeem toelaten te leren. Van een automatisch systeem (bvb. een computerprogramma) wordt gezegd dat het *leert* als het beter wordt in de uitvoering van een taak door het vergaren van kennis opgedaan door ervaring in die taak (Langley 1996). Het onderzoeksdomein van automatisch leren kent vele subdomeinen met elk een eigen manier van leren of een eigen taak. Een groot deel van het onderzoek in dit domein houdt zich bezig met het leren van nieuwe kennis uit datasets. Zo'n dataset wordt meestal voorgesteld als een tabel, waarbij elke rij een voorbeeld (gegeven) voorstelt, en elke kolom een eigenschap (attribuut) van de gegevens. We onderscheiden twee toepassingen. Bij predictief leren is de taak het voorspellen van de waarde van een specifiek attribuut. Bij descriptief leren is de taak het vinden van een algemeen geldende beschrijving van de voorbeelden. In dit onderzoek concentreren we ons vooral op predictief leren.

Het domein van automatisch leren kent vele raakpunten met **datamining**. Bij datamining zoekt men naar regelmatigheden (patronen genoemd) in grote gegevensbanken. Dit is een stap in het bredere gebied van **kennisvergaring uit gegevensbanken**, wat ook het voorbereiden van de gegevens en het verwerken van de resultaten omvat.

Door de vooruitgang in computertechnologie is het genereren en opslaan van gegevens de laatste jaren erg gemakkelijk en goedkoop geworden, wat resulteert in de beschikbaarheid van enorme hoeveelheden gegevens uit verschillende gebieden (bvb. financiewezen, bio- en chemo-informatica, marketing,...). Zulke gegevens worden praktisch altijd opgeslagen in relationele gegevensbanken (Elmasri and Navathe 2004), die meerdere gerelateerde tabellen kunnen omvatten. Dit conflicteert met de klassieke leermethoden die alle gegevens in één tabel veronderstellen. Een aantal zogenaamde propositionalizatiemethoden zijn ontwikkeld die een relationele gegevensbank transformeren tot één enkele tabel, zodat klassieke algoritmen kunnen uitgevoerd worden. De laatste jaren wordt echter meer en meer onderzoek verricht naar methoden die onmiddellijk uit een relationele gegevensbank leren. Het bijhorende onderzoeksgebied wordt **relationeel leren** of **relationele datamining** genoemd.

Een belangrijk deel van het onderzoek binnen relationeel leren is **inductief logisch programmeren** (ILP) (Muggleton 1992; Lavrač and Džeroski 1994), een domein dat zich situeert op de doorsnede tussen automatisch leren en logisch programmeren. In ILP worden de gegevens alsook de patronen voorgesteld als logische programma's (Lloyd 1987; Bratko 2001). Dit maakt het relatief eenvoudig om achtergrondkennis uit te drukken en geeft een expressieve, makkelijk begrijpbare, en theoretisch gefundeerde basis aan het domein.

Doel en motivatie

Bij relationeel leren of ILP leert men patronen uit relationele gegevensbanken, die normaalgezien uit meerdere tabellen bestaan, die met elkaar gerelateerd zijn. Deze relaties kunnen bijvoorbeeld een één-op-veel of veel-op-veel cardinaliteitsverhouding hebben. Een voorbeeld waarvoor een predictie moet gemaakt worden kan dus gerelateerd zijn aan een verzameling objecten die mogelijk relevant zijn voor de predictie. Bestaande relationele leermethoden behandelen deze verzamelingen gewoonlijk op één van volgende manieren: door het opleggen van condities aan de elementen in de verzameling of door het gebruiken van aggregaatsfuncties om ze samen te vatten.

Bestaande methoden zijn niet in staat om beide benaderingen te combineren, waardoor ze bepaalde patronen niet kunnen leren (Blockeel and Bruynooghe 2003). Een dergelijke combinatie zou bestaan uit het aggregeren over een deelverzameling van elementen die aan een specifieke conditie voldoen ("aggregeren over een selectie"). Bijvoorbeeld, in een gegevensbank die informatie over personen en hun bankrekeningen bijhoudt zou de predictie van een persoon kunnen gerelateerd zijn aan de *som* (aggregatie) van het saldo op zijn *spaar*rekeningen (selectie). Hoewel van dergelijke combinaties verwacht kan worden dat ze vaak relevant zijn bij de predictie zijn ze moeilijk om automatisch te leren, ten eerste omdat de zoekruimte van mogelijke patronen significant groeit en ten tweede omdat het moeilijker wordt om deze zoekruimte op een gestructureerde en efficiënte manier te doorlopen. Toch geven een aantal auteurs het belang van deze combinatie aan (Perlich and Provost 2003; Blockeel and Bruynooghe 2003; Krogel et al. 2003). Het belangrijkste doel in dit eindwerk is het combineren van beide benaderingen, dus het aggregeren over selecties.

2 Inductief logisch programmeren

Zoals gezegd in de inleiding worden de gegevens en geleerde patronen bij inductief logisch programmeren (ILP) voorgesteld als logische programma's, een belangrijk onderdeel van de eerste orde predicatenlogica. ILP methoden worden daarom ook eerste-orde methoden genoemd. Een gegeven wordt voorgesteld door een feit, een patroon door een verzameling regels.

Voorbeeld 1 *Beschouw een gegevensbank die informatie bijhoudt over personen en hun bankrekeningen. Voor elke bankrekening worden ook de bijhorende transacties bijgehouden. Een persoon wordt gekenmerkt door een identificator*

PERSOON

PersId	**Leeftijd**	**Doel**
john	24	positief
mary	27	negatief
billy	25	negatief

REKENING

PersId	**RekId**	**Type**	**Saldo**
john	123456	zicht	100
john	987654	zicht	200
john	789123	spaar	200
mary	345678	spaar	250
billy	456789	zicht	150

TRANSACTIE

RekId	**TransId**	**Datum**	**Type**	**Bedrag**
123456	tr090	010706	afhaling	50
123456	tr091	030706	storting	30
987654	tr098	120706	storting	70
789123	tr100	150706	afhaling	100
789123	tr101	150706	storting	80
345678	tr150	220706	afhaling	20

Figuur 1: Extensie van de *Rekening* gegevensbank.

(bvb. naam of rijksregisternummer), zijn leeftijd en een niet nader gespecificeerd doelattribuut. Een extensie van deze gegevensbank wordt getoond in Fig. 1. Bij het gebruik van ILP methoden wordt deze gegevensbank eerst omgezet naar een eerste orde voorstelling (Tabel 1). Een mogelijk patroon dat de positieve voorbeelden van de negatieve onderscheidt is het volgende:

$$persoon(PId, Leeft, positief) \longleftarrow Leeft < 26, rekening(PId, RekId, spaar, Saldo), Saldo > 180, !.$$
$$persoon(PId, Leeft, negatief) \longleftarrow .$$

Dit patroon zegt dat een persoon positief is als zijn leeftijd kleiner is dan 26 en als hij een spaarrekening heeft met saldo groter dan 180. In alle andere gevallen wordt een negatieve voorspelling gegeven.

Patronen (ook wel hypothesen genoemd) worden normaalgezien regel per regel geleerd. Aan de ruimte van mogelijk regels wordt een orderelatie opgelegd die toelaat om deze efficiënt te doorzoeken. Deze orderelatie is meestal gebaseerd op θ-subsumptie (Plotkin 1969).

Definitie 1 (θ-subsumptie) *Een regel r_1* θ-subsumeert *een regel r_2 (genoteerd als $r_1 \leq_\theta r_2$) als er een substitutie θ bestaat, zodat $r_1\theta \subseteq r_2$.*

Een substitutie $\sigma = \{V_1/t_1, V_2/t_2, ..., V_n/t_n\}$ vervangt gelijktijdig alle variabelen $V_1, V_2, ..., V_n$ door de termen (dit zijn constanten, variabelen of functies)

Tabel 1: Eerste orde voorstelling van de *Rekening* gegevensbank extensie van Fig. 1.

```
persoon(john,24,positief).
persoon(mary,27,negatief).
persoon(billy,25,negatief).

rekening(john,123456,zicht,100).
rekening(john,987654,zicht,200).
rekening(john,789123,spaar,200).
rekening(mary,345678,spaar,250).
rekening(billy,456789,zicht,150).

transaction(123456,tr090,010706,afhaling,50).
transaction(123456,tr091,030706,storting,30).
transaction(987654,tr098,120706,storting,70).
transaction(789123,tr100,150706,afhaling,100).
transaction(789123,tr101,150706,storting,80).
transaction(345678,tr150,220706,afhaling,20).
```

$t_1, t_2, .., t_n$.

Voorbeeld 2 *Regel* R_1 :

$$persoon(PersId, Leeft, Doel) \longleftarrow rekening(PersId, RekId, Type, Saldo)$$

θ-subsumeert regel R_2 :

$$persoon(PersId, Leeft, Doel) \longleftarrow rekening(PersId, RekId, spaar, Saldo)$$

d. m. v. de substitutie $\theta = \{Type/spaar\}$. Regel R_1 θ-subsumeert ook regel R_3 :

$$persoon(PersId, Leeft, Doel) \longleftarrow rekening(PersId, RekId, Type, Saldo), \quad Saldo > 500.$$

In dit laatste voorbeeld is $\theta = \emptyset$.

De θ-subsumptie relatie heeft een interessante eigenschap m.b.t. algemeenheid van regels: als $R_1 \leq_\theta R_2$, dan is R_1 minstens zo algemeen als R_2, waarbij een regel algemener gesteld wordt dan een andere als hij meer voorbeelden dekt. Deze algemeenheid is belangrijk om te kunnen snoeien in de hypotheseruimte: als een bepaalde hypothese H te weinig voorbeelden dekt, dan hoeven de hypothesen die gesubsumeerd worden door H niet meer onderzocht te worden. Een regel wordt geconstrueerd startende van de meest algemene regel ($waar \longleftarrow$) en wordt stap voor stap opgebouwd, gebruik makende van een verfijningsoperator gebaseerd op θ-subsumptie. Zo'n verfijningsoperator verfijnt een regel op een van volgende manieren:

- het uitvoeren van een substitutie op de regel, of
- het toevoegen van een taalelement aan de regel.

De taalelementen die kunnen toegevoegd worden (zoals `Saldo` $>$ `500` in Vb. 2) worden door de gebruiker gespecificeerd in een hypothesetaalbeschrijving.

3 Combineren van aggregaten en selecties

In deze sectie bekijken we hoe de combinatie van aggregaten en selecties bereikt wordt in de ILP context.

Complexe aggregaatscondities

De regels waarin we geïnteresseerd zijn, zijn niet van pure logische aard: ze kunnen aggregaatscondities bevatten, die we als volgt definiëren.

Definitie 2 (Aggregaatsconditie, syntactisch) *Een aggregaatsconditie is een literalpaar* $(F(V, Q, R), between(R, T_1, T_2))$*, waarbij* F *een aggregaatsfunctie is,* V *een variabele die voorkomt in de* aggregaatsquery Q*,* R *het resultaat van het toepassen van* F *op de (multi-)verzameling van alle antwoordsubstituties voor* V *waarin* Q *resulteert,* T_1 *and* T_2 *grenswaarden, en het between/3 predicaat nagaat of* $R \in [T_1, T_2]$*. De literal* $F(V, Q, R)$ *wordt de aggregaatliteral genoemd.*

De aggregaatsfuncties die we beschouwen zijn deze die resulteren in een numerieke waarde, zoals *max*, *min*, *avg* (gemiddelde), *sum* (som) en *count* (cardinaliteit). Vaak wordt als interval $]-\infty, T_2]$ of $[T_1, \infty[$ gebruikt en wordt het *between* predicaat vervangen door de overeenkomstige ongelijkheid. De voorgaande definitie van aggregaatscondities is louter syntactisch. De volgende definitie geeft de betekenis ervan weer.

Definitie 3 (Aggregaatsconditie, semantisch) *Een aggregaatsconditie is een functie* $c : \mathbb{F} \times \mathbb{S} \times \mathbb{I} \to \mathbb{B} : F(S) \in I \mapsto B$*, met* $\mathbb{F}$ *een verzameling aggregaatsfuncties,* $\mathbb{S}$ *een verzameling multi-verzamelingen,* $\mathbb{I}$ *een verzameling numerieke intervallen en* $\mathbb{B}$ *de verzameling booleaanse waarden. De waarde voor* B *is waar als de conditie* $F(S) \in I$ *voldaan is, en vals in het andere geval.*

In deze definitie wordt abstractie gemaakt van het feit dat S gegenereerd is door een query Q en een variabele V.

In de vorige sectie legden we uit dat de meeste ILP systemen gebruik maken van een algemeen-naar-specifiek orde in de zoekruimte van mogelijke regels om deze zoekruimte efficiënt te doorzoeken. De verfijningsoperator die daarbij gebruikt wordt verfijnt de reeds bekomen regel door het toevoegen van een taalelement of door het uitvoeren van een substitutie. Stel nu dat de reeds bekomen regel een aggregaatsconditie bevat, bijvoorbeeld

$persoon(PersId, positief) \leftarrow$
$\quad max(Saldo, rekening(PersId, RekId, Type, Saldo), M), M \geq 50000.$

Deze regel classificeert een persoon als positief indien het maximale saldo van zijn rekeningen groter is dan 50000. Dergelijke regel kan niet enkel verfijnd worden door toepassing van de verfijningsoperator op de regel zelf, maar ook door toepassing van de verfijningsoperator op de aggregaatsquery $rekening(PersId, RekId, Type, Saldo)$, resulterend in, bijvoorbeeld

$persoon(PersId, positief) \leftarrow$
$\quad max(Saldo, rekening(PersId, RekId, spaar, Saldo), M), M \geq 50000,$

waarbij *Type* gesubstitueerd werd door *spaarrekening*. Naast een substitutie kan ook een taalelement toegevoegd worden aan de aggregaatsquery. Dergelijke verfijningen resulteren in wat we noemen *complexe aggregaten*, die een combinatie vormen van aggregaten en selectiecondities.

Definitie 4 (Complexe en simpele aggregaatscondities) *Een complexe aggregaatsconditie is een aggregaatsconditie waar de aggregaatsquery Q een complexe query is, m.a.w. Q is verfijnd met selectiecondities.*
Een simpele aggregaatsconditie is een aggregaatsconditie waar de aggregaatsquery niet verfijnd is.

Verfijnen van regels met complexe aggregaatscondities

Om regels met aggregaatscondities op dezelfde manier te kunnen verfijnen als standaard regels, definiëren we een extensie van de klassieke θ-subsumptie relatie voor regels met aggregaten. We noemen de resulterende relatie α-subsumptie. In de volgende definitie stelt $S(C)$ het standaard deel van een regel voor; dit is de verzameling literals die geen deel uitmaken van een aggregaatsconditie.

Definitie 5 (α-subsumptie) *Een regel c_1 α-subsumeert een regel c_2 (genoteerd als $c_1 \leq_\alpha c_2$) als en slechts als $\exists\theta : S(c_1)\theta \subseteq S(c_2)$, en voor elke aggregaatliteral $F(V_1, Q_1, R_1) \in c_1$, is er een aggregaatliteral $F(V_2, Q_2, R_2) \in c_2$ zodat $R_1\theta = R_2$, $Q_1\theta \leq_\alpha Q_2$, en deze laatste α-subsumptie heeft enkel betrekking op een verzameling substituties σ over lokaal gedefinieerde variabelen in Q_1, zodat $V_1\sigma = V_2$.*

Een verfijningsoperator gebaseerd op α-subsumptie verfijnt een regel door het uitvoeren van een van volgende basisoperaties:

- het uitvoeren van een substitutie op de regel,
- het toevoegen van een taalelement aan de regel, of
- het uitvoeren van een substitutie op of het toevoegen van een taalelement aan de aggregaatsquery van een aggregaatliteral in de regel.

Het verfijnen van regels gebruik makende van α-substitutie brengt een aantal moeilijkheden met zich mee. Ten eerste wordt de zoekruimte significant uitgebreid. Ten tweede is er, hoewel de definitie van α-substitutie syntactisch erg gelijkaardig is aan die van θ-subsumptie, een belangrijk verschil tussen beide: de eigenschap dat $c_1 \leq_\alpha c_2$ impliceert dat c_2 meer specifiek is dan c_1 geldt niet meer. Dit kan te wijten zijn aan de semantiek van de aggregaatsfunctie of aan de monotoniciteitseigenschappen van de aggregaatsconditie. We bespreken elk van de problemen achtereenvolgens.

Uitbreiding van de zoekruimte

Door aggregaatsqueries te verfijnen wordt het aantal mogelijke verfijningen voor een regel drastisch verhoogd. Veronderstel dat een standaard regel op C manieren verfijnd kan worden. Als we nu ook aggregaatliterals toelaten met elk van deze C verfijningen als aggregaatsquery, elk van de V variabelen in deze verfijningen toelaten als aggregaatsvariabele en N mogelijke aggregaatsfuncties beschouwen, dan wordt het aantal verfijningen verhoogd tot $C + V \cdot N \cdot C + V \cdot N \cdot C \cdot (C + V \cdot N \cdot C)^L$ met L het aantal taalelementen toegevoegd in de aggregaatsquery.

Semantiek van de aggregaatsfuncties

Zoals gezegd resulteert het verfijnen van een aggregaatsquery niet automatisch in een specializatie van de verfijnde regel. Het kan ook resulteren in een generalizatie, of sterker nog, in geen van beide. Het resultaat hangt af van hoe de (multi-)verzameling van antwoordsubstituties voor de aggregaatsvariabele berekend wordt. We stellen in de tekst drie semantieken voor. Voor de aggregaatliteral

$$\begin{aligned}&som(Saldo, (rekening(PersId, RekId, Type, Saldo),\\ &\qquad transactie(RekId, TransId, TransType, Bedrag)), Resultaat)\end{aligned}$$

leidt dit tot volgende mogelijke resultaten voor het eerste voorbeeld (john) met de gegevens uit Tabel 1:

- $100+100+200+200+200 = 800$ (bij aggregatie over de multi-verzameling van saldo's)
- $100 + 200 = 300$ (bij aggregatie over de verzameling van saldo's)
- $100+200+200 = 500$ (bij aggregatie over de verzameling van rekeningen).

Bij de eerste semantiek kan het verfijnen van een aggregaatsquery zowel een specializatie als een generalizatie (of zelfs geen van beide) van de regel tot gevolg hebben, doordat de multi-verzameling van antwoordsubstituties zowel kan verkleinen als uitbreiden. Bij de tweede en derde semantiek wordt deze laatste gegarandeerd verkleind. De derde semantiek geeft het meest intuïtieve resultaat van de aggregaatliteral (som van de saldo's van de rekeningen met een bijhorende transactie).

Monotoniciteit

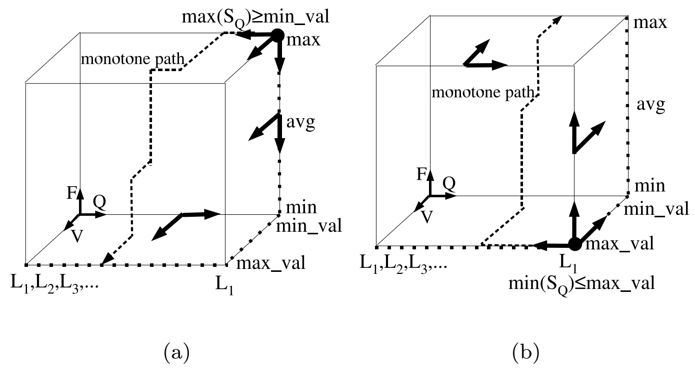
Zelfs indien we gebruik maken van de derde semantiek uit vorige paragraaf, kan een verfijning van een aggregaatsconditie er nog steeds voor zorgen dat we een generalizatie van de regel bekomen in plaats van een specializatie. Dit is te wijten aan de monotoniciteitseigenschappen van de aggregaatsconditie. In het algemeen noemen we een functie $f(x_1, ..., x_n)$ die een geordend domein op een geordend bereik afbeeldt, monotoon in x_i als een stijging of daling in x_i de orde bewaart, anti-monotoon in x_i als de orde omgekeerd wordt, en niet-monotoon in x_i in de andere gevallen. Om de monotoniciteitseigenschappen van een aggregaatsconditie $F(S) \in I \mapsto B$ te onderzoeken, moeten we dus eerst een orde definiëren op elk van de domeinen $\mathbb{F}, \mathbb{S}, \mathbb{I}$ en $\mathbb{B}$. Waar dit voor $\mathbb{S}, \mathbb{I}$ en $\mathbb{B}$ triviaal is, is het moeilijker een orde te definiëren voor $\mathbb{F}$. In de tekst stellen we verschillende klassen van aggregaatsfuncties voor, waarbij binnen elke klasse een orde geldt, en leiden daarna de monotoniciteitseigenschappen van een aggregaatsconditie af.

Er bestaat geen verfijningsstrategie voor aggregaatsqueries die garandeert dat een specializatie verkregen wordt, tenzij we ons beperken tot aggregaatscondities die monotoon zijn in de aggregaatsquery en de derde semantiek gebruiken. Als we echter, naast verfijningen in de aggregaatsquery, ook verfijningen in de aggregaatsfunctie en in het interval toelaten, kunnen we wel zo'n verfijningsstrategie definiëren. Voor elke geordende klasse van aggregaatsfuncties worden de mogelijke verfijningen visueel weergegeven in een verfijningskubus. Een voorbeeld voor de zogenaamde klasse van veralgemeende gemiddelden wordt gegeven in Fig. 2. De achterliggende idee is dat, startende van één enkele aggregaatsconditie, de volledige kubus gegenereerd kan worden door het volgen van monotone paden die aangegeven worden door de pijlen in de kubus. Langs deze monotone paden worden alsmaar specifiekere aggregaatscondities gegenereerd.

Geen van de bestaande leermethoden beschouwt verfijningen langs deze drie dimensies tesamen. Toch toont onze analyse aan dat alle dimensies nodig zijn om de volledige zoekruimte op te bouwen via de monotone paden.

Ook bij de verfijningskubussen hoort een subsumptie relatie, we noemen ze μ-subsumptie.

Definitie 6 (μ-subsumptie) *Een regel c_1 μ-subsumeert een regel c_2 (genoteerd als $c_1 \leq_\mu c_2$) als en slechts als $S(c_1) \leq_\theta S(c_2)$, en voor elke aggregaatsconditie $(F_1(V_1, Q_1, R_1), between(R_1, T_{11}, T_{12})) \in c_1$ bestaat er een aggregaatsconditie $(F_2(V_2, Q_2, R_2), between(R_2, T_{21}, T_{22})) \in c_2$ zodat $R_1\theta = R_2$, het tupel $(F_2, Q_2, T_{21}, T_{22})$ verkregen werd uit $(F_1, Q_1\sigma, T_{11}, T_{12})$ via een monotoon pad in de corresponderende verfijningskubus, $V_1\sigma = V_2$, en de substitutie σ bestaat uit θ uitgebreid met een substitutie over lokaal gedefinieerde variabelen in Q_1.*



Figuur 2: Verfijningskubussen voor de veralgemeende gemiddelden. (a) Verfijningskubus voor de startconditie $max(S_Q) \geq min\_value$. (b) Verfijningskubus voor de startconditie $min(S_Q) \leq max\_value$.

Deze relatie heeft, net zoals θ-subsumptie, maar in tegenstelling tot α-subsumptie, de eigenschap dat c_1 μ-subsumeert c_2 impliceert dat c_1 minstens zo algemeen is als c_2.

4 Complexe aggregaten bij eerste orde beslissingsbomen

In vorige sectie legden we de basis voor het gebruik van complexe aggregaten in het ILP raamwerk. Nu implementeren en testen we de ontwikkelde concepten en technieken in een concreet ILP systeem, nl. Tilde (Blockeel and De Raedt 1998).

Tilde: een eerste orde beslissingsboom algoritme

Tilde is een eerste orde beslissingsboom leeralgoritme. Beslissingsbomen zijn boomvormige patronen die de voorbeelden in een dataset recursief opsplitsen volgens hun uitkomst op een test. Elk voorbeeld komt op die manier terecht in een blad van de boom, waar zich de predictie voor dat voorbeeld bevindt. Als een nominaal attribuut voorspeld wordt, spreken we van een classificatieboom; bij een numeriek attribuut van een regressieboom. Tilde leert bomen van boven naar beneden, te starten bij de wortelknoop en te eindigen in de bladeren. De testen in de knopen zijn eerste orde literals. Bij elke knoop hoort ook een query die de conjunctie is van alle literals op het pad van de wortelknoop tot de knoop in kwestie die slagen; deze query wordt de huidige query genoemd. Bij het zoeken naar een test voor een knoop K wordt de huidige query in K uitgebreid met een taalelement uit de hypothesetaalbeschrijving. Een verfijningsoperator gebaseerd op θ-subsumptie genereert deze uitbreidingen, waarna ze getest worden op de voorbeelden horende bij K en de beste genomen wordt.

Toevoegen van aggregaatscondities bij Tilde

Om simpele aggregaatscondities toe te voegen aan de testen die TILDE kan leren, moeten deze toegevoegd worden aan de hypothesetaalbeschrijving. Voor het toevoegen van complexe aggregaatscondities in een knoop zijn er twee mogelijkheden:

- het verfijnen van de aggregaatsquery van een (simpele of complexe) aggregaatsconditie die voorkomt in de huidige query van die knoop, of
- onmiddellijk een complexe aggregaatsconditie toevoegen, onafhankelijk van de aggregaatscondities die reeds voorkomen in de huidige query.

De tweede manier maakt gebruik van een vooruitkijktechniek (Blockeel and De Raedt 1997) die vaak gebruikt wordt binnen ILP.

Verfijnen van aggregaatscondities bij Tilde

Aangezien bij TILDE verfijningen van een query Q bestaan uit een uitbreiding van Q met een of meerdere literals R, zijn de voorbeelden waarvoor de verfijning (Q, R) (de conjunctie van Q en R) slaagt een deelverzameling van de voorbeelden waarvoor Q slaagt. Dit betekent (per definitie) dat (Q, R) een specializatie is van Q. Een gevolg hiervan is dat een eenvoudige verfijningsoperator gebaseerd op α-subsumptie gebruikt kan worden voor het verfijnen van aggregaatscondities. Dergelijke verfijningsoperator is echter weinig efficiënt, vooral bij de vooruitkijkmethode, en daarom onderzoeken we enkele technieken om de snelheid van het algoritme te verhogen.

Een eerste techniek is het gebruik van gerandomiseerde bossen (Breiman 2001). Een gerandomiseerd bos is een verzameling beslissingsbomen, waarbij elke boom gebouwd wordt door bij elke knoop slechts een willekeurige steekproef van de verfijningen te testen bij het zoeken naar de beste test. Dit compenseert voor de groei van de zoekruimte bij complexe aggregaten. Bovendien hebben gerandomiseerde bossen bij propositioneel leren aangetoond de predictieve performantie te verhogen indien de verzameling testen heel groot is en een voldoende aantal bomen gebouwd worden.

Een tweede techniek (die chronologisch na het gerandomiseerd bos ontwikkeld werd) is het gebruik van een verfijningsoperator gebaseerd op de voorgestelde μ-subsumptie relatie. Deze kunnen we bij de methode die gebruik maakt van vooruitkijken als volgt toepassen. We schikken de verfijningen teruggegeven door de verfijningsoperator op die manier dat ze de monotone paden uit de verfijningskubussen reflecteren. Bij het zoeken naar de beste test worden alle verfijningen op alle voorbeelden uitgevoerd. Als blijkt dat een bepaalde verfijning V faalt voor een bepaald voorbeeld E, dan kunnen we voor E alle verfijningen die verkregen worden uit V via een monotoon pad, uit de zoekruimte snoeien, wat logischerwijze een snelheidswinst oplevert. Deze techniek heeft als voordeel over de gerandomiseerde bossen dat de volledige zoekruimte doorzocht wordt en dat er slechts één boom gebouwd wordt, wat de interpretatie van het model ten goede komt.

Experimenten

Het leren van complexe aggregaten en de zopas voorgestelde technieken om de efficiëntie te verbeteren worden geëvalueerd door de uitgebreide versie van Tilde te evalueren op een aantal ILP datasets. Onze experimentele evaluatie toont aan dat het gebruik van aggregaten een duidelijke winst in predictieve performantie oplevert. Het grootste deel van deze winst is wordt verkregen met simpele aggregaatscondities. De stap naar complexe aggregaten levert enkel een kleine bijkomstige winst op bij onze drie ILP benchmark datasets. Het gebruik van een artificiële dataset toont echter aan dat, indien het doelconcept complexe aggregaten bevat, het gebruik van complexe aggregaten een duidelijke winst biedt ten opzichte van simpele aggregaten. Het gebruik van de verfijningsoperator gebaseerd op μ-subsumptie levert een snelheidswinst tot factor 3 op bij de benchmark datasets. Het systeem uitbreiden tot een eerste orde gerandomiseerd bos leersysteem levert, naargelang het aantal bomen en de steekproefverhouding die gebruikt worden, een snelheidswinst of -verlies op. De predictieve performantie wordt bij gerandomiseerde bossen altijd verbeterd.

5 Predicties met complexe aggregaten

In de vorige secties beschouwden we het gebruik van complexe aggregaten in het antecedent van de hypothese (bvb. in de knopen van een beslissingsboom). We kunnen echter ook bij het consequent (de predictie, bvb. in de bladeren van een beslissingsboom) aggregaten gebruiken. Neem bijvoorbeeld modelbomen (Karalic 1992; Quinlan 1992). Dit zijn regressiebomen waarbij de bladeren een model bevatten in plaats van een constante waarde. Dit model voorspelt het numerieke doelattribuut (meestal) als een lineaire combinatie van een aantal invoerattributen. Als deze invoerattributen niet-deterministisch zijn (d.i. er zijn meerdere waarden voor deze attributen voor elk voorbeeld, door één-op-veel of veel-op-veel relaties in de dataset), moet er opnieuw een keuze gemaakt worden hoe deze verzamelingen van waarden behandeld worden. Bestaande leeralgoritmen gaan ook hier ofwel de waarde nemen van de objecten die aan een specifieke conditie voldoen ofwel aggregaatsfuncties gebruiken om de waarden samen te vatten, zonder beide benaderingen te combineren. In deze sectie stellen we het gebruik van complexe aggregaten voor bij het construeren van lineaire vergelijkingen in de bladeren van modelbomen.

Zoeken naar een gepaste heuristiek

De complexe aggregaten in de vergelijkingen in de bladeren worden niet van nul geleerd, maar worden ingevoerd indien ze een lineair verband met het doelattribuut vertoond hebben bij het bouwen van de boom. Dit betekent dat de heuristiekfunctie (d.i. de functie die de kwaliteit aangeeft voor elke mogelijke test voor een knoop) lineaire verbanden in rekening moet brengen. Bestaande heuristieken voor modelbomen die dit doen zijn minstens kwadratisch in het aantal attributen (Karalic 1992; Alexander and Grimshaw 1996; Malerba et al. 2004). Aangezien we complexe aggregaten willen gebruiken is het aantal attributen bij ons zeer groot, waardoor we op zoek gaan naar een efficiëntere heuristiek. Daar het merendeel van het onderzoek naar modelbomen in de propositionele context gebeurd is, bestuderen we eerst de in de literatuur voorgestelde heuristieken in die context.

De meest efficiënte heuristiek is de variantiegebaseerde heuristiek (Quinlan 1992; Wang and Witten 1997). Daarbij wordt de variantie (of een variant daarvan, zoals standaard deviatie of som van de kwadratische fout) in linker- en rechterkind van de te splitsen knoop geminimaliseerd. Variantiegebaseerde heuristieken zijn echter geen geschikte aanpak voor het leren van modelbomen, aangezien de variantie onafhankelijk is van de kwaliteit van een lineair model (Karalic 1992; Malerba et al. 2004). Onze analyse toont aan dat niet zozeer de predictieve performantie, maar wel de verklarende kwaliteit van de modelbomen beïnvloed wordt: onnodig grote modelbomen met niet-informatieve splitcondities worden geïnduceerd.

De meest performante heuristiek is het minimaliseren van de residuele variantie, waarbij de variantie genomen wordt ten opzichte van een meervoudig lineair model (in alle predictieve attributen) (Karalic 1992). Deze aanpak is echter kubisch in het aantal attributen en dus niet geschikt voor onze toepassing.

De heuristiek die we voorstellen combineert de voordelen van beide genoemde benaderingen. Ze houdt rekening met lineaire modellen in die zin dat de residuele variantie genomen wordt ten opzichte van een enkelvoudig lineair model (met als predictief attribuut het attribuut waarop gesplitst wordt). De bijhorende complexiteit is lineair in het aantal attributen en verschilt slechts een constante factor van de variantiegebaseerde methoden.

We implementeren de voorgestelde heuristiek in een modelboom leeralgoritme, Mauve genaamd. Alle numerieke attributen die voorkomen in de dataset kunnen gebruikt worden als predictieve attributen in de lineaire modellen in de bladeren van de modelbomen. Experimentele evaluatie toont aan dat Mauve kortere bomen met eenzelfde predictieve performantie als variantiegebaseerde technieken leert, indien er een stuksgewijs lineair verband is tussen de invoerattributen en het doelattribuut. Bovendien suggereren de resultaten dat dergelijke verbanden vaak voorkomen in echte datasets en dat het gebruik van meer ingewikkelde technieken in het algemeen niet leidt tot betere predicties of kortere modelbomen.

Relationele modelbomen met complexe aggregaatscondities

We breiden het Mauve algoritme uit naar de relationele context. Daarvoor opwaarderen we Tilde tot een eerste orde modelboom leeralgoritme. Tilde's heuristiek voor het leren van regressiebomen, die een variantiegebaseerde heuristiek is, wordt aangepast naar de ontwikkelde heuristiek bij Mauve. De lineaire vergelijking in een blad bevat de attributen (waaronder eventueel complexe aggregaten) die voorkomen in de knopen op het pad van de wortelknoop tot het blad. Het resulterende algoritme noemen we ReMauve.

Om globale lineaire verbanden in de data te modelleren voorzien we een speciale soort knopen: regressieknopen (Appice et al. 2003). In tegenstelling tot de splitknopen gaan regressieknopen de data niet splitsen maar geven ze de volledige verzameling bijhorende voorbeelden door naar hun enige kindknoop. De bedoeling van regressieknopen is het introduceren van een predictief attribuut dat kan gebruikt worden in de lineaire vergelijkingen in de bladeren. Om sterk gecorreleerde testen te vermijden, wordt, na het invoegen van een regressieknoop of een splitknoop met een numerieke test, het lineaire effect van de test afgetrokken van het doelattribuut.

Experimenten tonen aan dat ReMauve, bij datasets met veel numerieke attributen (zoals het geval is bij het leren van complexe aggregaten), beter scoort dan Tilde op gebied van predictieve performantie en modelgrootte. Als we de mogelijkheid om (complexe) aggregaten te leren afzetten, dan verschilt de predictieve performantie van ReMauve, in vergelijking met een ander systeem dat relationele modelbomen bouwt (Appice et al. 2003), niet significant, maar worden in het algemeen wel kortere bomen gebouwd. De mogelijkheid om complexe aggregaten te leren laat in sommige gevallen toe om de predictieve performantie te verbeteren.

6 Toepassingen

We stellen twee toepassingen van complexe aggregaten voor.

Een landbouwtoepassing

In de eerste toepassing analyseren we een dataset over genetisch gemanipuleerde landbouwproducten. De mogelijkheid dat genetisch gemanipuleerde gewassen gemengd worden met organische gewassen (bvb. door stuifmeel meegedragen door de wind) is een delicate aangelegenheid en daarom is de detectie van gemanipuleerde gewassen in velden met organische gewassen een uitdaging. In onze toepassing concentreren we op koolzaad gewassen.

Elk voorbeeld in de onderzochte dataset is een veldplan van 35 velden, met voor elk veld informatie over de cultiveringstechnieken van de laatste vier jaar (geteelde gewas, zaaidatum,...). De taak is het voorspellen van de graad van contaminatie met gemanipuleerd materiaal in de koolzaadgewassen van het middelste veld in het huidige jaar. De specifieke bedoeling van de toepassing is om na te gaan of naburige velden een invloed hebben op deze voorspelling en of complexe aggregaten (bvb. de gemiddelde zaaidatum van de velden binnen een afstand van 200m was vóór 1 juni) een voordeel bieden ten opzichte van testen op aparte velden (bvb er bestaat een veld binnen een afstand van 200m met zaaidatum voor 1 juni).

Resultaten tonen aan dat eigenschappen van naburige velden (zowel geaggregeerd of niet) inderdaad een invloed hebben op de predictie, maar dat ze de predictieve performantie niet verhogen. Verder onderzoek met meer realistische datasets is nodig.

Een biologische toepassing

Bij de tweede toepassing analyseren we een biologische dataset die de genen in het genoom van gist bevat. Bedoeling is om de functie van de genen te voorspellen aan de hand van informatie over gelijksoortige genen en secondaire structuren. Aangezien een gen meerdere functies heeft, die gestructureerd zijn in een hiërarchie, hebben we te maken met een zogenaamd hiërarchisch meervoudig classificatie probleem (Blockeel et al. 2002).

In deze toepassing gebruiken we het CLUS algoritme, een propositionele versie van TILDE en speciaal aangepast voor hiërarchische meervoudige classificatie toepassingen (Struyf et al. 2005). Dit betekent dat we eerst de relationele data moeten omzetten naar een propositionele tabel. Hiervoor gebruiken we WARMR (Dehaspe and Toivonen 1999), een relationeel frequent patroon leeralgoritme. WARMR gebruikt dezelfde verfijningsoperator als TILDE en dus kunnen ook complexe aggregaten geleerd worden. De invoer voor CLUS bestaat dus uit één tabel met als kolommen de frequente patronen (die complexe aggregaatscondities kunnen bevatten) die geleerd zijn met WARMR. De waarde voor elk voorbeeld is *waar* als het patroon voorkomt in dat voorbeeld, en *vals* anders.

De resulterende boom bevat bijna uitsluitend testen gerelateerd aan de gelijksoortige genen, met een complex aggregaat in de wortelknoop.

7 Besluit

De belangrijkste bijdragen uit dit eindwerk kunnen als volgt samengevat worden.

- Het introduceren van complexe aggregaatscondities, dewelke een combinatie van aggregaten en selectiecondities vormen, in het ILP raamwerk. De klassieke ILP verfijningsoperator is uitgebreid om complexe aggregaten te kunnen leren. De belangrijkste problemen die hierbij opduiken zijn de groei van de hypotheseruimte en het schenden van de algemeen-naar-specifiek orde in de hypotheseruimte die verondersteld wordt door de verfijningsoperator. Het laatste probleem kan gerelateerd zijn aan de semantiek van de aggregaatsfunctie of aan de monotoniciteitseigenschappen van de aggregaatsconditie.

- Het opstellen van een algemeen verfijningsraamwerk voor complexe aggregaatscondities dat de algemeen-naar-specifiek orde in de hypotheseruimte bewaart. Dit raamwerk is toepasbaar op elk relationeel leersysteem dat complexe aggregaten leert.

- Het uitbreiden van een bestaand eerste orde beslissingsboom leeralgoritme om complexe aggregaten te kunnen leren. Complexe aggregaten worden bekomen door het verfijnen van aggregaten die hoger in de boom voorkomen of door toepassing van een vooruitkijktechniek.

- Het opwaarderen van het beslissingsboom algoritme naar een eerste orde gerandomiseerd bos leeralgoritme. Een verzameling beslissingsbomen wordt geleerd waarbij elke knoop gebouwd wordt door een willekeurige steekproef van mogelijke testen te beschouwen. Gerandomiseerde bossen brengen een oplossing voor de groei in de hypotheseruimte en zijn in staat om de predictieve performantie te verhogen.

- Het gebruiken van complexe aggregaten bij het bouwen van predicties in het conclusiegedeelte van een hypothese. Meerbepaald onderzoeken we het gebruik van complexe aggregaten in de lineaire vergelijkingen in de bladeren van modelbomen. Daarvoor is het eerste orde beslissingsboom leeralgoritme opgewaardeerd tot een eerste orde modelboom leeralgoritme.

- Het bepalen van een geschikte heuristiekfunctie voor het leren van modelbomen. Daarvoor zijn bestaande heuristieken bestudeerd. Deze werden allemaal ofwel te complex bevonden, ofwel te weinig naar lineaire modellen gericht. Een nieuwe heuristiek werd voorgesteld die beide tekortkomingen vermijdt.

Wat betreft verder werk noemen we eerst en vooral het automatiseren van de generatie van grenswaarden voor de aggregaatscondities. In het voorgestelde werk worden deze grenswaarden ofwel opgegeven door de gebruiker, ofwel gegenereerd aan de hand van een discretizatieprocedure. Bestaande discretizatieprocedure genereren echter grenswaarden binnen de schaal van de getallen die voorkomen in de dataset. Voor aggregaatsfuncties als *sum* of *count* schiet deze aanpak tekort.

Een ander idee voor toekomstig werk is het combineren van aggregaten en selecties op een andere manier. In dit eindwerk legden we selectiecondities op aan de verzameling van waarden die geaggregeerd wordt. Men zou ook selectiecondities kunnen opleggen aan het resultaat van de aggregaatsconditie. In een biochemische applicatie kunnen we zo condities opleggen als *het element van het atoom met maximale lading is koolstof.* In dit geval moet een aggregaatsconditie een object teruggeven en niet enkel een waarde.

Bibliografie

Alexander, W. and S. Grimshaw (1996). Treed regression. *Journal of Computational and Graphical Statistics 5*, 156–175.

Appice, A., M. Ceci, and D. Malerba (2003). Mining model trees: a multi-relational approach. In T. Horváth and A. Yamamoto (Eds.), *Proceedings of the 13th International Conference on Inductive Logic Programming*, Volume 2835 of *Lecture Notes in Artificial Intelligence*, pp. 4–21. Springer-Verlag.

Blockeel, H. and M. Bruynooghe (2003). Aggregation versus selection bias, and relational neural networks. In *IJCAI-2003 Workshop on Learning Statistical Models from Relational Data, SRL-2003, Acapulco, Mexico, August 11, 2003.*

Blockeel, H., M. Bruynooghe, S. Džeroski, J. Ramon, and J. Struyf (2002). Hierarchical multi-classification. In *Proceedings of the ACM SIGKDD 2002 Workshop on Multi-Relational Data Mining (MRDM 2002)*, pp. 21–35.

Blockeel, H. and L. De Raedt (1997). Lookahead and discretization in ILP. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, Volume 1297 of *Lecture Notes in Artificial Intelligence*, pp. 77–85. Springer-Verlag.

Blockeel, H. and L. De Raedt (1998, June). Top-down induction of first order logical decision trees. *Artificial Intelligence 101*(1-2), 285–297.

Bratko, I. (2001). *Prolog Programming for Artificial Intelligence.* Addison-Wesley. 3rd Edition.

Breiman, L. (2001). Random forests. *Machine Learning 45*(1), 5–32.

Dehaspe, L. and H. Toivonen (1999). Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery 3*(1), 7–36.

Elmasri, R. and S. Navathe (2004). *Fundamentals of Database Systems* (4nd ed.). Addison-Wesley.

Karalic, A. (1992). Employing linear regression in regression tree leaves. In *European Conference on Artificial Intelligence*, pp. 440–441.

Krogel, M.-A., S. Rawles, F. Železný, P. Flach, N. Lavrač, and S. Wrobel (2003). Comparative evaluation of approaches to propositionalization. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, Volume 2835 of *Lecture Notes in Artificial Intelligence*, pp. 194–217. Springer-Verlag.

Langley, P. (1996). *Elements of Machine Learning*. Morgan Kaufmann.

Lavrač, N. and S. Džeroski (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Lloyd, J. (1987). *Foundations of Logic Programming* (2nd ed.). Springer-Verlag.

Malerba, D., F. Esposito, M. Ceci, and A. Appice (2004). Top-down induction of model trees with regression and splitting nodes. *IEEE Transactions on Pattern Analysis and Machine Intelligence 26*(5), 612–625.

McCarthy, J., M. Minsky, N. Rochester, and C. Shannon (1955). A proposal for the dartmouth summer research project on artificial intelligence.

Muggleton, S. (Ed.) (1992). *Inductive Logic Programming*. Academic Press.

Perlich, C. and F. Provost (2003). Aggregation-based feature invention and relational concept classes. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 167–176. ACM Press.

Plotkin, G. (1969). A note on inductive generalization. *Machine Intelligence 5*, 153–163.

Quinlan, J. (1992). Learning with continuous classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pp. 343–348. World Scientific, Singapore.

Struyf, J., S. Džeroski, H. Blockeel, and A. Clare (2005). Hierarchical multi-classification with predictive clustering trees in functional genomics. In *Progress in Artificial Intelligence: 12th Portugese Conference on Artificial Intelligence*, Volume 3808 of *Lecture Notes in Computer Science*, pp. 272–283. Springer.

Wang, Y. and I. Witten (1997). Inducing model trees for continuous classes. In *Proc. of the 9th European Conf. on Machine Learning Poster Papers*, pp. 128–137.